Microprocessor System Data Transfer Interface Design: An Expert System Approach Using Signal Timing Behavioral Patterns

by

BENEDIKT THEODOR HUBER
M.Sc., University of Victoria, 1986
B.Sc., University of Victoria, 1983

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

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

We accept this dissertation as conforming
to the required standard

Dr. K. F. Li, Supervisor, Dept. of Electr. & Comp. Eng.

Dr. N. J. Dimopoulos, Member, Dept. of Electr. & Comp. Eng.

Dr. E. G. Manning, Member, Dept. of Electr. & Comp. Eng.

Dr. M. H. Van Emden, Outside Member, Dept. of Computer Science

---

Dr. A. J. Al-Khalili, External Examiner, Dept. of Electr. & Comp. Eng.,
Concordia University

Supervisor: Dr. K. F. Li

Abstract

DAME (**D**esign **A**utomation of **M**icroprocessor-based systems using an **E**xpert system approach) is an expert system for configuring and designing a customized microprocessor systems from original specifications. This work deals with the development of the data transfer interface design module in DAME: the Interface Designer.

The automated Interface Designer is developed by extracting common features, functions and behavior of microprocessor components and representing them using knowledge representation techniques. The design is accomplished through pattern matching, by performing actions and procedures based on recognition of the standard behavior patterns of microprocessor component signals.

The development of the Interface Designer production system is divided into three parts: a hierarchial network of frames that represents the components, a hierarchial network of frames that represents the interface and a set of forward chaining rules that represents the design expertise. Equivalent abstraction levels are developed for the component model, interface model and design rules, allowing the design process to proceed using a top-down methodology.

The component behavior is abstracted at several levels. At the more abstract behavior level, the data transfer behavior is divided into a set of fundamental information transfers, namely the address, data, request, direction, type, delay, size and width information transfers. At the more detailed level, each information transfer is divided into state and timing information transfers, where state information represents the conceptual meaning of the state of a signal, and the timing information specifies when the state information is usable. Finally, the timing information is represented using a set of propagation delay invariant timing patterns. Only a limited number of timing patterns is required, thus allowing a limited number of design rules to accomplish the interface design.

Interface design is carried out by sub-dividing the interface into progressively more detailed interface sub-blocks, until eventually the interface is built up from a set of parameterized primitive circuits that represents the lowest level basic building blocks of an interface. The set of primitive circuits developed gives the Interface Designer the ability to connect signals based on the timing patterns. The timing behavior of the output of the interface is determined as a function of the primitive circuit parameters and the timing behavior of the input of the interface. Once the interface design is complete, the output timing behavior of the interface is verified to assure that all component input timing constraints are satisfied.

Each of the primitive circuits developed is also given using VHDL. This allows the complete interface to be generated using VHDL code once the design is complete, permitting simulation for verification and synthesis for implementation of the interface. Several small test systems are designed and simulated to check the validity of the Interface Designer.

Examiners:

Dr. K. F. Li, Supervisor, Dept. of Electr. & Comp. Eng.

Dr. N. J. Dimopoulos, Member, Dept. of Electr. & Comp. Eng.

Dr. E. G. Manning, Member, Dept. of Electr. & Comp. Eng.

Dr. M. H. Van Emden, Outside Member, Dept. of Computer Science

Dr. A. J. Al-Khalili, External Examiner, Dept. of Electr. & Comp. Eng.,
Concordia University

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Dr. Kin. F. Li for his help and guidance throughout the course of this work. I would also like to thank NSERC for providing financial support for this research.

Glossary

ALU	Arithmetic Logic Unit
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
CAD	Computer Aided Design
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CRL	Carnegie Representation Language
CRT	Cathode Ray Tube
DAME	Design Automation of Microprocessor-based systems using an Expert system approach
DIP	Dual In-line Package
DMA	Direct Memory Access
DSP	Digital Signal Processor
EPROM	Erasable Programmable ROM
HDL	Hardware Description Language
IO	Input / Output
IB	Interface Block
ISB	Interface Sub-Block
ISBP	Interface Sub-Block Primitive
LCC	Lead-less Chip Carrier
LSI	Large Scale Integration
MSI	Medium Scale Integration
NMOS	N-Type Metal Oxide Semiconductor
omp	Order of Magnitude Propagation delay
PAL	Programmable Array Logic
PGA	Pin Grid Array
P-M-S	Program-Memory-Switch
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SSI	Small Scale Integration
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver/Transmitter

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Rationale Behind Microprocessor System Design Using an Expert System Approach

Microprocessor based systems (also called microcomputers) are designed and constructed using off-the-shelf components according to application specific requirements. The explosive growth of the range of applications for microprocessor systems, from household appliances such as microwaves to scientific instrumentation such as the Mars Rover, indicates there is a high demand for customized microprocessor system design. Despite the increasing complexity of today's 32 and 64-bit microprocessors, embedded system design has remained largely as it was 20 years ago when 8-bit microprocessors were state of the art. Some industry analysts predict a looming complexity crisis due to a lack of trained engineers and a lack of good automation tools [61], which will slow down the much heralded explosion of consumer products using sophisticated microprocessors.

The high demand for customized designs and the complexity of new components make a synthesis tool for microprocessor system design very attractive. Such a tool would allow rapid development of new products, reducing the time to market and lowering development cost. It would relieve the designer of some of the routine drudgery of a design task, while at the same time reducing the number of errors in the design since automatic design verification could be performed. It would allow a design engineer not familiar with the latest components, or a novice designer, to produce a design with those components.

The lack of a comprehensive theory of system integration and design choices has led to a more or less empirical set of rules for microprocessor system design, which an experienced system designer can draw upon to give a solution to a design problem. A synthesis tool using an expert system approach would allow the categorizing and codifying of an expert's knowledge so that a microprocessor system can be generated automatically.

1.2 Work Covered in this Dissertation

A design engineer with interface design expertise uses information provided by component data sheets and knowledge about previous microprocessor system designs to build the data transfer interface as shown in Figure 1-1. To automate the design process,

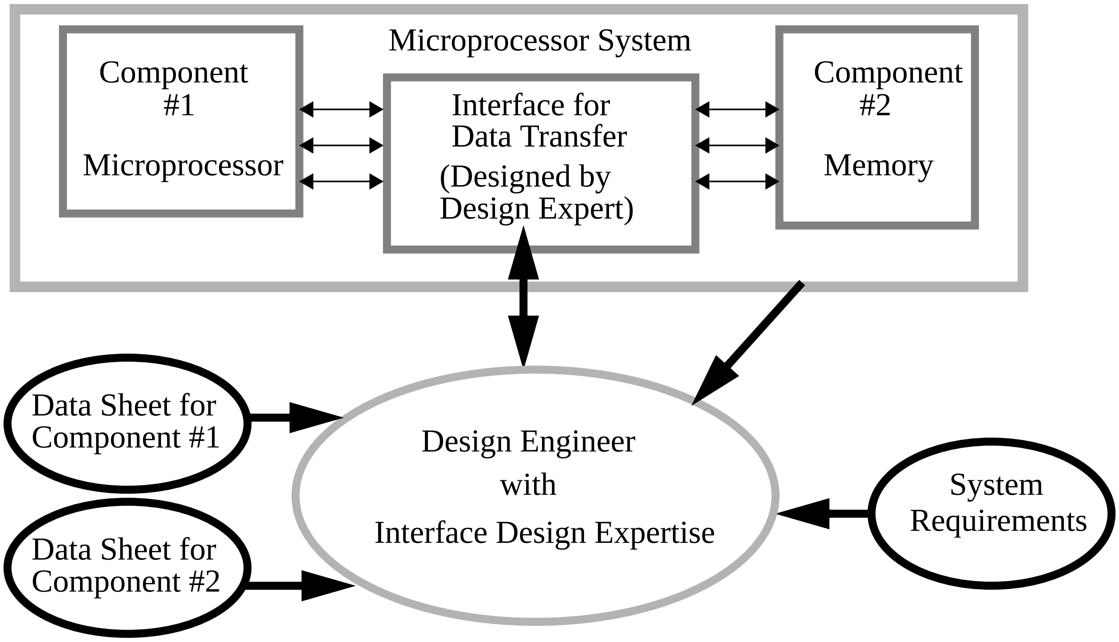

Microprocessor System
Component #1
Microprocessor
Interface for Data Transfer
(Designed by Design Expert)
Component #2
Memory
Data Sheet for Component #1
Data Sheet for Component #2
Design Engineer with Interface Design Expertise
System Requirements

FIGURE 1-1. Data Transfer Interface Design

the interface design engineer is replaced with an expert system. An *expert system* is a computer program that relies on a body of knowledge to perform a task normally performed only by a human expert.

Microprocessor system design has many aspects, from the design of the general architecture of the system, component selection, component interconnection and interface design to system implementation. To limit the scope of this work, the proof of concept expert system developed was confined to the design of the data transfer interface given a set of microprocessor system components. It is assumed that components have been selected and the overall architecture of the microprocessor system has been determined. This system is called the *Interface Designer.*

The design process is not as straight forward as it initially seems. As a human designer proceeds, she will make design decisions based on experience of previous designs and build upon hidden, underlying assumptions. The automation of the interface design developed for this work requires detailed analysis and representation of these experiences and assumptions.

To fully automate the interface design process, a functional analysis and representation of all signals involved in microprocessor interfaces is required. If a signal is present, what is its function? (Often a signal will serve several functions, even though it appears to only serve a single function). Why must it be connected? How does the signal interact with other signals to carry out the function? How can its function and interacting behavior be represented so that design automation can proceed?

Even though most of the interfaces used by the various microprocessors and related peripherals are fairly standardized, subtle variations exist [52]. Therefore a brute force approach to automated interface design, where signals having the same function are connected directly, will often fail.

This work postulates that an automated Interface Designer can be developed by extracting common features, functions and behavior of components and attaching conceptual meaning to these features through abstraction and inheritance, and representing the components using standard expert system knowledge representation techniques. Furthermore, design can be accomplished through ‘pattern matching’ by performing actions and procedures based on recognition of the standard behavior patterns.

Central to this work is the development of a limited number of representative timing patterns which can be used to represent the timing behavior of component signals, and a set of pattern matching rules used to capture the human designer’s expertise for interconnecting signals with different timing patterns using a set of pre-designed primitive circuits (elementary building blocks). The primary advantage of this approach is a reduction in the level of detail, and hence the complexity, of the design process and the information that must be modeled and represented by the Interface Designer: The level of detail needs only be sufficient to allow the pattern matching rules to select one of the pre-designed primitive circuits.

Figure 1-2 gives an overview of the interface design expert system developed for this work. The central part in the development of an expert system is the representation of the body of knowledge in a form usable by the expert system. This work is organized by dividing the body of knowledge into three distinct parts: A *component model* that represents all aspects of a component, an *interface model* that represents the interface that will be designed and the *design expertise* in the form of rules, which represents the design methodology and techniques.

Specifically this work makes the following contributions:

- It develops a set of standard timing patterns that can be used to represent the timing behavior of signals in a data transfer interface.
- It develops a set of primitive circuits that can be used to interconnect signals which have timing behavior based on the standard timing patterns.

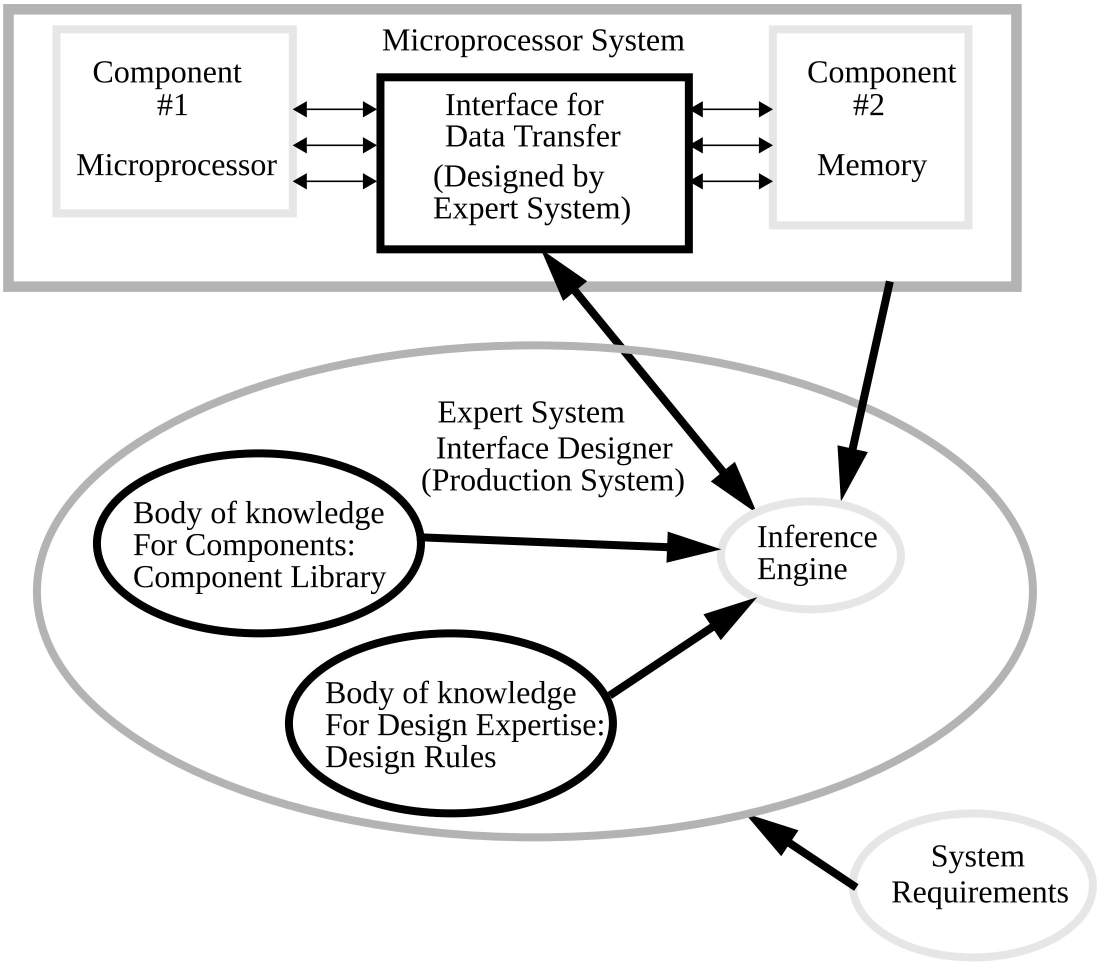

Microprocessor System
Component #1
Microprocessor
Interface for Data Transfer
(Designed by Expert System)
Component #2
Memory
Expert System
Interface Designer
(Production System)
Body of knowledge For Components: Component Library
Inference Engine
Body of knowledge For Design Expertise: Design Rules
System Requirements

FIGURE 1-2. Interface Design Expert System

- It develops a representation of the data transfer protocol in terms of information transfers, where each information transfer is based on one of the timing patterns.
- It develops a simple and complete representation of the component incorporating the standard timing patterns.
- It develops a representation of interface that will be generated.
- It develops a representation of the design expertise required for interface design in the form of rules.
- It develops a method of generating the output timing behavior of the designed interface, and it develops a technique that can be used to verify that the timing behavior of the designed interface satisfies the timing behavior of the components being connected.
- It develops a method to allow implementation and testing of the interface in real-world applications.
- It implements and tests the Interface Designer using real-world interface design examples.

1.3 Dissertation Organization

This dissertation contains eight chapters, including this introduction, followed by a Bibliography and an Appendix.

Chapter Two gives some background information for the disciplines involved in the development of an expert system for microprocessor system design: microprocessor systems, digital system design and expert systems. The chapter concludes with a description of several other microprocessor system synthesis tools that have been developed.

Chapter Three discusses the approach used to develop the automated microprocessor system designer. It first gives a simple example to illustrate some of the issues involved in microprocessor system design. It then outlines the techniques used to represent the component, the interface and the design rules.

Chapter Four develops the model for representing microprocessor system components. The model covers all aspects of a component, such as the behavior of a component, its signals and the timing relationships between signals. It presents the methods used to model the signals themselves, the different states signals can attain and the timing relationships between state changes. It develops a method of representing the protocol of the signals using information transfers based on a limited number of timing patterns.

Chapter Five presents the model for representing the interface that connects the microprocessor system components. The hierarchy of the interface model is developed from the high level interface blocks to the low level primitives which are used to eventually build up the interface. A representation of the primitives is given using VHDL to facilitate the eventual testing and implementation of the interface.

Chapter Six discusses the method used to perform interface design. The design expertise is developed in the form of pattern matching rules. The rules perform specific actions depending on the recognized patterns at the different component and interface hierarchy levels.

Chapter Seven presents the Interface Designer implemented in the Knowledge Craft expert system shell. It discusses the components entered into the Interface Designer component library. This is followed by a step by step description of a 68000 microprocessor to 6116 memory interface design example, showing some of the data structures produced, including the VHDL representation of the interface. A VHDL simulation is used to verify the correct operation of the interface. The chapter concludes with a summary of the microprocessor system design problems solved with the automated Interface Designer.

Chapter Eight provides conclusions and discusses future work.

The Appendix includes various material that supplements the main body of this dissertation.

1.4 Trademarks

Several software packages were used in the development of this work:

Knowledge Craft is a trademark of Carnegie Group Inc.

Mentor Graphics is a trademark of Mentor Graphics Corporation

QuickHDL, Qvhcom and *Qhsim* are trademarks of Mentor Graphics Corporation

XACT is a trademark of Xilinx, Inc.

UNIX is a trademark of AT & T Technologies

SunOS is a trademark of Sun Microsystems Inc.

Chapter 2

Background

This work is concerned with the automation of the design of microprocessor systems and brings together the three areas of investigation: microprocessor system design, digital system design and expert systems. This chapter provides background information for these areas. The first section presents the fundamentals of microprocessor systems and their organization. This is followed by an introduction of the digital system design techniques that are needed for microprocessor system design. The next section presents expert system and knowledge representation techniques that can be used to model the design process. The chapter concludes with an overview of other design automation systems in the literature and their relevance to this work.

2.1 Microprocessor Systems

The microcomputer era started in the early seventies after technologies had been developed to fabricate a simple 4-bit CPU, called a *microprocessor* on a single chip. A microprocessor is an entire *central processing unit* (CPU) and is useless without support circuits such as memory components, interface components, timing and control circuitry and a power supply. A *microcomputer*, also called a *microprocessor system,* is a stand alone, complete computer system capable of functioning without any additional equipment[18].

The basic microprocessor system consists of the CPU, memory in the form of *RAM* (read/write random access memory) and *ROM* (read only memory), and *IO* (input/output*)* components for external communication. Special purpose IO interfaces allow the microprocessor to receive data from input components such as keyboards and floppy disks, and to transmit data to output components such as displays and printers. If the microcomputer is a single entity that has all memory, CPU and IO included on the same chip, it is often called a *microcontroller* [65]. Microcontrollers are often limited in terms of speed, amount of memory and IO capability: thus the need to design custom microcomputer systems has not been eliminated with the introduction of microcontrollers.

In general terms a microcomputer consists of a number of modules that are linked together by a bus. A *bus* is a collection of parallel conductors designed to transfer information between separate modules within a microprocessor system. A *card* is a collection of one of more modules on one physical printed circuit board that can be inserted into a connector that has a series of signal wires that connect to a *system bus*. Although the terms *card* and *module* are sometimes used interchangeably, in this work a card represents a printed circuit board with a bus connector, whereas a module is a partition of a microprocessor system that performs as certain function in the same sense as in the concept of "modular design and modularity". A card that has several modules on it may have a connector that connects to the system bus, and may also have a *local bus* that connects the different modules on a card.

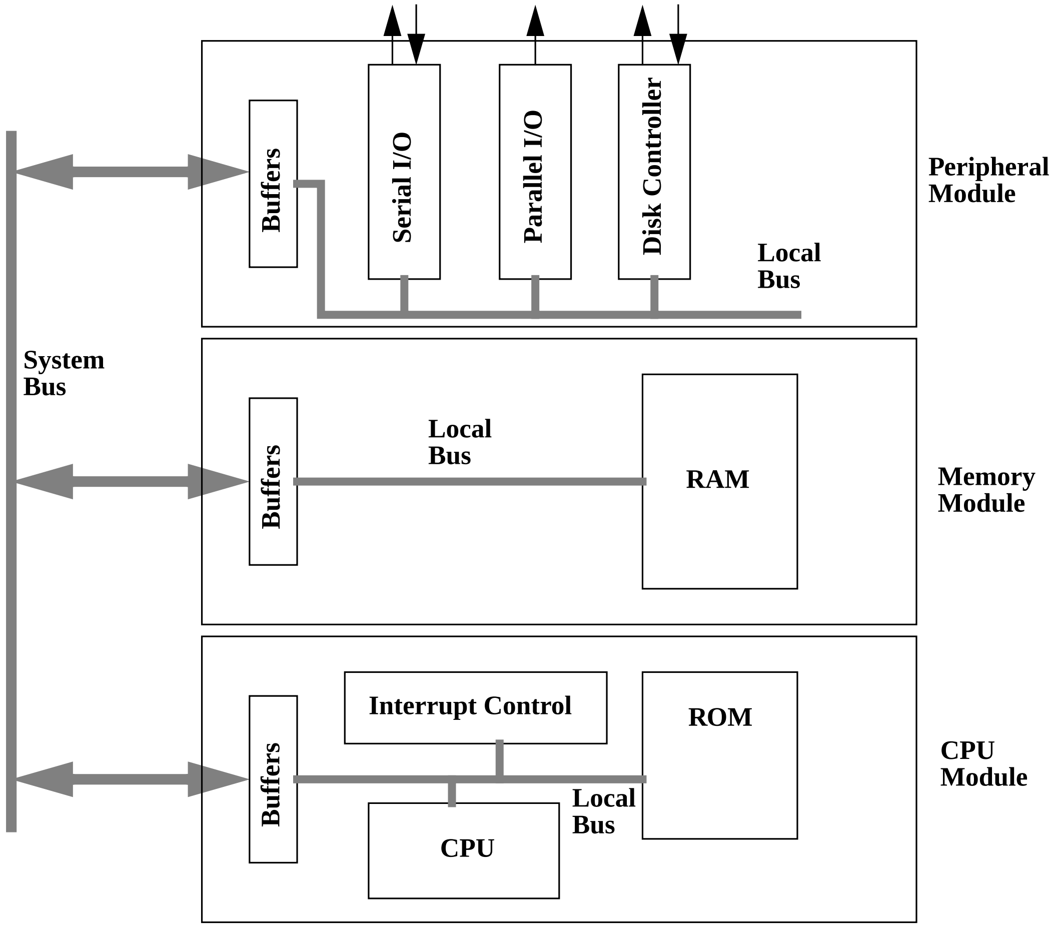


FIGURE 2-1. Block Diagram of a Simple Microcomputer

In Figure 2-1[18], the three modules could be separate printed circuit boards in which case they could be called cards, or they could be modules that all reside on a single printed circuit board, in which case the whole system would be called a *single board computer.*

All communication between components takes place over the microprocessor system bus. To facilitate error free communication, interface design requires three major considerations: purpose/function of the interface, voltage levels and current levels, and timing requirements. In the microprocessor system design literature, three types of bus are usually identified: the address bus, the data bus and the control bus [9][18][35][53][65]. A typical microprocessor uses a data bus to transfer information, and an address bus to indicate the external location where this information should be transferred. Four functions are typically provided by the control bus: memory and IO synchronization, CPU scheduling involving interrupts, bus arbitration allowing other components to use a bus, and utilities such as system clock and system reset. All microprocessors have essentially similar address and data bus structures [6][48]. The differences are usually found in the control bus and it is normally the control bus signals that make peripheral components compatible or incompatible.

With the advancement of semiconductor technology, faster and more architecturally powerful microprocessors are available every few months. For the end users however, it is often important for the new microprocessors to be both software and hardware compatible with the older components. Software backward compatibility allows the software developed for older microprocessors, a sizable investment, to be reused with the newer processors. Hardware backward compatibility allows microcomputers to be upgraded to newer, faster microprocessors by simply replacing the microprocessor chip, and it allows the reuse of peripheral expansion boards that were designed for systems using the older microprocessors.

The desire of manufacturers to provide users with hardware and software backward compatibility resulted in an evolution of microprocessor components over time [65]. The first 8-bit microprocessor, the Intel 8008, was followed by the Intel 8080 and 8085. Intel next developed the 8086 16-bit microprocessor which evolved into the 32-bit Intel 80386, 80486 and the Pentium processor. The Motorola processors follow a similar stream. The 8-bit 6800 was developed into the 16-bit 68000[1], which evolved into the 32-bit 68020, 68030 and 68040 microprocessors.

Many other processor families exist today such as the PowerPC series developed jointly by Motorola, IBM and Apple, the Alpha series developed by Digital Equipment Corporation and the SPARC series developed by Sun Microsystems. Many microprocessors were also developed for specific applications requiring certain types of arithmetic

[1] This work uses both 68000 and MC68000 when referring to the Motorola 68000 microprocessor.

operations. Microprocessors that are optimized for digital filtering and fast fourier transforms are called *DSP*s (digital signal processors). DSP components are usually optimized to perform operations such as multiply and accumulate in a single clock cycle. They often have separate memory for program and data space and are very fast when used for an application that uses the optimized operations. Such components include the Motorola 56000 and 96000 series, the Texas Instruments 32020 series and the Intel I860 series. The DSPs have similar interfaces to the general purpose microprocessors, and therefore the results of this work are directly applicable to DSP systems.

New and novel uses of microprocessors are discovered on a daily basis, requiring the design of custom microprocessor systems to fit the specific applications. The explosive growth of microcomputer applications coupled with the rapid release of new and improved microprocessor system components places a high demand on skilled custom microprocessor system design engineers. A design system that can help to reduce the cost and decrease the development time of a custom microprocessor system would be very valuable — a major motivation of this work is to build such an automated design system.

2.1.1 Microprocessor System Interface Protocols

A *signal protocol* refers to a set of conventions that describes the correct etiquette and precedence of interactions between the signals of one or more components to accomplish a specific task. When developing the Intel 8086 series (8088, 8086, 80186, 80286, 80386, 80486, etc.) and the Motorola 68000 series (68000, 68008, 68010, 68020, 68030, 68040, etc.) microprocessors, the component manufacturers made the devices hardware backward compatible in part by using similar signal protocols to move information on and off the microprocessor. Connecting two components that have an identical signal protocol is a simple process since the signals involved in the protocol can be connected directly. Unfortunately, when making a device hardware backward compatible, often only parts of the signal protocol were preserved. This resulted in subtle but important variations of the signal protocols that make interface design more difficult, since the signals often can not be connected directly.

A human interface designer can recognize and manipulate the signals, even if small variations are present in the signal protocol between components, while a simple software based automated designer that was programmed to handle only specific signal protocols may be unable to complete the design. For example, the 68000 and the 68020 both use non-multiplexed address and data buses, and a data strobe to indicate a data transfer is in progress. In both microprocessors, signals are provided to indicate that the data transfer will be completed, in the form of an acknowledge signal. For the 68000 a single `DTACK*` signal is provided, which must be used to acknowledge every data transfer, while for the 68020 the `DTACK0*` and `DTACK1*` signals are provided, one or both of which must be used to terminate the data transfer depending on which signals on the data bus are used for the data transfer. A human designer who is familiar with interface design for the 68000 would recognize that taken together, the `DTACK0*` and `DTACK1*` signals are similar to the `DTACK*` signal, and therefore can complete the 68020 interface design based on his previous experience with the 68000. One important aspect of this work is the development of expert system techniques to capture the essential features of signal protocols so that design of such systems can proceed based on the similarities between protocols.

For this work, several major families of components were analyzed, and the similarities and differences in their signal protocols were extracted. These families included the Motorola 6800 and 68000 series, the Intel 8086 series, and the Zilog Z80 series. Other microprocessors and microcontrollers were also examined to determine the similarity in their signal protocols to the above families of components. These components include the Motorola 56000, 68HC11, 6800, 6809, the Intel 8051 and the Texas Instruments 32020.

2.1.2 Microprocessor System Component Properties

Microprocessor system design requires the analysis of several aspects of microprocessor system components. These aspects include properties such as the component packaging, component power, meaning of the binary information flowing onto and off the component and the characteristics of the electrical signals that are used to send information off and onto the component. This work develops a model that allows representation of all these aspects of a component in a knowledge base.

The fragile microprocessor component die is usually embedded in a plastic or ceramic package which brings the signals to metallic leads called *pins* on the outside of the package so that they can be connected to the system through soldering or by insertion into a socket. Power is supplied to various pins on a component. LSI/VLSI microprocessor components typically require 5V to operate. Some older CMOS (Complementary Metal Oxide Semiconductor) families can tolerate voltages from 3V to 12V. The latest high speed microprocessor components (usually CMOS) sold commercially usually operate using 2.3V-3.3V power.

A Binary Digit is called a *bit* and represents a binary choice of 0 or 1. This binary choice is implemented as two voltage levels on a signal wire, a high is usually 2.3V-5V, and a low is usually 0V-0.5V. For a collection of bits, each bit usually is associated with a weight, with the most significant bit having the highest weight, and the least significant bit the lowest. The weight of the bit is [$n2^k$], where n is the symbol 0 or 1, and k is the bit position. For example, a byte has k=0 for the least significant bit and k=7 for the most significant bit.

A microprocessor communicates with the outside world through its external bus signals connected to either a local bus or a system bus. The microprocessor bus is usually divided into data, address and control buses. The information present on the buses must be interpreted with knowledge of the purpose or function of the bus. For example, the information on the address bus indicates a location in the memory space of the microprocessor, while the information on the data bus can represent a floating point number, an integer number, a CPU instruction or a text character.

Component manufacturers usually provide two types of specifications for microprocessors signals: *DC characteristics* specify DC voltages that are observed at device inputs and outputs during operation. *AC characteristics* specify the dynamic behavior of a component. AC characteristics include the rise and fall time of signals, the signal propagation delay and signal setup and hold times. The *rise and fall times* give the time taken by a signal to change voltage levels. The *propagation delay* is the amount of time taken for a change on an input signal to produce a change on an output signal. *Setup and hold times* specify the times during which a signal is not allowed to change [85].

2.1.3 Microprocessor System Components

Several different types of components are used to build up a microprocessor system. Memory components are used to store information. Memory is organized in blocks of varying size called *pages*. The description of which component occupies which page is called a *memory map*. A circuit called an *address decoder* is built to generate a signal to activate the proper memory page. The speed of memory in general is specified in terms of access time. *Access time* is usually defined as the time elapsed from the moment that a memory device is told to provide some data (i.e. the memory is *accessed*), to the moment when memory provides the data [65].

IO components have been developed to allow information input or output from the microprocessor system. These components come in many forms including analog to digital and digital to analog converters, timers, synchronous and asynchronous serial transmitters and receivers, keyboard and disk controllers. The signals used to communicate between the IO component and the microprocessor are similar to the signals used to communicate between the microprocessor and memory.

Many microprocessors families have special components that can be "attached" to the main CPU and that can perform specific tasks more efficiently than the CPU. These components are called *coprocessors*. Coprocessors are usually *tightly coupled* to the main microprocessor. Tightly coupled means the coprocessors were specifically designed to work with a specific microprocessor, having many interface signals that must be connected directly to the main microprocessor without any interface circuitry.

Additionally, manufacturers often provide some components that are needed for the design of an operational microprocessor system. These components can be divided into two classes:

1. Components required for clock generation.
2. Components required to interface the CPU and memory or IO, called bus interface circuits.

These components are usually designed to work specifically with a component and are tightly coupled to that component. One such example is the Intel 8288 bus controller that must be used with the 8086 microprocessor [41].

2.1.4 Capabilities of Microprocessor System Components

Microprocessor system components have the ability to perform operations such as moving data over the data bus signal wires, or they can respond to external stimuli such as an interrupt signal. An operation a component can perform is called a *capability* of a component. A detailed analysis of component capability is required to allow modeling of the component for an automated design system. There are three types of capabilities that are commonly found in microprocessor systems: data transfer, bus arbitration and interrupt capability. What follows is a brief description of these capabilities.

The *data transfer capability* encompasses all operations whose task it is to move some *specific* information from one component to another. This information can be data in memory, which is transferred to a microprocessor register, or data such as an interrupt vector which is transferred during a CPU interrupt procedure.

A bus is a collection of signal wires which are used to accomplish some capability, such as data transfer. Often more than one component[1] in the microprocessor system may want to use the bus for some purpose such as data transfer, and requires exclusive control

[1] 'Component' as used here refers to both single components such as microprocessors and to modules of components such as printed circuit cards containing complete sub-systems.

of all the signals on the bus. In such a case the bus must be shared between components. The ‘sharing’ process is called *bus arbitration*. If a component has the ability to share a bus, it has *bus arbitration capability.*

A microprocessor component may have the ability to be notified by an external component needing attention. The ability of a microprocessor to interrupt its current processing and execute program code that services the component needing attention is called *interrupt capability.* For interrupt capability there must be a method of altering the instruction execution path of the microprocessor, using a signal going into the microprocessor. Often the method of how the execution path is altered is done using an interrupt vector: the interrupting component supplies an indirect address (interrupt vector) pointing to the code to be executed for the specific interrupt. In such a case the interrupt vector transfer can be considered a data transfer. This shows that a capability can have other capabilities embedded within: i.e. for the example here the interrupt capability will have a data transfer capability embedded within it.

2.1.5 Microprocessor System Summary

Microprocessor systems are built up of various components such as microprocessors, RAM, ROM and IO components. Each component has well defined capabilities that allow it to perform specific operations, such as data transfer, bus arbitration and interrupt capability. The components within the microprocessor system communicate over specific system buses. Specific tasks within a capability are performed by the component’s bus signals interacting in a protocol specified by the component manufacturers.

A successful microprocessor system designer, and hence the microprocessor system design expert system, requires expertise in various areas such as microprocessor system architecture, the evolution of the different microprocessor families, the capabilities of a component and the signal protocols used to transfer information between components. The design process used to generate the functional microprocessor system uses the digital system design techniques discussed next.

2.2 Digital Systems Design

Digital systems include all types of information processing machines which are designed to store, transform and communicate information in digital form. Digital systems can be viewed and designed at different levels of abstraction from a complete system, such as a microcomputer connected to a laser printer, to the most detailed small building blocks, such as transistors, resistors, diodes and capacitors. The formal design of a digital systems involves several hierarchial tasks called design phases as shown in Figure 2-2. Each design phase is used to refine information obtained or generated at the higher abstraction levels until, at last, a completely implemented design is obtained [25].

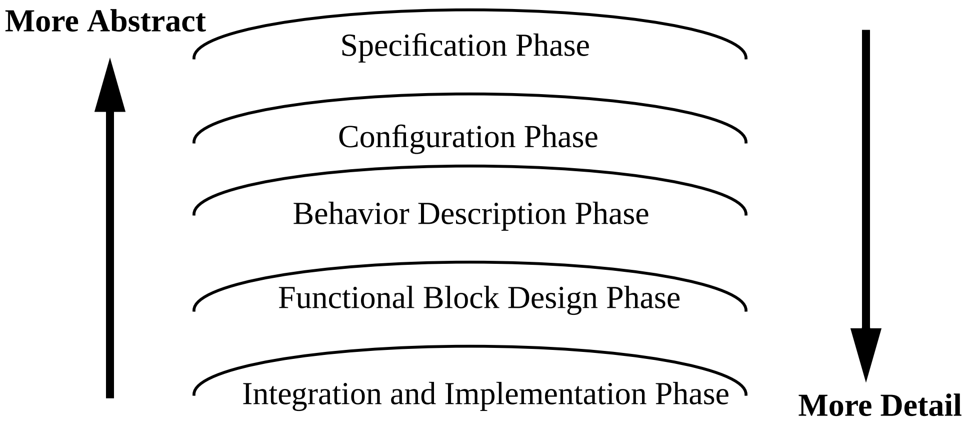

More Abstract
Specification Phase
Configuration Phase
Behavior Description Phase
Functional Block Design Phase
Integration and Implementation Phase
More Detail

FIGURE 2-2. Digital System Design Phases

During the specification phase the system responsibilities, design constraints, and operating environment are established. In the configuration phase, the system is partitioned into functional blocks, such as microprocessors for processing information, memory for storing information and IO functional blocks for communicating with the world outside the digital system. Interface requirements between functional blocks are established at this design phase in terms of the functionality of the component. In the behavior description phase, the individual functional blocks are described in more detail. Typically the bus size, speed and more precise function of each functional block are determined. During the functional block design phase an available component or group of components is selected which most closely fits the specification from the behavior description. During the integration phase of microprocessor system design, the functional blocks are connected to produce the final design. During the implementation phase, the actual digital system is built.

This work is primarily concerned with the automation of the interface design between functional blocks during the integration phase of system design. Intuition and experience play a far greater role in the design process than is generally recognized [86]. The successful digital system designer, and hence an automated digital system design expert system, must be familiar with system design techniques from circuit boards, VLSI components, MSI/LSI gates to elementary building blocks of digital system at the transistor level. Digital system design techniques are analyzed in detail in this work, to allow representation using expert system techniques.

2.3 Knowledge Based Expert Systems

A general definition of an expert system from a functional point of view can be given as: “An expert system is a [computer] program that relies on a body of knowledge to perform a somewhat difficult task usually performed only by a human expert. The principal power of an expert system is derived from the knowledge the system embodies rather than from search algorithms and specific reasoning methods. An expert system successfully deals with problems for which clear algorithmic solutions do not exist” [66]. This includes the problems of machine vision, natural language processing, pattern recognition, game playing, machine learning and system synthesis.

2.3.1 Knowledge Representation

We define knowledge as information about the world that allows an expert to make decisions [66]. *Knowledge representation* is the process of representing this information formally. Knowledge can be classified according to the degree to which fundamental principles and causal relationships are taken into account. *Shallow* knowledge is only concerned with the information required to solve a particular type of problem, while *deep* knowledge represents the internal and causal structure of a system and the relationships between its underlying components. For microprocessor system design, we must be able to represent both shallow and deep knowledge. Shallow knowledge is required to represent the overall input-output behavior of the intended system, while deep knowledge is required to represent the internal structure of the fundamental components and their interaction.

Since knowledge varies greatly in terms of content and appearance, many different knowledge representation schemes have been developed. Some of the general knowledge representation techniques include semantic networks, inclusion hierarchies, frames, schemata and production rules [79].

The term *semantic network* has been used by many different people to mean many different things [66]. The earliest definitions of semantic networks reflected the psychological models of human memory and built structures that represented the meaning of words. In general, semantic networks rely on two fundamental concepts:

- *Nodes*, which are used to represent concepts, objects or events
- *Links* (also called *relations*), that represent relationships between the nodes

For a graphical representation, relations are drawn as arrows and nodes are drawn as rectangles, ovals or boxes. The nodes in a semantic network can be given as sub-classes of other nodes using the *is-a* relation as shown in Figure 2-3. The is-a relation demonstrates

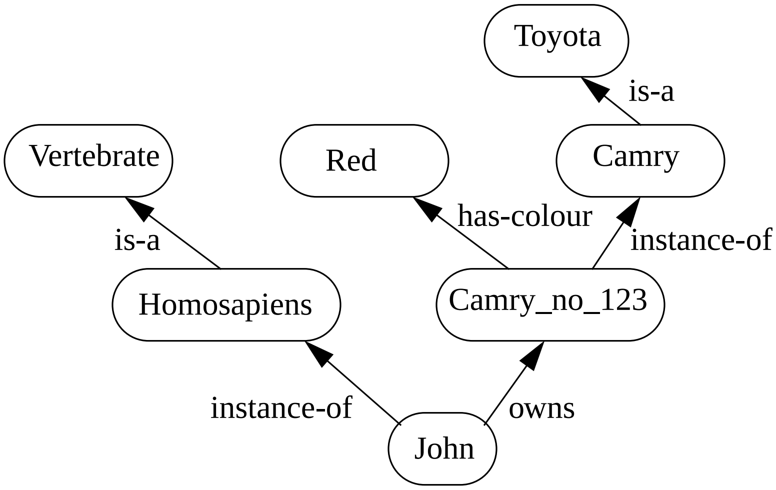


FIGURE 2-3. Semantic Network for John

the concept of *inheritance* since it links a class with its super-class, where the super-class represents a typical member of the class. The *instance-of* relation identifies a specific physical instance of a class. For example, Camry_no_123 inherits the property that it is a Toyota through the Camry super-class.

Humans' knowledge about the world seems to be often organized hierarchially by grouping items we know of into classes, superclasses and even bigger super-superclasses. An *inclusion hierarchy* represents this class structure by relating classes with the "is-a" inclusion relation. Inclusion hierarchies are important for knowledge representation since they provide a framework that allows properties from a superclass to be inherited by all its child classes. Figure 2-3 shows a graphical representation of a semantic network.

The basis for inheritance is the concept that objects or concepts form groups whose members tend to share common properties. Inheritance allows us to find information that is not stored where we look initially. This leads to what is sometimes called *cognitive economy*, where information is stored in only one place, but can still be retrieved from many places [66]. Inheritance reduces storage requirements, simplifies maintenance and provides a method for reducing the complexity of the representation of an object through abstraction and information hiding.

A *frame* is a collection of knowledge relevant to a particular object, situation or concept given in terms of attribute names called *slots* and values for the attributes called *fillers* [79]. Frames provide an effective method of organizing knowledge as simple, easily implemented data structures for information entry and retrieval. Default values and/or information associated with a slot are called *slot attachments*. Attachments can be constraints that must be satisfied by the filled in value, a procedure that can be used to determine the value of the slot (called an *if-needed* procedural attachment), or a procedure called after a slot has been filled in (called an *if-added* procedural attachment).

A frame that is associated with a class of objects or a category of situations is sometimes called a *schema* or *template frame.* A schema is a general frame that can be used as a template or plan for creating a specific instance of a frame by *instantiating* this general frame. A schema provides a simple method of representing inclusion hierarchies: each class in the inclusion hierarchy is represented by a schema frame.

The concepts of inclusion hierarchies, frames, schema and semantic networks are brought together in the *frame based semantic networks* developed for this work. In frame based semantic networks, the slots represent the relations of the inclusion hierarchy, while the frames represent the nodes. An example frame for John from Figure 2-3 is shown in Table 2-1. For this work, a specific relation will be given as ^*relation_name*, while a frame will be given as `frame_name`. For example, the frame `John` shown in Table 2-1 is an ^*instance_of* the frame `Homosapiens`.

FRAME: John **Slot**	**Filler**
instance_of	Homosapiens
owns	Camry_no_123

TABLE 2-1. Semantic Network Frame for John

A frame based semantic network was chosen for this work to represent all aspects of the components and interface. The frame based method was necessitated by the diversity and repeatability of the information and made it possible to represent the components and the interface in a hierarchial fashion.

Production rules, also sometimes called productions, are condition-action rules developed in the human modeling world. Whereas frames represent knowledge about the objects or concepts, production rules represent knowledge about how to manipulate and/or use the information found in frames. The action of a production rule specifies what the rule should do, while the condition specifies when the action should be performed.

2.3.2 Productions Systems

A *production system* is a program that consists of a series of production rules, a database of state information and a method of invoking the production rules called the inference engine as shown in Figure 2-4. Knowledge is encapsulated in both the production rules and the database of state information.

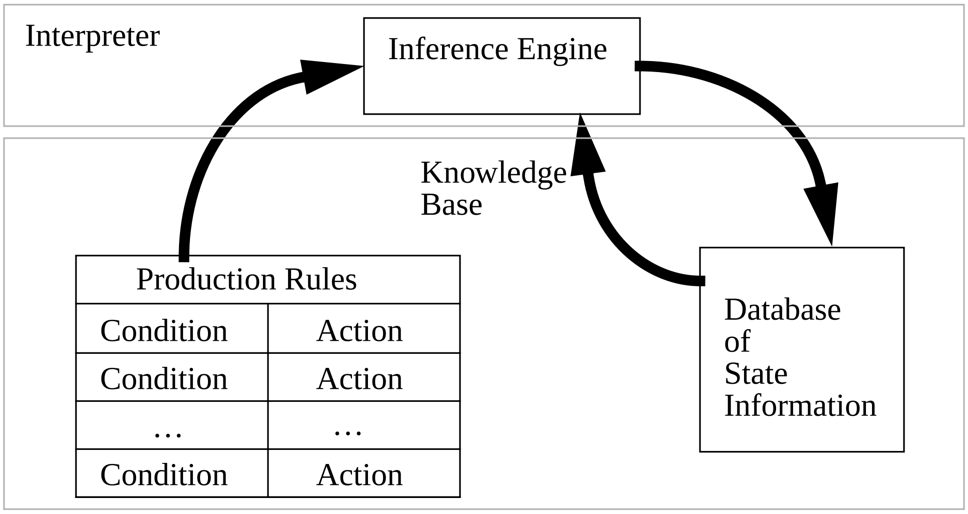


FIGURE 2-4. Structure of a Production System

The database of state information is stored in what is frequently called the *working memory*, while the production rules are stored in the *production memory*. The inference engine takes the production rules and tests if any of the conditions are satisfied by checking the database of the state information and then modifies the database of state information according to the said action. The state information is sometimes called *facts*, while the production rules are sometimes simply called *rules*. The facts can be dynamic or static: as the inference engine matches conditions and executes actions, some parts of the database of state information will be modified (dynamic), while other parts of the state information will never be modified (static).

A production system is a powerful tool that provides a reasoning process that can be used with the frame based semantic networks used in this work. Two different reasoning processes are possible with a production system, forward or backward chaining.

A rule such as “If the timing belt has cracks, then replace timing belt” can be viewed as either a *forward rule*:

If the timing belt has cracks
Then replace the timing belt

Or as a *backward rule*:

replace the timing belt
If the timing belt has cracks

For inference using the backward rule, the goal is taken as a hypothesis. A series of sub-goals are derived which are required to prove the original goal. If these new sub-goals are not immediately available in the form of facts, they are treated as new hypothesis that must be proven correct. Reasoning of this type is called *backward chaining inference* since it proceeds from the hypothesis to the data.

For inference using the forward rule, available facts are used to deduce new facts that hopefully will lead to the eventual deduction of the final goal. This is called a *forward chaining inference*. In a forward chaining production system, when all conditions in a rule are satisfied, the rule is said to be *triggered*. All rules that are triggered make up the *conflict set*. When actions of a rule are performed, it is said to have been *fired*. The determination of which of the triggered rules should be fired is called the *conflict resolution strategy.* Several conflict resolution strategies exist, such as specificity ordering (the most specific rule triggered will be fired), recency ordering (the most recently triggered rule will be fired) or context limiting (only a rule active in the current context will be fired) [87].

The choice to use forward or backward chaining inference depends on the situation. In general, if the solution space is large a forward chaining approach is more efficient, while a backward chaining system is more efficient for a more restricted solution space. Microprocessor system design has a very large solution space: a large number of different possible systems can be designed. The possible number of hypothetical solutions is too large to be checked against the available facts collected from the input specification. For the interface design application, the more efficient forward chaining inference method was therefore chosen.

2.3.3 Expert System Shells

By separating the knowledge of an expert system from the inference engine, expert system tools can be developed which provide a generic inference engine and knowledge base management functions. Such an expert system development tool is also called an *expert system shell*. The use of a commercial expert system shell allows the knowledge design engineer to focus on fundamental problems of knowledge representation and organization, and the rapid prototyping of new ideas and concepts.

The expert system shell chosen for this work is Knowledge Craft, since it was available in the laboratory and provides all the required facilities. Knowledge Craft is a sophisticated expert system shell that provides access to its knowledge engineering facilities through a graphical user interface called a *work center* [16]. The work center provides a knowledge base editor to allow easy entry of frame based semantic networks, user defined relations and production rules. It also provides access to the forward and backward chaining inference engines and includes debugging facilities that assist in the expert system development process.

2.4 Design Automation

The development of computer aided design started in the 1960s with the development of simple design programs used to assist in the layout of engineering drawings, primarily for printed circuit boards. As the evolution of CAD systems continued it was realized that the design program could actually relieve the user of some of the design decision and perform some of the design and design verification tasks automatically, and not just act as drawing aids. With the advent of microelectronics, the complexity of designs increased to such an extent that CAD with increasingly sophisticated design capability became a necessity. The manual design of integrated circuits of more than 10,000 gates has been found to be almost impossible [7]. The tremendous growth of ASIC (Application Specific Integrated Circuit) designs in the late 1980s, in the form of gate arrays and custom silicon designs, necessitated the development of automatic synthesis tools which could be used by designers inexperienced in the art of VLSI layout. The automatic synthesis tools were able to translate a design entered into a CAD schematic capture program into a PC board layout [36], while others were used for the programming of programmable logic devices using a language such as PLASM [33]. Silicon compiler tools were developed that were able to translate designs represented using hardware description languages into low level silicon designs [15].

2.4.1 High-Level Synthesis of Digital Systems

Designs can be described at various levels of abstraction detail as seen in Figure 2-5. At the top level is the *P-M-S* (Processor-Memory-Switch) system description which gives the behavior in terms of communicating processors and the structure in terms of processors, memory and switch descriptions. This level is followed by the *Instruction Set* level, also called the algorithmic level, which describes the system’s behavior in terms of input and output and the structure as memory ports and processors.

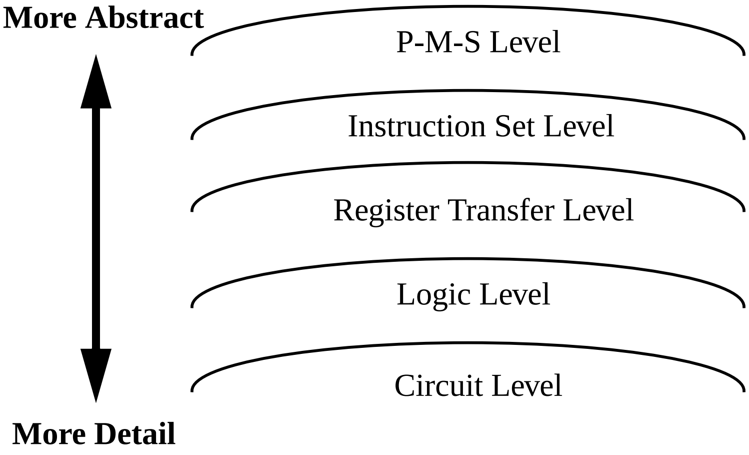


FIGURE 2-5. Abstraction Levels for Digital Systems

The next lower detail level is the *Register Transfer* level which gives the behavior in terms of information transferred between registers in the system and the structure in terms of registers, multiplexors, ALUs and buses. The next level is the *Logic* level which gives the design behavior in terms of logic equations utilizing structures such as gates and flip-flops. The bottom level is the *Circuit* level which gives the design in terms of network equations and a structure of transistors and their connections. For this work, high level synthesis refers to automated design covering all abstraction levels from the P-M-S level to Circuit level.

High level synthesis promises several advantages for system design:

- *The design cycle is shortened.*
- *The number of errors is reduced.*
- *Different design options can be considered.*
- *Documentation about the design process can be generated automatically.*
- *The number of people able to use custom IC technology is increased.*

2.4.1.1 High Level Description of Digital Circuits

To generate the interface between two components, digital design techniques have to be used. Several techniques have been developed for the automatic design of digital circuits, though not specifically for microprocessor system design. Most of these techniques work with a high level description of the digital circuits required and translate the design into a logic level description of the circuit which can be directly implemented in VLSI circuits or gate package designs. Some of the models are based on high level design description languages. For example ASP (A circuit Synthesis Program) is a system based on a high level design description language which uses both expert-system based and algorithmic methods to accomplish the design [4].

VHDL (Very high speed integrated circuit (*VHSIC*) Hardware Description Language) [1][32][2] and *VERILOG* [81] are standardized hierarchic *hardware description languages* (HDL) which can represent components from the system level, to the component level, and to the gate level.

A sophisticated hardware description language such as VHDL usually provides the following[85]:

- A method for decomposing the design hierarchially.
- A well defined interface for each design element, to allow elements to be connected to each other.
- A precise behavioral specification to allow the element to be simulated.
- A behavior specification that can be given as either an algorithm or a hardware structure to define an element's operation. This makes it possible to initially describe an element using an algorithm, and to allow higher level elements that use it to be verified. Later, once the hardware structure has been designed, the element can be replaced with the actual hardware structure.
- A method for modeling concurrency, timing and clocking of both synchronous and asynchronous structures.
- Compilers that allow hardware structures to be directly synthesized from algorithms.

2.4.1.2 High Level Synthesis of Microprocessor Systems and HDL

The design of the interface between components is one step of the microprocessor system design process. If a representation of the component interface can be generated in a HDL format, HDL synthesis tools can then be used to directly translate the interface design into hardware at the gate level. Since HDL synthesis tools can not be used to generate the HDL description of the interface itself, even if a HDL representation of the components is available [52], other techniques have to be used to design the interface and generate a HDL description of the interface.

This work develops a microprocessor interface design expert system using a rule based production system. High level synthesis languages are inconvenient to use for the state information database of such a system, since the knowledge must be represented as frame based semantic networks, which is impossible with a HDL. However, the output from the Interface Designer is best given using a HDL such as VHDL, since it then is possible to use synthesis tools to translate the interface into hardware designs using various implementation technologies.

This work uses VHDL to represent the designed interface between components. VHDL uses some unique terminology to describe circuits. A design with input and output signals is called an *entity*. The inputs and outputs to an entity are called *ports*. An *architecture* describes the function of an entity. The architecture can be given either behaviorally or structurally. A behavior description is given algorithmically using processes, which can be sequential or concurrent. A *structural* description is given using *instances* of other entities and by specifying how their ports are connected. An instance of an entity is called a *component*.

2.4.2 Expert Systems and Artificial Intelligence for Design Automation

Knowledge-based expert systems have been integrated into CAD design synthesis tools to automate the design process of VLSI systems including logic synthesis, layout synthesis, system behavior simulation, circuit behavior simulation, chip behavior simulation and so forth [15]. However most of the individual tools are not integrated with each other, requiring manual intervention at many stages of the design process. In the field of computer systems design some expert systems exist which can produce designs automatically, but they are often restricted in terms of flexibility and sophistication.

2.4.2.1 The XCON Configurer of Computer Systems

A successful commercial system for the configuration of computer systems is the rule based XCON [51] (originally called R1 before commercialization). It was developed to configure Digital Equipment Corporation’s (DEC) VAX computers. XCON takes a list of components on an order and constructs an acceptable configuration of the components by determining if any modifications have to be made to the order for reasons of system functionality. It will produce a diagram of the system layout to show how the different components will be associated. It will check for items such as correct cable length and adequate power supplies. The XCON system is not capable of performing system synthesis, only system verification. The verification in the XCON system occurs at the P-M-S level of abstraction.

Initial attempts by DEC using conventional programming languages to build a program to configure VAX computers failed due to the lack of algorithmic solutions claimed. The R1 system developed at Carnegie Mellon University in cooperation with DEC used the ‘do whenever’ style of forward chaining rules and succeeded in the task of configuring VAX systems [66].

2.4.2.2 The DEMETER Design Environment

One experimental expert system to perform system design is DEMETER [76]. DEMETER integrates a series of separately developed tools into one coherent design environment with emphasis on the highest levels of system design. It performs designs above the Register Transfer level. It provides tools to enter complete system specification, check for consistency and perform optimizations.

2.4.2.3 The MAPLE and PECOS Hardware Synthesis Systems

Two different experimental systems developed to perform microprocessor hardware design at the component level are MAPLE [77] and PECOS [77]. Both are expert hardware synthesis systems which will produce a component list from an input specification. The system interacts with the designer to produce system specifications which will facilitate the selection of chips which satisfy the design at the P-M-S level of abstraction. They have a natural language interface and are able to explain the selection of components from a library of components. These systems do not provide information about how to connect the components together.

The MAPLE and PECOS systems contain databases with information about components of microprocessor systems (memories, microprocessors and peripheral components), about pre-designed boards that can be used to assemble a system, and about reports of past designs. MAPLE emphasizes the case based reasoning approach by searching a case history database for a case matching the input specification. If none is found, it modifies a similar case to meet the input specification.

2.4.2.4 The KDMS Hardware/Software Synthesis System

The KDMS expert system is a tool under development that can be used for the integrated design of hardware and software of microprocessor systems [45]. KDMS synthesizes a system by invoking a sequence of problem solvers. Problem solvers are provided from the high abstraction P-M-S level to the detailed circuit levels. A problem solver’s responsibility is to find a path from some problem state to a solution state.

Besides synthesizing the microcomputer hardware, the KDMS system also generates a control program that directs the activity of the entire machine that has been designed, in a unified high level language. The high level language program is then translated into processor specific assembly language. KDMS uses a top down design approach to implement a microprocessor based system from the initial specification by recursively breaking the system down into a frame based set of sub-modules. When the system encounters a circuit function too specific to be realized in an existing device, the user must provide the HDL description and timing specification before proceeding.

2.4.2.5 The MICON Single Board Computer Designer

MICON [10][11] is a knowledge-based single board computer designer which can produce complete designs from original specifications. It accomplishes the low level design by connecting modules together which have compatible interface signals. Compatible interface signals are assured by manually designing a standard interface for each component in the component library, whose signals can be connected directly to other components. The predefinition of interface logic limits the flexibility of the MICON system since incompatible interface signals cannot be connected together. It also may increase the maintenance costs of the system because the interface logic must be predesigned for each new component that is entered into the library. The MICON system covers all aspects of the design abstraction levels from the P-M-S level to the circuit level.

2.4.2.6 The DAME Microprocessor System Designer

The DAME (**D**esign **A**utomation of **M**icroprocessor-based systems using and **E**xpert Systems approach) system [22] [23] [24] [25] [38] [39] [40] is aiming to produce a customized microprocessor system from original input specifications. There is an ever increasing number of components available for microprocessor system design. Often, components from different manufacturers or even components from the same manufacturer can not simply be connected directly since they have different interface specifications. None of the microprocessor design systems discussed above is capable of automatically generating the interface for two components that can not be directly connected. The automatic generation of interface logic is one of the primary goals of the DAME system, which sets it apart from other microprocessor design expert systems. This work focuses on the interface design aspect in the Integration phase of the DAME system. The complete DAME design system will eventually cover system abstraction levels from the P-M-S level to the circuit level.

One of the main difficulties in the automated design of the interface occurs for the interconnection of components that do not have identical signal protocols. Variations of the details in the signal protocols are numerous and often specific to a component. This work solves the problem at a fundamental level by analyzing and modeling the protocols of the signals used in the interface and designing an interface based on the protocols. Abstraction is employed to extract similarities between protocols, allowing a limited number of design rules to be developed that can generate the interface.

2.5 Summary

Microprocessor system design is the process of constructing a microprocessor system that satisfies a given specification. The microprocessor system design process requires domain knowledge or expertise in the architecture of microprocessor systems and their components. A microprocessor system designer must be familiar with every aspect of a component, be it the component's capabilities, packaging, power requirements, organization and interpretation of the information sent via the component's signals.

A microprocessor system is a sophisticated digital system, which requires the microprocessor system designer to have expertise about digital system design techniques from the specification and configuration phases to the integration and implementation phases.

The development of an expert system requires the storage of an expert's domain knowledge. This can be done by representing the domain knowledge as hierarchial, frame based semantic networks and production rules. A production system consisting of an inference engine, a database of state information and production rules is then used to accomplish the automated design.

Several expert systems have been developed that automate the microprocessor system design process. The main differences between the different automated design systems is the detail to which the design process is automated. The MAPLE hardware synthesis system uses information about microprocessor system components and pre-designed boards to modify a previous design found in a case history database. The MAPLE system can not design a complete system, it can only modify an existing system. The KDMS system requires the user to enter the HDL timing description of any undefined functional interface blocks. This requires the user of the system to have expertise in microprocessor design techniques. The MICON system uses standard component building blocks as templates to assemble a system starting with high level requirements. The MICON system requires the manual pre-design of a standard interface for any component that is entered into the MICON component database. This work, which is part of the DAME system, solves the problem of interface design by abstracting the often complex protocols of the signals as a limited number of timing patterns. Because of the abstraction, it is possible to develop a limited number of rules that can accomplish the interface design. The next chapter presents an overview of the Interface Designer of the DAME expert system.

Chapter 3

Interface Design Expert System Development Issues

3.1 Introduction

The goal of this work is the development of a proof of concept expert system that can automatically design an interface between microprocessor system components. This expert system is referred to as the *Interface Designer* in this work.

Interface design is the process of interconnecting several microprocessor system components using a digital circuit that enables them to operate correctly. *Correct operation* means all components in the microprocessor system operate within the specification provided by the component manufacturers. Correct operation is the primary design goal, while secondary design considerations are speed, cost, power consumption, size, weight and the time to market of the final product.

The dominant problem encountered in interface design is the large number of design possibilities, that may operate correctly or incorrectly, that exists in the design space. Furthermore, the design space is problem specific and therefore there is no guarantee that a particular design methodology will work in all cases [52]. This work emphasizes a well structured, hierarchial organization of the design space to make the complex design problem more tractable.

This chapter gives an overview of the organization of the Interface Designer. A simple example of a data transfer interface is given to put the microprocessor interface design process in perspective. Next, the general approach and methodology of the system developed are discussed. The Interface Designer's structure is divided into three parts: a *component model* representing the components to be connected, the *interface model* representing the circuitry used to connect the components and the *design knowledge* representing the design expertise to build the interface.

3.2 Data Transfer Interface Example

This section gives a simple data transfer interface design example to illustrate important concepts, the problems encountered, the issues to be considered in interface design, and how a human interface designer resolves them. A similar design problem was considered by the Interface Designer and is presented in Chapter 7.

3.2.1 The MC68000 System Interface Example

Figure 3-1 shows a typical interface between a MC68000 CPU and a memory bank made up of two MK6116 2K by 8 static RAM components [18]. The MC68000 is a

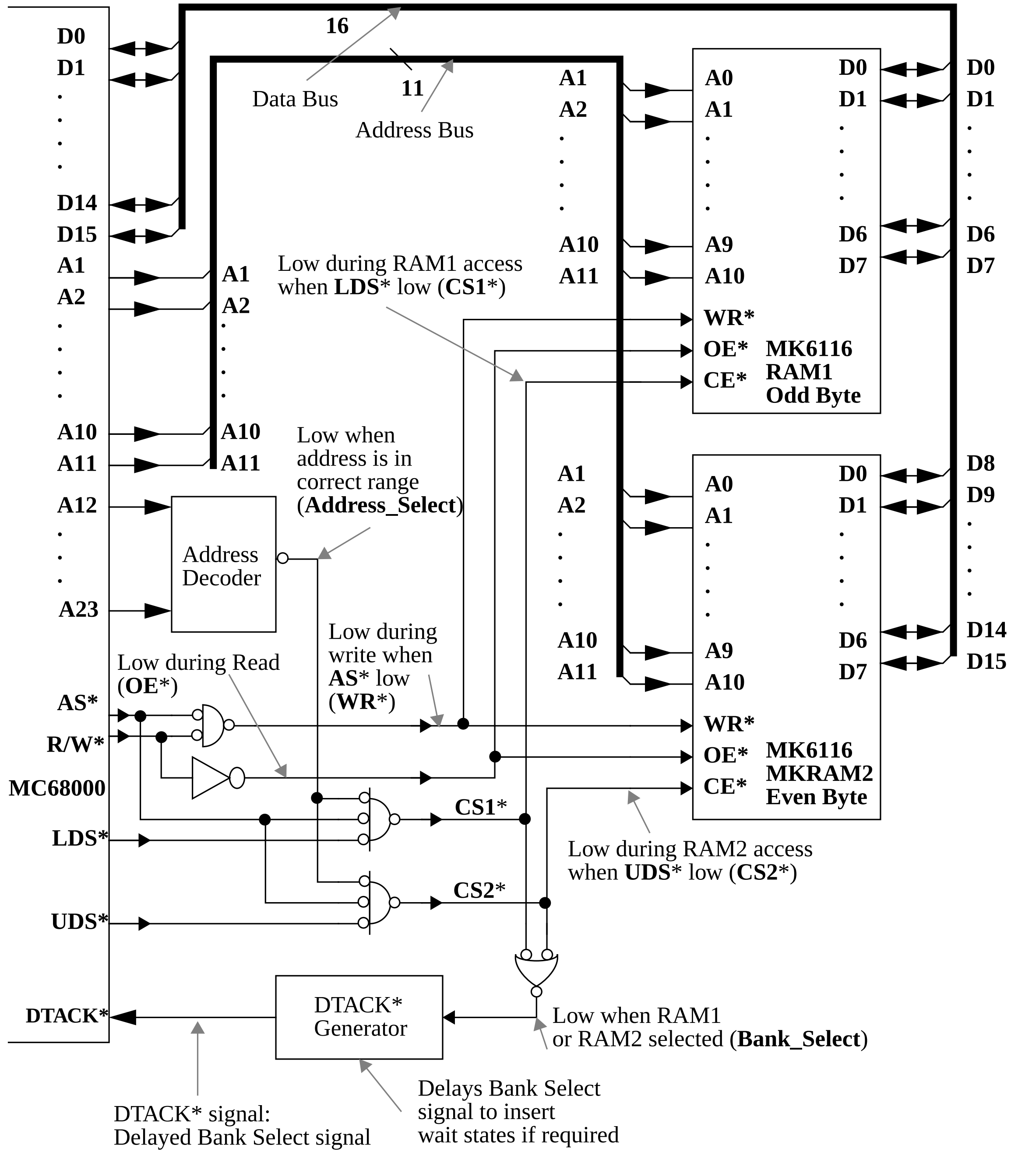


FIGURE 3-1. Interface Between MC68000 CPU and MK6116 Static RAM

microprocessor with 32-bit internal registers, but has an external 16-bit data bus (D0-

D15). The MK6116 static RAM contains 2048 internal 8-bit wide storage locations accessible over an 8-bit data path (D0-D7). The two memory components are arranged in parallel to provide 8 or 16-bit data to the MC68000.

The MC68000 provides a 23-bit address bus (A1-A23), to address individual memory locations. The AS*[1] signal is activated whenever the address signals are valid and contains usable information. The least significant address signal on the MC68000 is the A1 signal, addressing data on a 16-bit boundary. Two signals, UDS* (Upper Data Strobe) and LDS* (Lower Data Strobe) are used to indicate which 8-bit half of the 16-bit wide data path (D0-D15) is used for data transfer: an active UDS* indicates that (D8-D15) is used, while an active LDS* indicates that (D0-D7) is used. The R/W* (Read/Write) signal is used to indicate if the data transfer is a read or write operation. The DTACK* (Data Transfer Acknowledge) signal is used to terminate the data transfer cycle: data transfer is considered in progress until an active DTACK* signal is received by the MC68000.

The MK6116 provides an 11-bit address bus (A0-A10), which is used to address individual memory locations. A0 is the least significant address signal. The CE* (Chip Enable) signal must be activated whenever the MK6116 is accessed. The OE* (Output Enable) signal must be activated during a read, while the WR* (Write) signal must be activated during a write. Data is transferred over the data signals (D0-D7).

3.2.2 The Timing Diagram of the Example Components

The *timing diagram* gives the voltage state of signals as a function of time: it shows when a signal is at a high or low voltage, when the signal voltage changes or when the voltage present on a signal can be used for some purpose. For example, a timing diagram of a read data transfer cycle for the MC68000 can be seen in Figure 3-2 [18] while a timing diagram for the MK6116 data transfer read cycle is shown in Figure 3-3 [18].

Timing diagrams also provide important information related to the overall operation of a device. Important signals such as UDS*, LDS* and AS* in Figure 3-2 and CE* and OE* in Figure 3-3 are used to activate and terminate the data transfer cycle. These signals must be asserted and negated once and only once for each required operation. Any state change of these signals during a read or write cycle, even for a short period of time, is illegal and may cause malfunction of the component. A short, unwanted transition of a signal is often called a *glitch*. Figure 3-4 shows some illegal glitch transitions for the MK6116

[1] A '*' at the end of a name indicates that the signal is active low: the asserted state is represented by a signal at low voltage level.

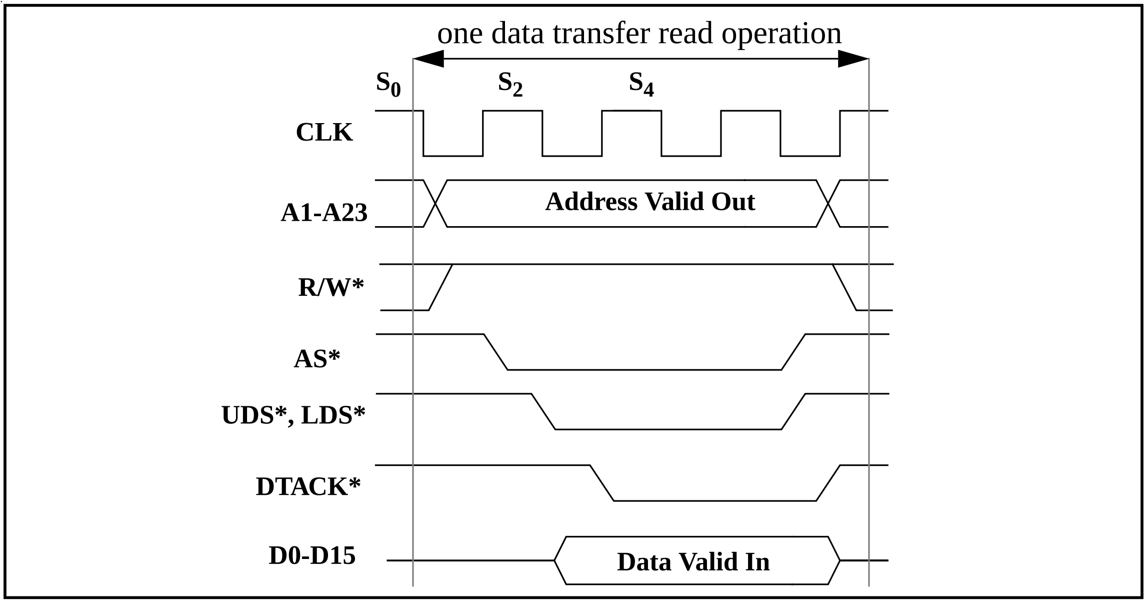


FIGURE 3-2. Timing Diagram of the MC68000 Read Cycle

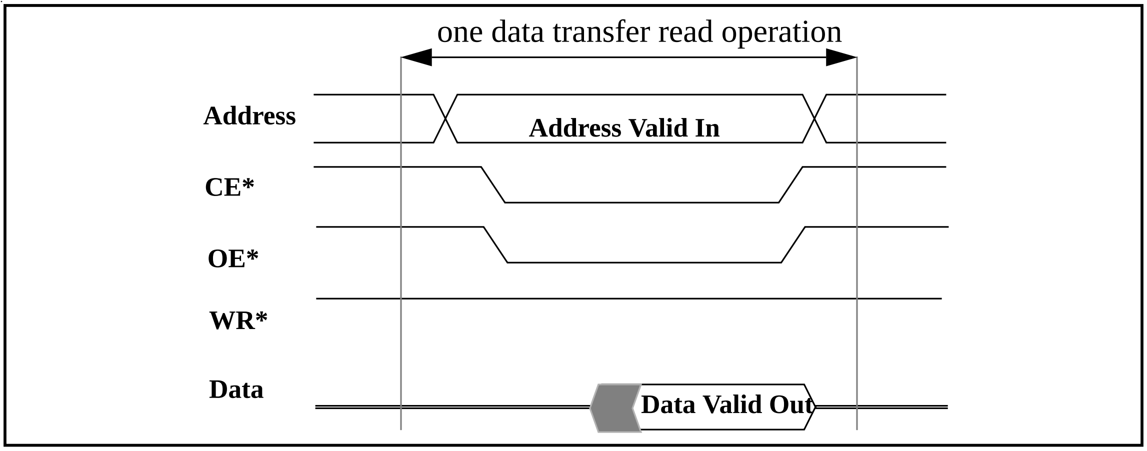


FIGURE 3-3. Timing Diagram for the MK6116 CMOS Static RAM Read Cycle

input signals that can cause device malfunction. As shown in the next sections of the detailed analysis of the example interface, human designers take several precautions to assure that the required signals are glitch free.

Timing diagrams provide knowledge about interrelationships between signals, the meaning of information found on signals and relation of signals to the overall operation of a device. For example, in Figure 3-2 the timing diagram provides information that an address is required for data transfer and when the address is valid relative to the AS*, UDS* and LDS* signals. Knowledge and understanding of the timing diagram of a component is therefore one of the most important requirements to interface design. This work

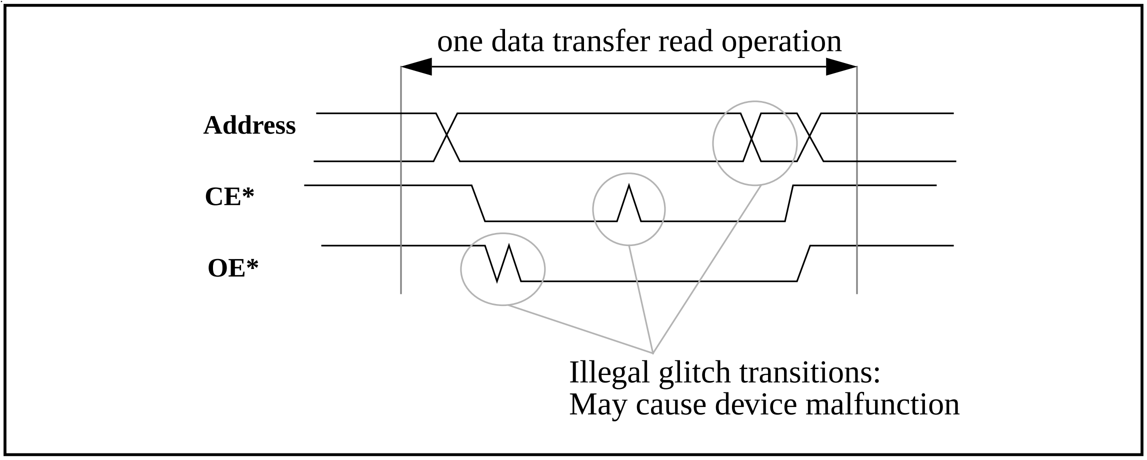

one data transfer read operation
Address
CE*
OE*
Illegal glitch transitions:
May cause device malfunction

FIGURE 3-4. Example Illegal Glitch Transitions for MK6116 CMOS Static RAM Read Cycle

develops an efficient method that encapsulates the important aspects of a timing diagram in a data structure that can be used by a pattern matching, rule based expert system.

3.2.2.1 Interface of the Address Signals

Figure 3-1 shows some of the address signals on the MC68000 (`A1-A11`) are connected to address signals on the MK6116 (`A0-A10`). Since the MK6116 has 2K locations, 11 address signals are required. `A1` of the MC68000 is connected to `A0` on the MK6116.

An address decoder is used to determine where in the MC68000's address space the memory is located. The decoder ensures that the MK6116 memory components are mapped to a specific range of locations in the MC68000's address space. The address decoder uses the (`A12-A24`) signals to produce the `Address_Select` signal, which will be asserted (low) whenever the MC68000 requires access to the MK6116 memory components. The `Address_Select` signal is then used in combination with other control signals to generate the `CS1*` (Chip Select 1) and `CS2*` (Chip Select 2) signals, which are used to enable each of the memory components.

3.2.2.2 Interface Data Signals

The interface data signals in Figure 3-1 are connected so as to allow 16-bit word access to the memory from the MC68000. This means that `D0-D7` from the odd memory byte MK6116 are connected to `D0-D7` on the MC68000, while `D0-D7` from the even byte MK6116 are connected to `D8-D15` of the MC68000.

3.2.2.3 Other Control Signals

The address decoder in Figure 3-1 generates a signal that is low when the correct address for the MK6116 RAM components is present on the address bus by decoding the high order address bits. The `Address_Select` signal is combined using a logical AND with either the `LDS*` or `UDS*` and the `AS*` address strobe to generate the `CS1*` and `CS2*` signals which are connected to the `CE*` inputs of the MK6116s. The `LDS*` and `UDS*` signals are used to indicate data transfer over the `D0-D7` and `D8-D15` signals of the MC68000 respectively. By ANDing the `LDS*` or `UDS*` signals with the `Address_Select` signal, the `CE*` signal is generated for the appropriate RAM component. From the timing diagram provided by the manufacturer of the MC68000 shown in Figure 3-2, it can be seen that `UDS*`, `LDS*` and `AS*` are activated only when the `R/W*` signal and the address signals are stable and/or valid. This means, that after ANDing the `AS*`, `LDS*` or `UDS*` and `Address_Select` signals together, the resulting `CS1*` and `CS2*` signals will be glitch free signals activating once and only once for every read or write operation.

The `CS1*` and `CS2*` signals in turn are logical ORed to generate a `Bank_Select` signal, which will be active when either of the two memory components is selected. The term *bank* is used to indicate a collection of memory components that are addressed as one block. The `Bank_Select` signal is also used to generate the `DTACK*` signal by passing it through a delay found in the DTACK Generator. A memory cycle is not terminated until the `DTACK*` signal is received. Thus by inserting a different delay, memory with different access times can be used for the design. For memory with longer access time, a longer delay will be provided.

3.2.3 Observations about the Interface Design Example

The end product of data transfer interface design is an interface similar to the one shown in Figure 3-1. This work is concerned with the automation of the design process, which requires a fundamental understanding of what needs to be done, why it must be done and how it is done. This section discusses some concepts used, the steps taken and the reasoning behind the steps taken in the data transfer interface example. The summary points in italic, following each bullet point, are provided to give the reader an overview of the types of concepts and heuristics that must be represented in the knowledge base.

- The interface serves a purpose. The objective of the interface is to transfer some specific data over the (`D0-D15`) data signals of the MC68000. All other signals in the interface are used to facilitate this data transfer.

The interface design is performed in the context of a specific purpose.

- The interface connects signals which are used to send information in and out of a component. Signals are connected after analyzing the information that is transferred over them.
  Signals and the information on signals must be represented so they can be analyzed.
- Signals are grouped according to the function of the information on them.
  Signals must be classified and grouped according to the function they perform.
- Signals are connected between components, sometimes directly, sometimes through intervening circuitry.
  A method must be provided to connect components directly through wires, if appropriate. If signals can not be directly connected, an interface circuit must be generated.
- Signals with similar function are directly connected together. For example the address signals (`A1-A11`) on the MC68000 are connected to address signals (`A0-A10`) on the MK6116s.
  Design knowledge to recognize and connect signals of similar function is required.
- Some signals must be converted to the proper format before they can be connected, such as the `R/W*` of the MC68000 signal being connected to the `OE*` signal on the MK6116. The format includes characteristics of the signal such as the polarity, and the method of encoding information on the signals.
  A method to represent the information signal format is required.
  Design knowledge that enables design of interface circuitry to convert a signal to the proper polarity is required.
- Some signals must be ‘conditioned’ with other signals before they become usable by another component. The term ‘conditioned’ refers to combining two signals in a boolean logic operation to produce a third signal. An example of this is the `R/W*` signal being ANDed with the `AS*` signal to generate the required glitch free `WR*` signal on the MK6116. The selection of signals for conditioning requires the detailed analysis of the timing diagrams.
  Design knowledge of how to generate clean, glitch free signals is required.
  A representation of timing diagrams is required.
- A component that connects to a microprocessor will have a select input (MK6116 `CE*` in the example) which will be asserted only when the component should be active. The design engineer must consider the conditions under which the component should be activated, where in the address space the component is located, what type of data transfer the component should respond to, and which part of the data bus the component connects to. The designer will often generate a single signal for each condition. All of these signals are then ANDed together to obtain the resulting select signal.
  Design knowledge on how to activate a unique component is required.
  Design knowledge to generate a signal for each activation condition is required.
  Design knowledge is required to produce a single signal from multiple signals.

- Components are placed in the address space of the processor by decoding the address into a signal such as Address_Select, which is then used in the generation of a signal that activates the corresponding device.
  Design knowledge for generating an address select signal is required.
- A component can have signals that indicate which part of the data bus will be used for data transfer. In the example these signals are the LDS* and UDS* data strobe signals.
  Design knowledge about how to activate and use the correct data signals is required.
- Some signals are multi-purpose. The LDS* and UDS* signals in the design example carry information about both the missing MC68000 A0 address bit, about how wide the data transfer is (8-bit or 16-bit) and about when the data transfer occurs.
  A method for representing and utilizing multi-purpose signals is required.
- Different signals from different functions may be combined in some fashion to generate new signals. For example the address select signal is ANDed with the LDS* and UDS* signals to generate the CS1* and CS2* signals, which drive the CE* signals on the RAM components. The CS1* and CS2* signals are combined using an OR function to generate the Bank_Select signal.
  Design knowledge about why and how to combine signals must be provided.
- A method may be provided to adjust the time allowed for the completion of the data transfer. This makes it possible to use memory devices with different access times. In the example this is done using the DTACK* input signal on the MC68000. The DTACK generator produces a delayed Bank_Select signal which terminates the data transfer after a certain time interval has elapsed.
  Design knowledge about how to change the time to completion of data transfer is required.
- Signals may be generated that will only be used internal to the interface. For example, the Bank_Select signal in Figure 3-1 is generated but does not connect directly to either the MC68000 or the MK6116. Such a signal is called an *internal signal*.
  A method for generating internal signals is required.
  Design knowledge on how to generate and use the correct internal signals is required.

By analyzing many microprocessor components, knowledge representations have been developed for this work in the form of the component model, the interface model and design rules given in later chapters.

This work emphasizes the reduction of the complex design and data representation problem through abstraction. The next section gives an overview of the approach used for the development of the Interface Designer and explains how abstraction is used to reduce the complexity of the design problem.

3.3 Approach Used for Development of the Design Automation System

This section gives an overview of the approach and methodology used in the development of the Interface Designer.

3.3.1 Imitating a Human Designer

Humans are in general better in grasping the overall structure of designs and coming up with high level strategies to solve the problem than in systematically working through a series of detailed steps in an algorithmic methodology. One of the goals of this work is to give the synthesis system the ability to perform higher level reasoning and make design decisions that are based on human experts' knowledge through the recognition of patterns. Sometimes new goals, constraints or other conditions emerge only as the design proceeds, requiring human intervention in the design process [52]. This work develops techniques that allow the human expert's knowledge to be captured in the database so that the design can be completed without human intervention.

3.3.2 Partitioning of the Interface Design System Knowledge

The Interface Designer is structured as a production system as shown in Figure 3-5. The representation of knowledge is divided into three parts: the component library (the *component model*), the interface data structures (the *interface model*), and design knowledge in the form of production rules (*design rules*).

The system state database contains knowledge about the components that are being connected and knowledge about the interface connecting the component. The knowledge about the components is *static*: it is supplied by the manufacturer of the component and is stored in a library. The knowledge about the interface is *dynamic*: the interface data structures are created and modified during the execution of the Interface Designer production system. The inference engine builds up the interface data structure using the production rules and data from the component library.

3.3.3 Abstraction of the Design Knowledge Representation

The description or specification of a system where some aspects are emphasized, while others are suppressed, is called *abstraction*. A good abstraction emphasizes details that are significant to the task at hand, while it suppresses those details that are insignificant or immaterial [72]. At the highest abstraction level, only general aspects are given, while at the lower levels more and more details are provided. The design rules are abstracted in a similar fashion: the more abstract level design rules are concerned with the

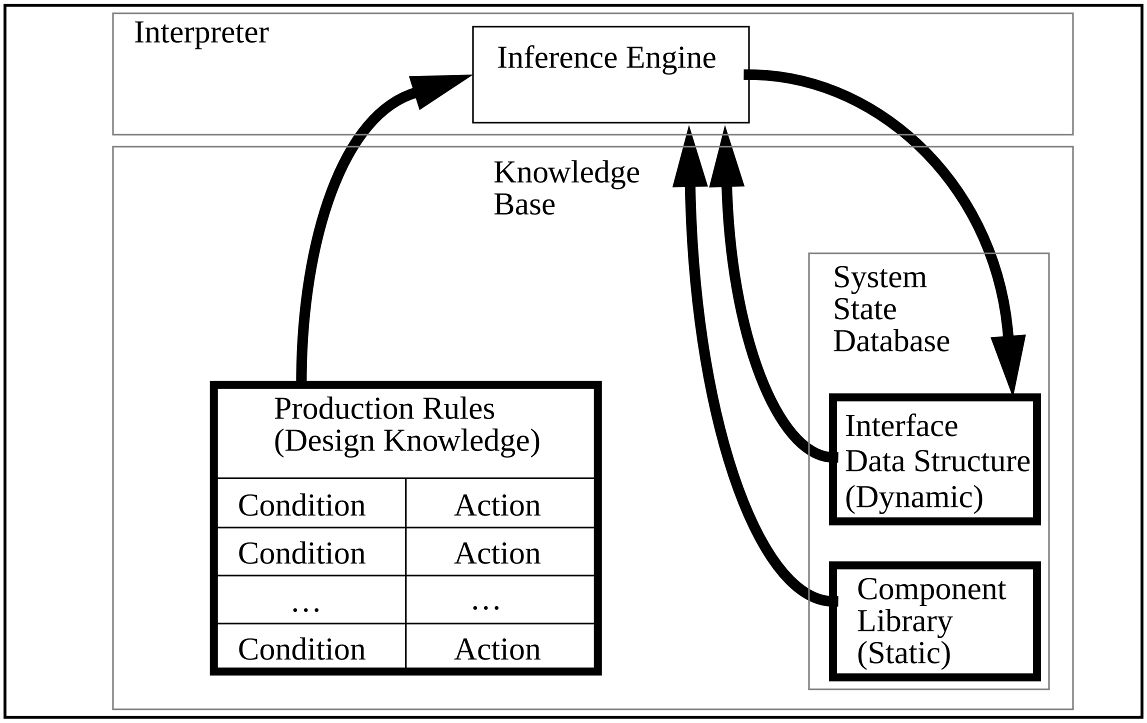


FIGURE 3-5. Structure of the Interface Designer

general aspects of a design while the more detailed level design rules accomplish the more detailed specific tasks.

The abstraction levels developed for the component model, the interface model, and the design rules follow each other closely. For each abstraction level in the component model there is a corresponding abstraction level in the interface model and corresponding design rules to accomplish design at that level.

Abstraction of the interface design process is achieved through limiting the context of the design rules: the condition of each design rule includes a test for the current design level. Only if the current design level is active, can the rule be fired.

3.3.4 Design Based on Recognizable Patterns

A *pattern* is the configuration, behavior or other feature characterizing observable system properties [75]. From analyzing interfaces designed by design engineers it was found that some of the designs were accomplished by recognizing patterns and performing certain actions according to the recognized patterns. For example, a designer might recognize that component A and component B both have signals whose function is to convey the address of a data transfer, and therefore decides to connect them to each other. The designer then looks at the timing diagram of the address signals from the manufacturer's data sheet and recognizes that address from one component is multiplexed, while the other is non-multiplexed. The designer therefore decides to insert a latch which converts the multiplexed signal to a non-multiplexed signal. This example also illustrates abstraction in the design process: the recognition that an address exists is performed at a higher abstraction level than the recognition of the address signal timing behavior as multiplexed/non-multiplexed.

Rule based production systems are powerful tools that are capable of pattern recognition and can be used to perform design provided that the relevant patterns can be represented in appropriate data structures. A large part of this work is concerned with the development of a method for representing the design knowledge using pattern based data structures and rules that can manipulate these structures.

Frames allow for the organizing, describing, relating and constraining of knowledge and therefore provide a method to represent typical features or patterns. Since they also provide a method for inheritance, data abstraction and information hiding, frames were chosen as the primary knowledge representation method for the Interface Designer.

3.4 Representing Components and their Behavior

The behavior of a component is the way it acts, reacts or functions under particular circumstances. A component model has been created that can represent the behavior as a set of hierarchial frame based objects. At the higher abstraction levels the behavior describes operations a component can perform at the system level, such as transferring data or interrupting the processor, while at the lower abstraction levels, the operations are broken down into detailed description of the signal behavior involved in the operation, such as signal A changes state 10nsec before signal B changes state. This section presents an overview of the component model. The details are provided in Chapter 4.

3.4.1 Modelling Capabilities of Components

A component’s ability to perform certain operations at the system level, is called the *capability* of a component. For this work, capabilities have been classified into two types. Type I capabilities accomplish identifiable tasks that can be found in components of different families and by different manufacturers. For a Type I capability the interface between the components is not predetermined: the manufacturer specifies how the signals behave, not which signals must be connected together. The system designer must decide which signals should be connected and what interface circuitry must be inserted between signals.

The common Type I capability classes of a microprocessor component are interrupt capability, bus arbitration capability and data transfer capability.

Type II capabilities accomplish tasks that are more difficult to identify and are specific to a set of components. For example it is difficult to describe the task accomplished when connecting the 8288 bus controller and the 8086 microprocessor. The 8288 bus controller was made to be used specifically with the 8086 microprocessor, and the interface between them is fixed (also called *tightly coupled*): The manufacturer specifies which electrical signals must be connected together between the components. There is no choice or flexibility in designing the interface. Interface design for a Type II capability is straight forward and can be accomplished using a simple procedure that directly connects each signal.

This work develops an expert system that can design Type I capability interfaces. A model was developed to represent components with Type I capabilities as hierarchial data structures. The behavior of a capability is given in terms of its protocol. Specifically, the component model developed abstracts the protocol of a capability into several hierarchial levels, where the higher levels give the protocol in abstract, general terms, while the lower abstraction levels reveal more detail.

3.4.2 Modelling the Capability Protocol

Each system component will carry out the task of the capability by communicating various information over signal wires. The communication of information over signal wires is called an *information transfer.* Each of these information transfers will accomplish one specific task associated with the capability. The interaction of the different information transfers used to carry out the task of the capability is called the *capability protocol*.

All information transfers involved in a capability are classified according to their function. For example the passing of an address from one device to another is called an address information transfer. By studying many different microprocessor components, a set of information transfer classes was developed that can be used to represent the data transfer capability of most microprocessor system components.

3.4.2.1 Synchronizing the Protocols between Components

It was found that the function of one of the information transfers was to indicate when a protocol starts. For example a data transfer protocol will have a signal indicating when the data transfer commences and bus arbitration will have a signal indicating when a bus is requested. The start information is used to synchronize all information transfers between components: all information transfers will take place relative to the start information. Similarly, there will be an information transfer indicating the end of a capability protocol. The data transfer capability protocol developed for this work will have specific information transfers indicating the start and the end of a protocol.

3.4.2.2 Overall Control of a Capability Protocol

The start and end information transfers are used to synchronize the protocols between components. The method of determining the time between the start and end information transfers is called the overall control of a capability protocol. If the time is fixed, it is called *synchronous overall control,* if the time can be changed through the use of another information transfer, it is called *asynchronous overall control*.

3.4.3 Modelling Information Transfers

The information transferred between components can be divided into two parts: the information that is embedded in the states of signals, called *state information*, and information that indicates when the information transfer takes place relative to a time reference, called *timing information*.

An information transfer is normally associated with a time reference signal and signals with state information as shown in Figure 3-6. The transition on the time reference signal is used as a time reference. The signals with the state information will hold useful information for some time interval relative to the transition. Some components may use more than one signal and more than one transition as the time reference.

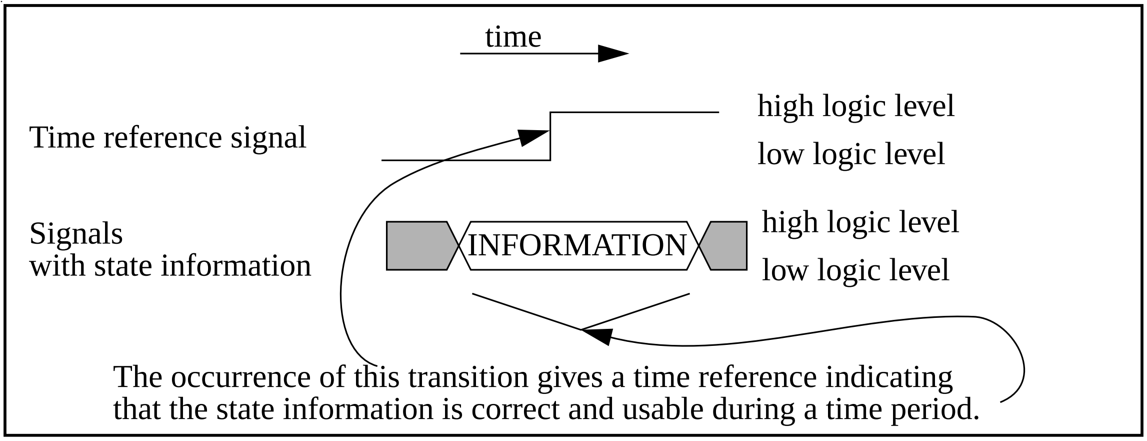

time
Time reference signal
high logic level
low logic level
Signals
with state information
INFORMATION
high logic level
low logic level
The occurrence of this transition gives a time reference indicating
that the state information is correct and usable during a time period.

FIGURE 3-6. Information Embedded in the State of Signals and its Time Reference

These states are usually the voltage level of the signal (e.g. 5V for high logic level or 0V for low logic level). Information is embedded in the state of signals, and some kind of mapping is required from the physical manifestation of the state to the meaning of the state. The representation for the state information of a signal developed for this work is in the form of a list of states and their associated meaning. For example, the `R/W*` signal of the MC68000 microprocessor is used for the direction information, which indicates whether a read or a write operation is being performed. The state information representation will associate a logic 1 on the `R/W*` signal with a read operation and a logic 0 with a write operation.

The timing behaviors of many microprocessor components were investigated and it was found that many variations of the relationship between the time reference and the state signals exist. The essential features and behavior important to modeling timing information were extracted from the timing diagrams of many microprocessor components. The result was a set of universal timing patterns, called *timing templates* which are used to represent the timing information for data transfer of any microprocessor component, which will be discussed in Chapter 4.

3.5 Representing the Interface

This work develops an expert system that can design the digital system which comprises the interface between components. The expert system requires a representation of the interface digital system which will be built up during the interface design process. An interface model was developed for this purpose, which represents the interface between components as a set of hierarchial objects. This section presents an overview of the interface model. The details are provided in Chapter 5.

3.5.1 Partitioning the Interface

One common approach to represent a digital system such as the data transfer interface is to partition it into more tractable pieces called sub-systems [71] as shown in Figure 3-7. The term ‘more tractable’ refers to sub-systems that are less complex than the entire system they make up and therefore are easier to design [45].

This work takes the approach of partitioning the interface digital system into subsystems which can be designed from components we are familiar with. A familiar component for digital design might be a Flip Flop or an AND gate, and they are considered *primitive* since they can not be further sub-divided. For more complex systems, the partitioning will have a number of layers. This means that the digital system is first partitioned into

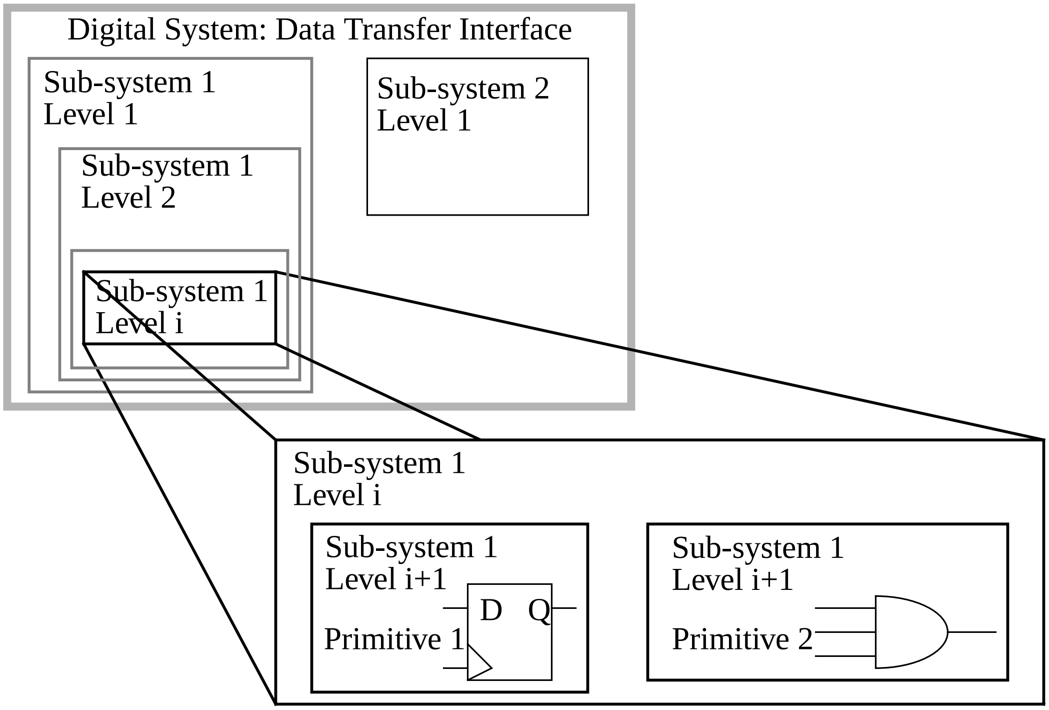


FIGURE 3-7. Partitioning a Digital Systems into Sub-systems

sub-systems, and then those sub-systems are partitioned into smaller sub-systems, until the sub-system can be readily designed out of simple primitive components at the bottom most layer. This type of design structure is often called top down design and integrates well with the hierarchial component model developed.

3.5.2 Hierarchy of the Interface Digital System

Emphasis was placed in the development of the abstraction hierarchy of the interface model to follow the abstraction levels developed for the component model hierarchy. This provides a significant advantage when developing the design rules since it allows the design process to be organized to proceed at the same hierarchial levels.

The interface abstraction layers are shown in Figure 3-8. An *interface block* (IB) connects two components with capability x. An IB is sub-divided into *Interface Sub-Blocks* (ISB). Specifically each information transfer of x, such as information Y, is connected by an *information connection ISB* within the IB. The Information Connection ISB is divided into *State ISBs* and *Timing ISBs* for connecting the state and timing information of Y, respectively.

The ISB primitives (ISBP) are the elementary building blocks that can be used to build up an interface. The ISBPs are classified into two groups according to how they are

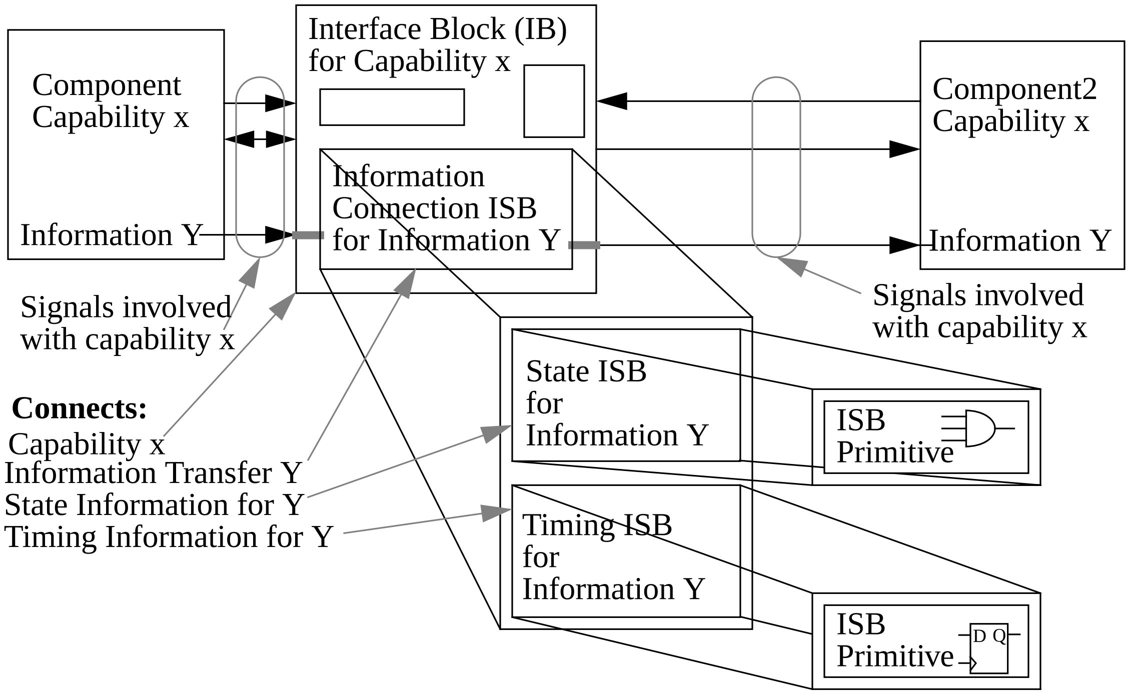


FIGURE 3-8. Interface Hierarchy

commonly used in the interface: combinatorial and memory ISBPs. A combinatorial ISBP takes the states of signals and changes them to some other state. A combinatorial ISBP is a combinatorial circuit which implements a boolean function, such as a AND gate. A memory ISBP is any digital circuit that delays a signal going into it. For example, it can be a delay line, which physically delays a signal by passing it through a long wire, or it could be a flip-flop that delays an output transition until a clock signal is received.

3.6 Representing the Interface Design Knowledge

The design process uses a top down approach: an interface is successively refined until it is completely built up from ISBPs. The organization and hierarchy of the component representation and the interface representation lend themselves naturally to top down design using a rule based production system. The current interface status and the component library are part of the system state database. Rules matched on the contents of the state database are used to modify the contents of the state database to successively refine the interface until it is complete. This section presents an overview of the design knowledge required for interface design. The details are given in Chapter 6.

The Interface Designer is activated by an externally supplied *connection request*. The connection request represents the knowledge about what components must be connected, what the purpose of the connection is (i.e. the fact that it will be the data transfer connection) and various parameters that will be relevant for a connection. The generation of the connection request is outside the scope of this dissertation and it is assumed to be provided either manually by a design engineer or by another sub-system of the DAME expert design system.

Rules are provided for the various levels of design. At the highest level, rules are provided to check the existence of the required capability of the components being connected and if both components have the required capability, an IB will be created for the capability.

Knowledge is provided about how to connect a capability in terms of the information transfers that make up the capability protocol. The knowledge is in the form of rules that indicate what to do with each information, or what to do if the information is not present.

Rules are provided for sub-dividing the Information Connection ISBs into State and Timing ISBs for connecting the state information and timing information. The rules developed are independent of the class of information being connected allowing the same set of rules to be used by all classes.

Knowledge is provided to fill in a State ISB with the appropriate ISBP. The ISBP can be anything from a simple inverter to a multiple input/multiple output combinatorial circuit. Knowledge is provided in the form of rules that can utilize information about the states going into, and the required state on the output of the Information Connection ISB.

Timing information is connected using a Timing ISB. Knowledge at this level comes in the form of rules that can recognize signal behavior in relation to a time reference and that can make adjustments to the signal so that it has a different relationship to the time reference, if necessary. The adjustment is accomplished by inserting an appropriate memory ISBP into the Timing ISB.

Other design knowledge in the form of rules is represented by the Interface Designer. This knowledge includes heuristics about minimization/maximization of all aspect of the design, such as cost, power consumption or system speed and it includes checking a design for completeness to make sure that all signals are connected and that all ISBs are completed. This knowledge also includes rules to verify timing behavior of the interface to assure that the final design meets the manufacturer’s specification for the component.

3.7 Frame Representation of the Components and Interface

The data structures used to represent the components and the interface are broken down into a set of frame based hierarchial objects. This section introduces the prototype, device and instance frames that were developed to represent a component and interface at different levels of abstraction.

A component or interface is built up from a set of *device frames*. All frames will have slots that are filled either with static values, such as numbers, or the names of other frames. For example, Figure 3-9 shows some device frames for the hypothetical X2000 microprocessor. The X2000 component is represented by the device frame named `X2000`. The *^number-of-pins* slot is filled with the number of pins (40), and the *^has-capability* slot is filled with the name of the device frame representing the data transfer capability of the X2000, namely `X2000-DATA-TRANSFER-CAPABILITY`. The data transfer capability in turn has the *^uses-timing* slot filled with the `X2000-ADDRESS-TIMING` device frame name.

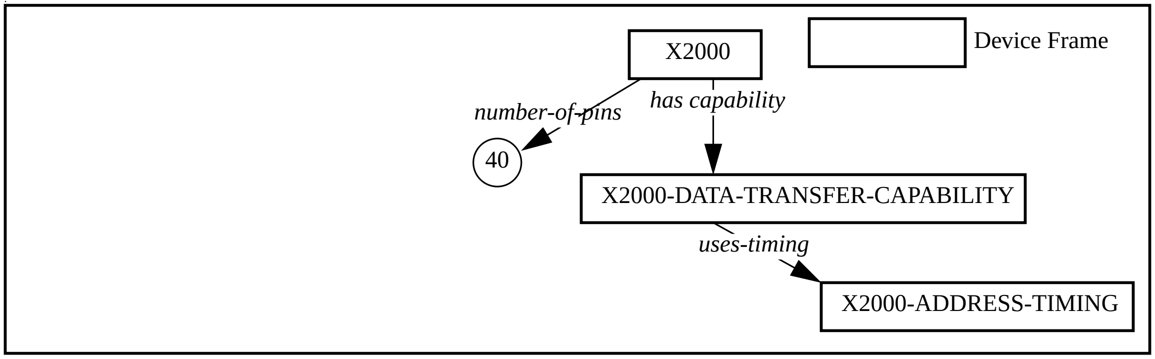


FIGURE 3-9. X2000 Device Frames

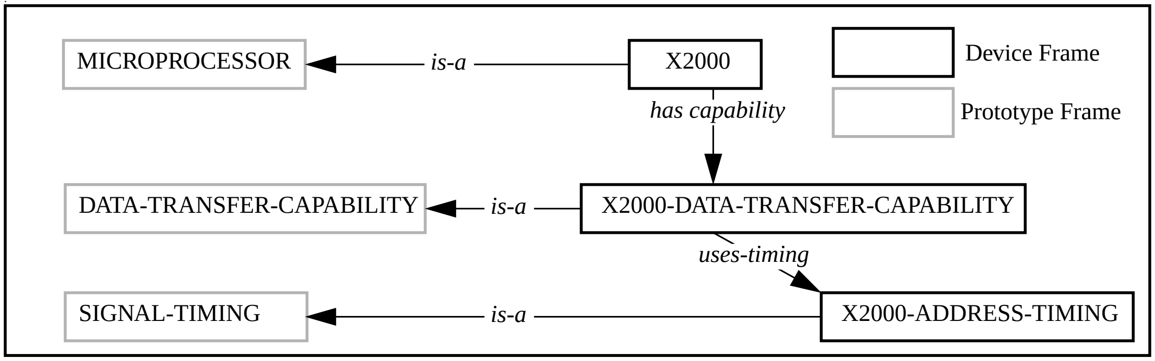


FIGURE 3-10. Device and Prototype Frames

A device frame is created by generating an instance of a a template frame, called a *prototype frame,* as shown in Figure 3-10. Each device frame is linked to its prototype using the ^*is-a* relation and will inherit characteristics from its prototype frame. A device frame will have the same slots as its prototype frame. The slots of the prototype frame are either empty or filled with default data.

The prototype frames are organized into a hierarchy of classes and sub-classes. This allows knowledge relating to components to be organized into prototype frame classes whose members tend to share common properties and concepts. Inheritance allows us to find information (knowledge) about a prototype frame from its parent frames. This leads to an efficient knowledge representation method since information that is common to several child prototype frames can be stored in a single place in the parent prototype frame. The more abstract, general prototype frames are near the top and the more detailed and specific prototype frames are near the bottom of the hierarchy as shown in Figure 3-11. A prototype frame will inherit all slots from its parent through an ^*is-a* relation. In Figure 3-11, the `COMPONENT` prototype representing any microprocessor system component is divided into `MICROPROCESSOR` and `MEMORY` component sub-classes. In turn the `MEMORY` components are divided into `RAM` and `ROM` component sub-classes.

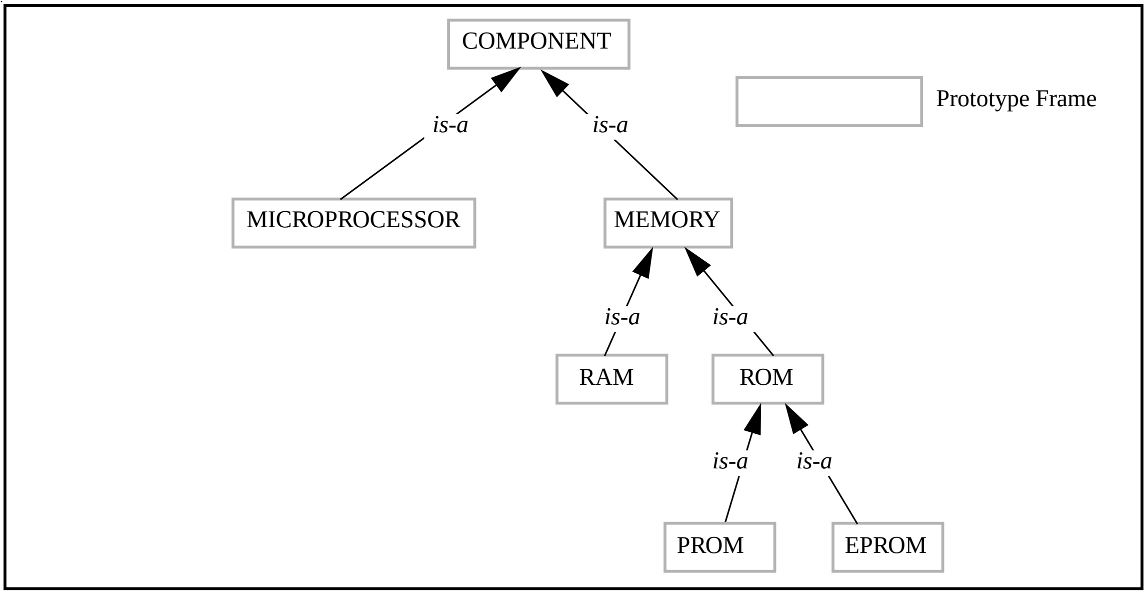


FIGURE 3-11. Prototype Hierarchy

A more complete example of device and prototype frames is shown in Figure 3-12. The MC`68000` and `Z80` device frames are created by instantiating the `MICROPROCESSOR` prototype, the `MK6116` device is created by instantiating the `RAM` prototype frame and the MK2764 is created by instantiating the EPROM prototype frame. All four devices, the MC68000, Z8000, MK6116 and MK2764, are considered sub-classes of the COMPONENT prototype frame and inherit all COMPONENT properties (such as ‘Uses Power’) through the ^*is-a* relations. Device properties can be specified in the device frame, or they can be inherited from the prototype default values. For example, the MK6116 is a 24 signal pin device, which is the default specified in the ^*number-of-pins* slot of the COMPONENT prototype frame.

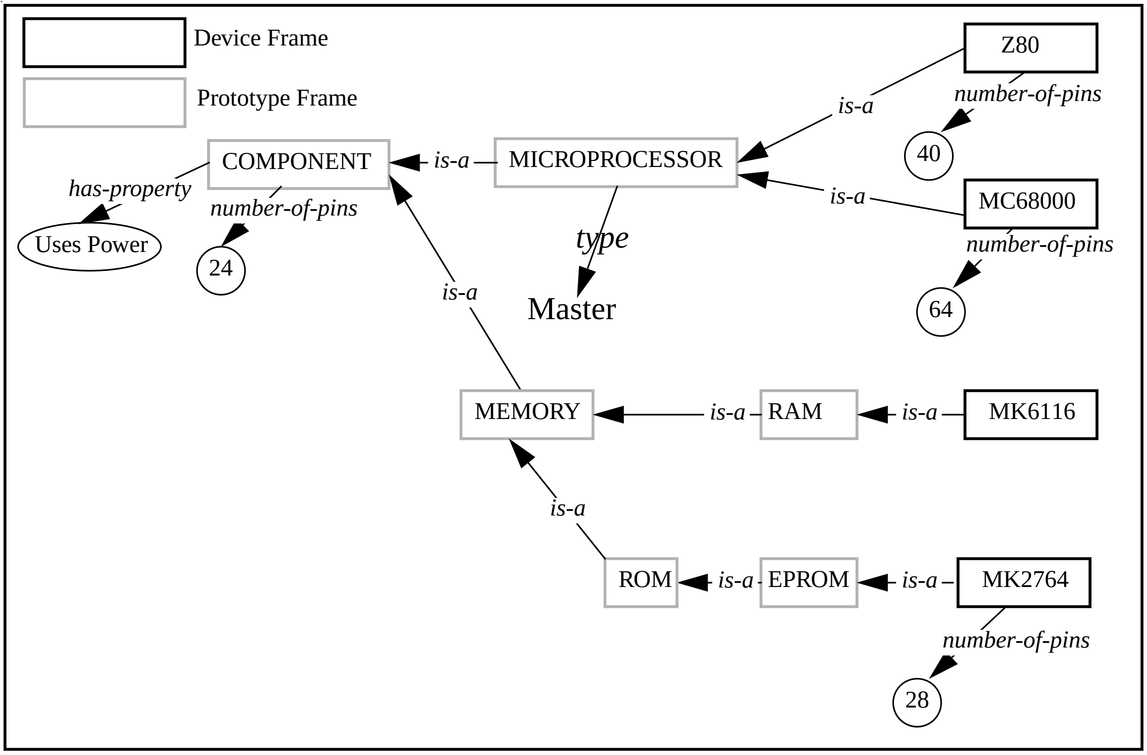


FIGURE 3-12. Example Device Frames

The component prototype frames represent the set of possible frames that can be used to build a component in the component library. The interface prototype frames represent the set of possible frames that can be used to build an interface. The device frames represent instantiations of the prototype frames used to represent a specific components or interface. The number of prototype frames is limited by the interface design rule base. Only those prototype frames that can be manipulated by the rule base are allowed to exist. The number of component device frames is limited only by the number of devices entered into the component library, while the number of interface device frames is limited only the number of different interfaces that can be designed. The strict separation of the device and prototype frames provides an important advantage for the maintenance of the Interface

Designer: If device frames of a new component are created by instantiating existing prototype frames and entered into the component library, the rule base does not have to be modified to be able to design with the new component. In other words, the maintenance of the component prototype frames and the rule base is separated from the maintenance of the component library.

The `MC68000` device frame represents all MC68000 microprocessors. To represent a single, specific MC68000 in the microprocessor system to be designed, an instance of the `MC68000` device frame is created, and given a name such as `U1`, as shown in Figure 3-13. The instance frame is linked to the device frame through the ^*instance-of* link. The instance frame is used to represent an actual physical device that will be installed in a system. A frame at the instance level can not be instantiated since it represents an actual component. An instance frame will inherit all properties from the parent device frame and the grandparent prototype frames through the ^*instance-of* relation. The example in Figure 3-13 shows the instance frames of a small microprocessor system that consists of a MC68000 microprocessor, two MK6116 RAMs and one MK2764 EPROM.

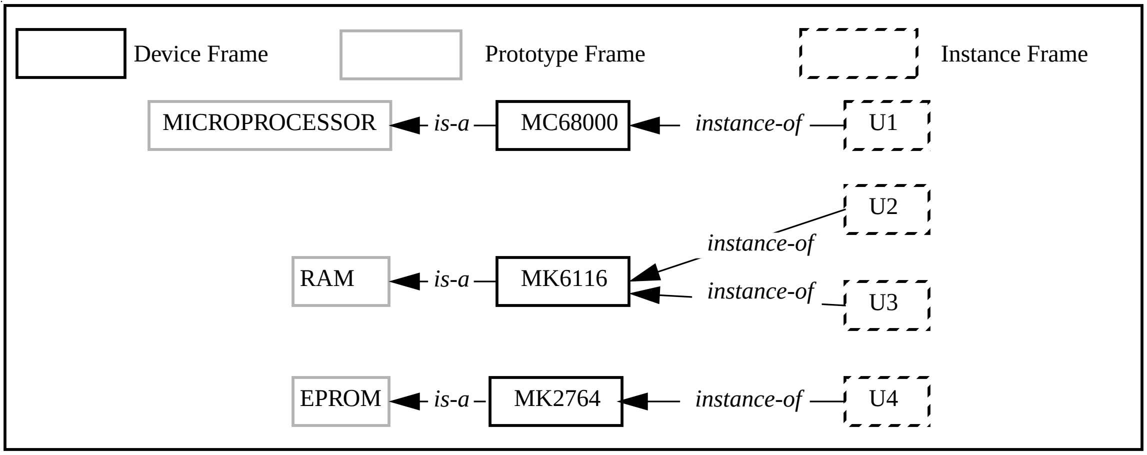


FIGURE 3-13. Component Instance Frames

3.8 Summary

This chapter developed the overall structure of the rule based interface design system. The design system was partitioned into a component model that represents the static components, an interface model that represents the interface between the components, and the design knowledge in the form of design rules which build up the interface, as shown in Figure 3-14. The abstraction levels of the component model, the interface model and the design rules follow each other closely. This facilitates the development of well structured

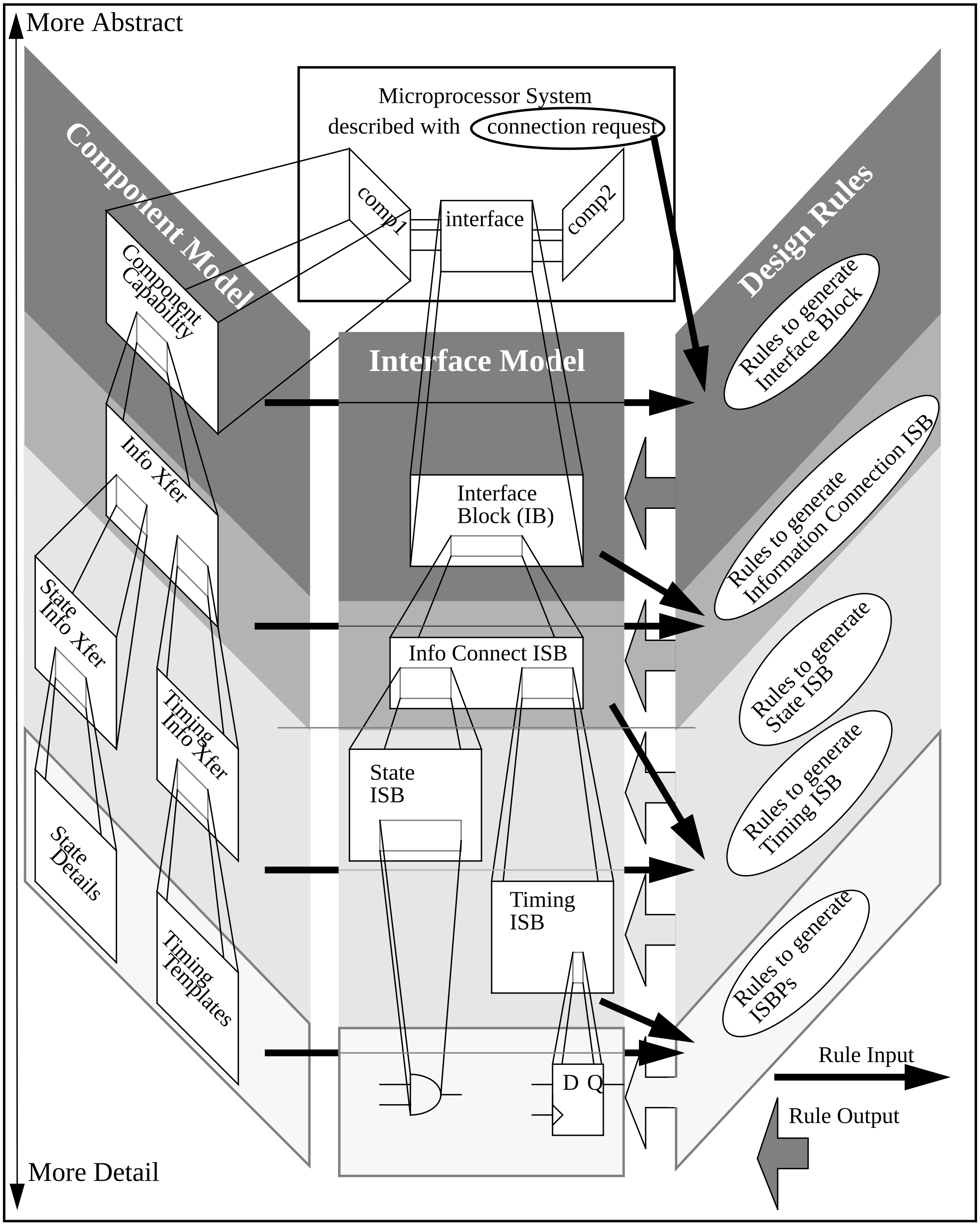


FIGURE 3-14. Interface Designer Knowledge Representation

design rules that can perform design at each level of the hierarchy. The component model is abstracted into capabilities whose protocol is built up from information transfers. The information transfers are abstracted into state and timing information transfers. In turn, the timing information transfers are built up from timing templates. During the design process, design rules create IBs that connect capabilities. The IBs are sub-divided into ISBs using design rules. The ISBs are finally sub-divided into ISBPs which can be implemented using discrete logic or VLSI gates. The ISBPs are chosen using rules that recognize the timing behavior of the timing templates.

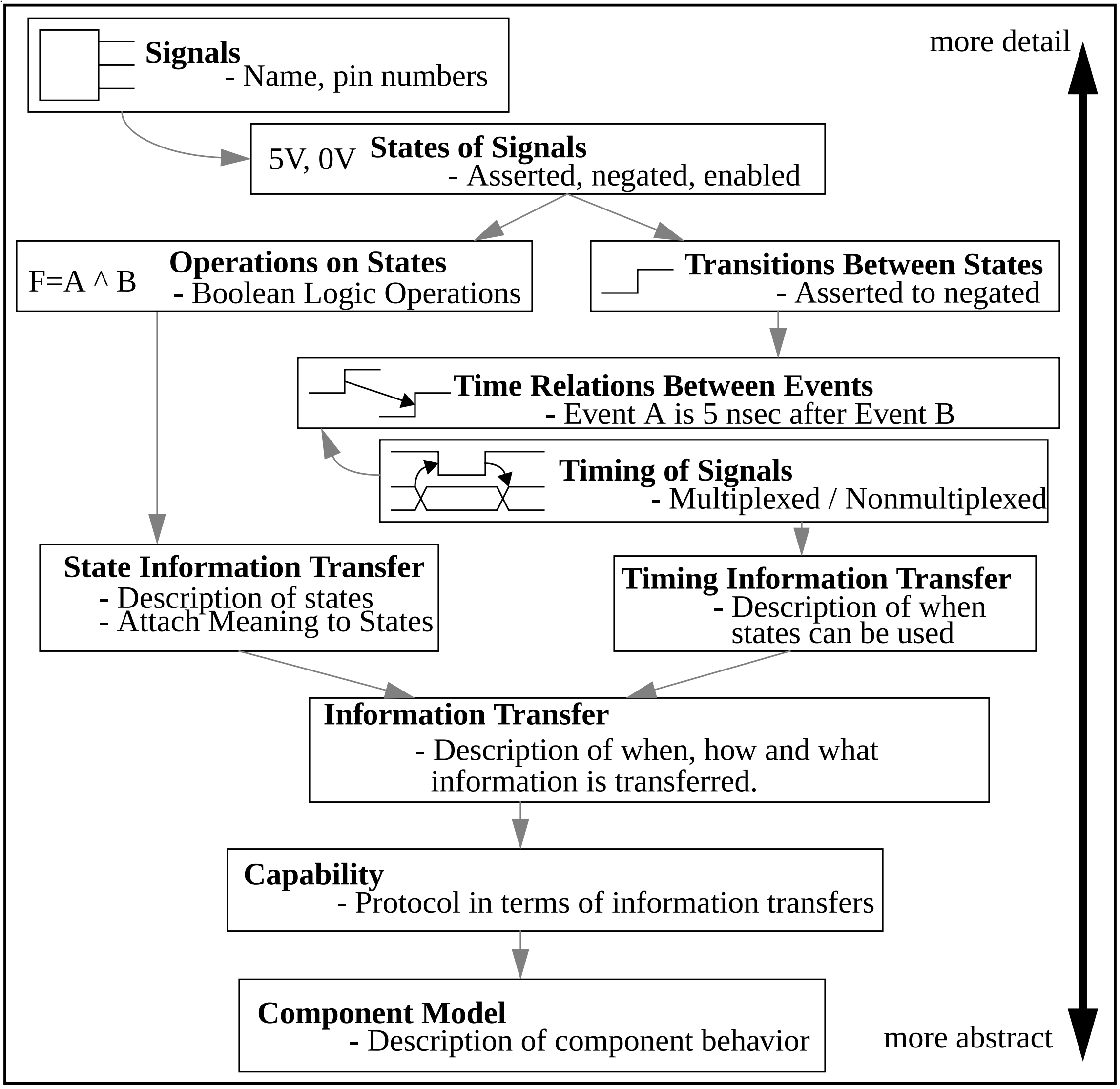
The next three chapters give a detailed description of each of the three parts of the interface design system: the component model, the interface model and the design rules.

Chapter 4

Microprocessor System Component Model

4.1 Introduction

The development of a production system based interface designer requires a representation of the components in the form of a component library. This chapter discusses how a component is represented as a frame based semantic network called the *component*



Signals
- Name, pin numbers
5V, 0V
States of Signals
- Asserted, negated, enabled
F=A ^ B
Operations on States
- Boolean Logic Operations
Transitions Between States
- Asserted to negated
Time Relations Between Events
- Event A is 5 nsec after Event B
Timing of Signals
- Multiplexed / Nonmultiplexed
State Information Transfer
- Description of states
- Attach Meaning to States
Timing Information Transfer
- Description of when states can be used
Information Transfer
- Description of when, how and what information is transferred.
Capability
- Protocol in terms of information transfers
Component Model
- Description of component behavior
more detail
more abstract

FIGURE 4-1. Outline of the Component Model Presentation

model. The component model is able to represent all aspects of a component's behavior from the abstract capability to the detailed voltage specification of a signal's logic level.

The primary objective in developing the component model was the development of a representation that is sufficient to accomplish reliable interface design. This means that only the aspects that are required to carry out and complete the task of interface design are modeled.

An outline of the component model and its abstraction hierarchy was given in chapter 3. This chapter gives a complete description of the component model, starting with a detailed description of the electrical signals and working up to the abstract description of the capability protocol. The order of presentation is shown in Figure 4-1. First, the representation for the signals and the electrical states of signals is given. This is followed by a representation of state changes of signals in the form of transitions and events. The time behavior of the signals is developed as a limited number of timing patterns, such as multiplexed and non-multiplexed signal timings. The concept of information transfer between components is developed as a combination of state information and timing information transfers. A description of the data transfer capability protocol is presented in terms of the information transfer, which finally allows the description of a component in terms of its capabilities.

4.2 Signals

Devices such as microprocessors and memories have *pins*. A pin is a metallic contact which connects electrically to circuitry inside a device package. A pin used to transfer information to or from a device is called a *signal*. A signal is represented by a name such as `A17` or `R/W*`, its location on the package such as pin 31, and its electrical characteristics.

The signals in microprocessor systems are often divided into groups according to their function. A group of signals associated with a specific function is called a *port*. For example `A0`, `A1`, `A2` etc. are signals associated with the address port of the MC68000. Each port will have an abstract classification associated with it, which represents the kind of information being transferred over the port signals, such as address, data or direction information. A unique name is used to designate each port such as `XYZ`, `ADDRESS`, `DATA`.

4.3 The State of a Signal

The logic state of a signal is a characteristic that can be propagated through a wire or through a logic circuit. Every logic state has a unique name and has a physical manifestation that corresponds to one or more electrical state. All the logic states defined for signals are given in Figure 4-2.

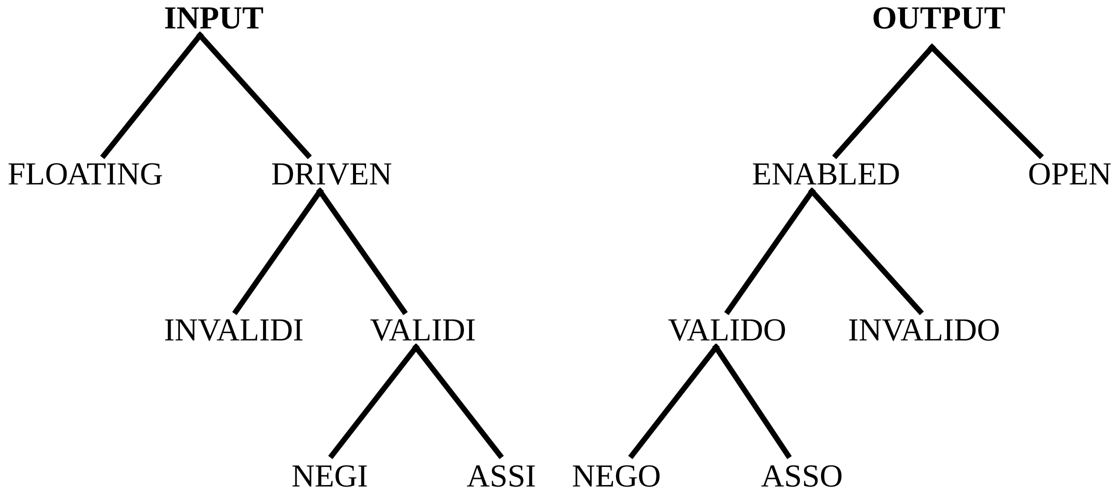


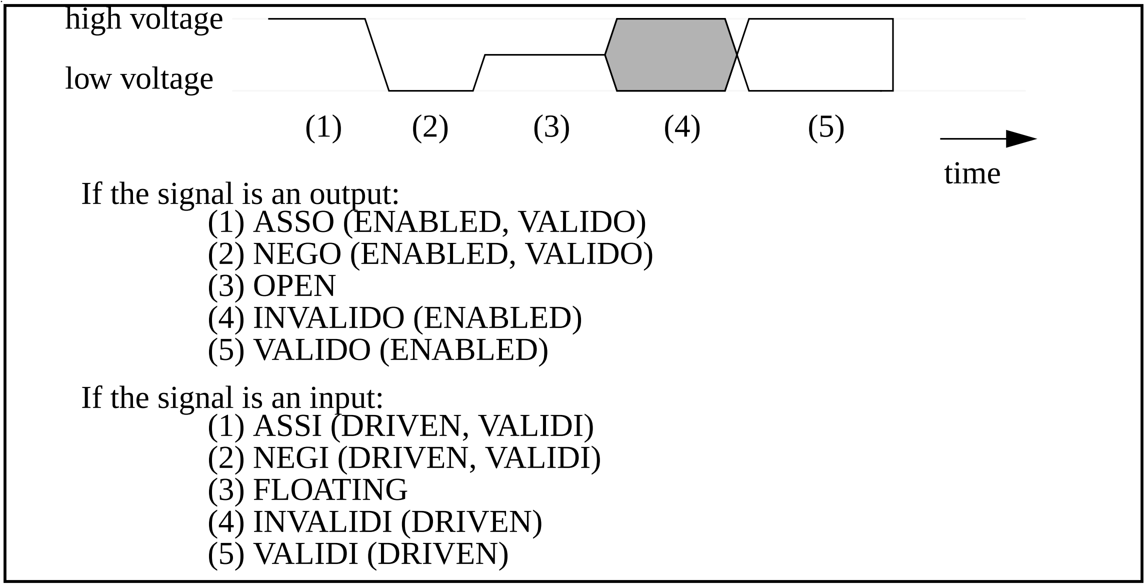
FIGURE 4-2. Logic State Hierarchy

The logic states are organized as follows. A signal can receive information, called an INPUT state or transmit information, called an OUTPUT state. A signal in OUTPUT state can be either OPEN (disconnected) or ENABLED. An ENABLED signal supplies either a high or a low voltage. An ENABLED signal is either a VALIDO or INVALIDO output. VALIDO means that useful information is present on the pin, while INVALIDO means that the information on the pin can not be used. If the signal is VALIDO it can be one of two binary states, either ASSO (asserted output) or NEGO (negated output).

A signal in INPUT state can be either FLOATING (not driven by anything) or DRIVEN. A DRIVEN signal is either at a high or low voltage level. A DRIVEN signal is either a VALIDI input or INVALIDI input. VALIDI means that information must be provided on the pin since it will be used internally, while INVALIDI means that the information on the pin will not be used. If the signal is VALIDI it can be one of two binary states, either ASSI (asserted input) or NEGI (negated input).

The asserted and negated levels of a signal have different interpretations depending on manufacturer's definition. *Asserted low* or *active low* means that the asserted state is represented by a low voltage while negated state is represented by a high voltage. *Asserted* *high* or *active high* means that the asserted state is represented by a high voltage while the negated state is represented by a low voltage.

A graphical representation of the voltage levels versus time, as commonly found in timing diagrams provided by component manufacturers, is shown in Figure 4-3.



If the signal is an output:
(1) ASSO (ENABLED, VALIDO)
(2) NEGO (ENABLED, VALIDO)
(3) OPEN
(4) INVALIDO (ENABLED)
(5) VALIDO (ENABLED)

If the signal is an input:
(1) ASSI (DRIVEN, VALIDI)
(2) NEGI (DRIVEN, VALIDI)
(3) FLOATING
(4) INVALIDI (DRIVEN)
(5) VALIDI (DRIVEN)

FIGURE 4-3. Voltage Levels Associated with Sates

4.3.1 Compatible States

When a manufacturer specifies a certain state for an input signal, it means that the input is required to attain that state for correct operation. For this reason the input state is called an *input requirement*. When a manufacturer gives a certain state for an output signal, it means that the output will be the specified state. For this reason an output state is called an *output specification*: the output signal pin will be at the specified state. Two states are *IO compatible* if an output specification satisfies the input requirement. An output of one device can be connected to the input of another device if the output state is IO compatible with the input state. Table 4-1 shows compatible input and output states.

Input State	**Compatible Output States**
INPUT	OUTPUT
FLOATING	OPEN
DRIVEN	ENABLED INVALIDO VALIDO ASSO NEGO
INVALIDI	ENABLED INVALIDO VALIDO ASSO NEGO
VALIDI	VALIDO ASSO NEGO
ASSI	ASSO
NEGI	NEGO

TABLE 4-1. Compatible States

4.3.2 Representing the States of a Signals

The state of a signal is a basic property of a component. A notation has been developed to represent the state of a signal in the Interface Designer and for discussion purposes. The syntax of the notation is given using the *BNF* (Backus-Naur Form), where the following symbols have meaning:

::==	definition
|	exclusive-OR
{}	repeat zero or more times
[]	optional
‘ ’	string

The BNF notation has been extended in two ways: square brackets are used to enclose an optional symbol or group of symbols, while curly braces are used to enclose a symbol or group of symbols that may be repeated zero or more times. BNF notation will also be used to describe other properties of signals, such as state changes.

The expression that represents the state of a signal is called a *signal state* and is given as:

<*signal state*>::== ‘(’<*logic state*> <*signal name*>‘)’
<*logic state*> ::== INPUT | FLOATING | DRIVEN | VALIDI | INVALIDI | ASSI | NEG I

| OUTPUT | OPEN | ENABLED | VALIDO | INVALIDO | ASSO | NEGO
<*signal name*> ::== <*identifier*>
<*identifier*> ::== Unique character string

A *boolean signal state* is a signal state that only involves the boolean logic states asserted and negated:

<*boolean signal state*>::== '('<*boolean logic state*> <*signal name*>')'
<*boolean logic state*> ::== ASSI | NEGI | ASSO | NEGO

For example, (VALIDI D0) represents the valid input logic state of the signal D0 and (ASSO UDS) represents the asserted output boolean logic state of the UDS signal.

A <*port state*> gives a shorthand method of representing the state of all the signals in a port, provided that all the signals have the same state:

<*port state*>::== '('<*logic state*> <*port name*>')'
<*port name*> ::== <*identifier*>
<*port*> ::== '(' <*signal name*> {<*signal name*>} ')'

For example, the port state (VALIDO ADDRESS) where ADDRESS:=(A0, A1, A2, A3, A4, A5, A6, A7), indicates that the address signals A0-A7 have a valid output state.

The signal state represents a property of a signal, as in the example given above, where (VALIDI D0) represents the valid input logic state property of the signal D0. The property of a signal can be either true or false: if the signal D0 has a valid input logic state, the signal state (VALIDI D0) is true, if the signal D0 does not have a valid input logic state, the signal state (VALIDI D0) is false. The statement 'the signal D0 has a valid input logic state' is equivalent to saying 'the signal state (VALIDI D0) is true'.

4.4 Using Signal States to Describe Situations

Signal states are used to describe situations in microprocessor systems. For example, the Z80 microprocessor signal state (ASSO WR*) is used to indicate that the current memory access operation is a write operation. Often, a situation is described by the state of several signals. For example, for the MC68000, a memory access in the User Program data space is indicated if the state (NEGO FC0) is true, the state (ASSO FC1) is true, and the state (NEGO FC2) is true.

To describe the state of several signals applicable to a situation, the *signal state expression* was developed, incorporating the AND, OR and NOT operators between the states of signals. A signal state expression using the AND operator indicates that each of the argument signal states is present on the given signals, or each of the argument signal states is true. A signal state expression using the OR operator indicates that at least one of the argument states is present on the given signals, or at least one of the argument signal states is true. A signal state expression using the NOT operator indicates that the argument signal state is not present on the given signals, or the argument signal state is false.

<signal state expression> ::== *<signal state>* | *<or state expression>* | *<and state expression>* | *<negation state expression>*
<or state expression> ::== '(OR' *<state list>* ')'
<and state expression> ::== '(AND ' *<state list>* ')'
<negation state expression> ::== '(NOT' *<signal state expression>* ')'
<state list> ::== *<signal state expression>* {*<signal state expression>*}

All signal states in a signal state expression must either be input signal states or output signal states. The mixing of input and output signal states is not allowed since the interpretation of such an expression would be ambiguous.

For example,

(AND (NEGI R/W) (OR (ASSI LDS) (ASSI UDS))) (EQ 4-1)

Equation 4-1 shows a signal state expression that indicates that the `R/W` signal is negated and either `LDS` or the `UDS` signal is asserted.

In practice it was found that only boolean logic states are used in signal state expressions. A signal state expression using boolean signal states is equivalent to a boolean logic expression, where the boolean variables are replaced with signal states. This allows boolean algebraic theorems, such as De Morgan's theorem, to be applied to manipulate the state expression. For example,

(AND (ASSO A0) (NEGO A1))=
(NOT (OR (NOT (ASSO A0)) (NOT (NEGO A1)))) (EQ 4-2)

4.5 State Changes in Signals

4.5.1 Transitions

A *transition* is the change of the logic state of an input or output signal from one logic state to another logic state at some instant in time. A transition is given as:

<transition> ::== '(' [*<logic state1>*] '!' *<logic state2>* *<signal name>* ')'
<logic state1> ::== *<logic state>*
<logic state2> ::== *<logic state>*

If *<logic state1>* is omitted, it is assumed to be the opposite state of *<logic state2>*. *<logic state1>* and *<logic state2>* must either both be input signal states or both be output signals states. The opposite logic states are shown in Table 4-2, and are derived from the logic state hierarchy in Figure 4-2 by determining which states are at the same level in the hierarchy.

Logic State	**Opposite Logic State**
ASSO	NEGO
ASSI	NEGI
VALIDO	INVALIDO
VALIDI	INVALIDI
ENABLED	OPEN
DRIVEN	FLOATING

TABLE 4-2. Opposite States

For example, assume an output signal `XYZ` changes state from asserted to negated. This transition can be written as (ASSO ! NEGO `XYZ`) or in a more compact form as (! NEGO `XYZ`).

A *port transition* is the change of state of the set of signals associated with a port, where each signal in the port has the same transitions:

<*port transition*> ::== ‘(‘ [<*logic state1*>] ‘!’ <*logic state2*> <*port name*> ‘)’

A port transition is simply a shorthand way of specifying the transition for each signal in a port, provided that all signals have the same transitions. For example the port transition (! ASSO `ADDRESS`) can be expanded into: (! ASSO `A0`), (! ASSO `A1`), (! ASSO `A2`), ...(! ASSO `A11`) if the `ADRRESS` port consists of the signals `A0, A1, ...A11`.

4.5.2 Events

The *event* extends the concept of transition from a single signal to several signals. A signal state expression represents a certain combination of signal states. When the combination of signal states becomes true at some instant in time, an event occurs. An event is defined as follows:

<*event*> ::== <*transition*> | ‘(’ ‘!’ <*signal state expression*> ‘)’

For completeness, a transition is also defined as an event. The ! in front of the <*signal state expression*> indicates the event that occurs when the signal state expression changes from false to true. The signal states that make up an event must either all be input signal states or all be output signal states.

Examples of events are:

(OPEN A1) ! (VALIDO A1) (EQ 4-3)

(! (OR ((ASSO LDS) (ASSO UDS)))) (EQ 4-4)

Equation 4-3 shows an event, which includes a transition, that occurs when the A1 signal changes from OPEN to VALIDO. Equation 4-4 shows an event using a signal state expression. Normally the UDS and LDS signals are negated. This means, that normally the signal state expression (OR ((ASSO LDS) (ASSO UDS))) will be false. As soon as both or either of the LDS or UDS signals change from negated to asserted, the state expression (OR ((ASSO LDS) (ASSO UDS))) will become true, and the event occurs.

4.5.3 Detectable Events

A *detectable event* is the change of state of one or more output signals which can be detected using some type of electronic circuit. The concept of detectable events is developed to allow the detection of the time of occurrence of the event: if the time of occurrence can be detected, it can then be used as a relative time reference to other events. Detectable events involve changes of signal states between the asserted and negated logic states. Any event that only involves boolean logic states is a detectable event. Theoretically there are other state changes that are detectable, such as (OPEN ! ENABLED XYZ). In practice these events can not be detected reliably, and are not considered detectable in this work. For example, Equation 4-4 is a detectable event, while Equation 4-3 is not a detectable event.

4.5.4 Complementary Events

An event occurs when one or more signals change logic state. When the signals that are involved in the event change back to the original state, a *complementary event* occurs. The event:

(<*logic state1*> ! <*logic state2*> <*signal name*>) (EQ 4-5)

has complementary event:

(<*logic state2*> ! <*logic state1*> <*signal name*>) (EQ 4-6)

and the event:

(! <*signal state expression1*>) (EQ 4-7)

has complementary event:

(! (NOT <*signal state expression1*>)) (EQ 4-8)

For example, the complementary event to Equation 4-3 is:

(VALIDO A1) ! (OPEN A1) (EQ 4-9)

The complementary event for Equation 4-4 is:

(! (NOT (OR ((ASSO LDS) (ASSO UDS))))) (EQ 4-10)

4.6 Modeling Time Relationships Between Events

4.6.1 The Timing Link Between Events

Electrical signals in microprocessor systems are used to transfer information. Knowing which signal is used to transfer the information and how the information is formatted is not enough to transfer the information successfully. It must also be known when the information will be transferred. For example, the knowledge that some address signals will contain a binary formatted address is of no use unless one knows exactly when the signals contain the address. Manufacturers specify when information transfer takes place by giving the relative times between events of certain signals. The time relationships between events of signals are given in the component manufacturer data books in the form of timing diagrams and tables of timing parameters. This is illustrated in Figure 4-4 for the MC68000 read cycle. Several signals are involved in the data transfer, such as CLK, A1-A23, R/W*, AS* and UDS*/LDS*. The manufacturer specifies when the address signals A1-A23 become valid (i.e. when the (INVALIDO ! VALIDO ADDRESS) event occurs) relative to the time of occurrence of events on the CLK, AS* and UDS*/LDS* signals. Different types of relations between events are provided, such as ‘event A precedes event B’ or ‘event A is always 10nsec after event B’. For example in Figure 4-4 the (! ASSO AS*) event and the (! VALIDO ADDRESS) event indicate that the address becomes valid before the asserted AS signal.

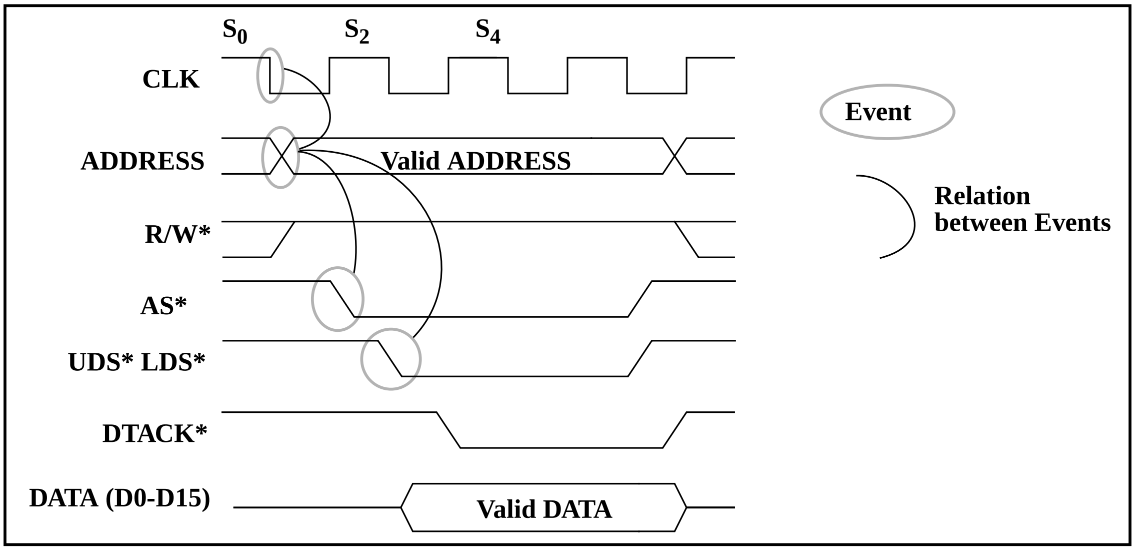


FIGURE 4-4. Timing Diagram of the MC68000 Read Cycle

Interface design requires knowledge about the time relationships between events. A method was developed to represent the relationships in the Interface Designer. The time relationships between events are represented using *timing links*. A timing link gives the relative time of an event, called the *tail event*, relative to another event, called the *head event*. There is no implied precedence between the head and tail events: the tail event can occur before or after the head event. The timing link represents an unidirectional time relationship from the head to the tail event. Unidirectional means that if a timing link gives the time of event A relative to event B, it may not be possible to infer the time of event B relative to event A. For example, if a link specifies that a tail event (! VALIDO ADDRESS) is always 20ns before a head event (! ASSO AS*), one can not assume that the (! ASSO AS*) event always will be 20nsec after the (! VALIDO ADDRESS) event.

Time relationships between events can be viewed as directed graphs. The graph nodes **E** represent the events and the directed graph links **R** represent the timing links between events. For example, Figure 4-5(b) shows the directed graph representation of the timing diagram shown in Figure 4-5(a). The directed graph gives the time relationship R of the $\mathbf{E_2}$ event relative to the $\mathbf{E_1}$ event.

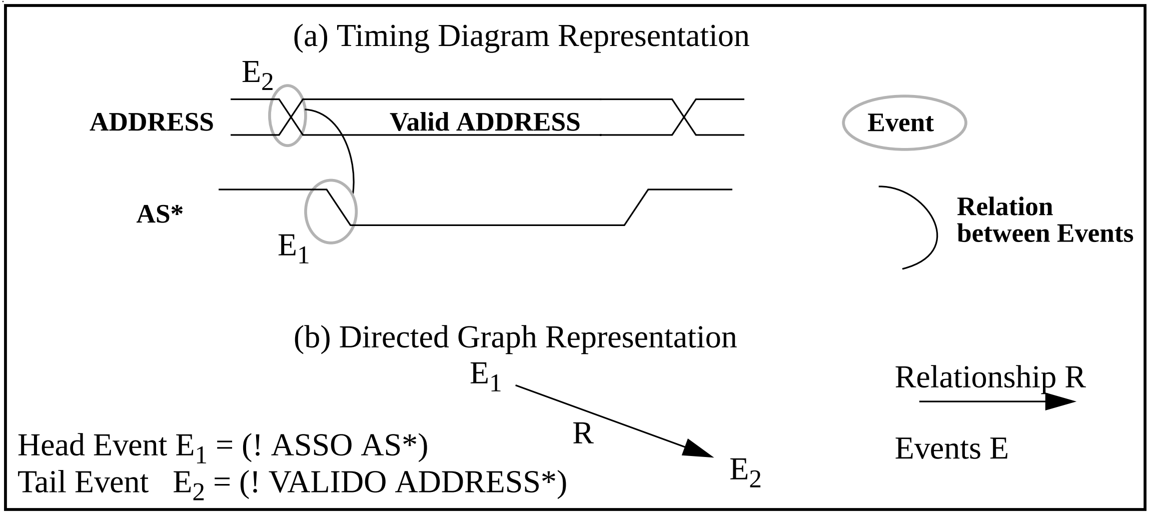

(a) Timing Diagram Representation
E2
ADDRESS
Valid ADDRESS
Event
AS*
E1
Relation between Events
(b) Directed Graph Representation
E1
R
E2
Relationship R
Events E
Head Event E1 = (! ASSO AS*)
Tail Event E2 = (! VALIDO ADDRESS*)

FIGURE 4-5. Example of Event Time Relationship

4.6.2 Repeated Event Sequences in Timing Diagrams

Often a timing diagram is used to show a sequence of events. For example, Figure 4-6(a) shows a sequence of complementary events for the WR* signal. Repeated event sequences for microprocessor components often correspond to repeated operations the components can carry out. For illustration purposes in this work, complementary events

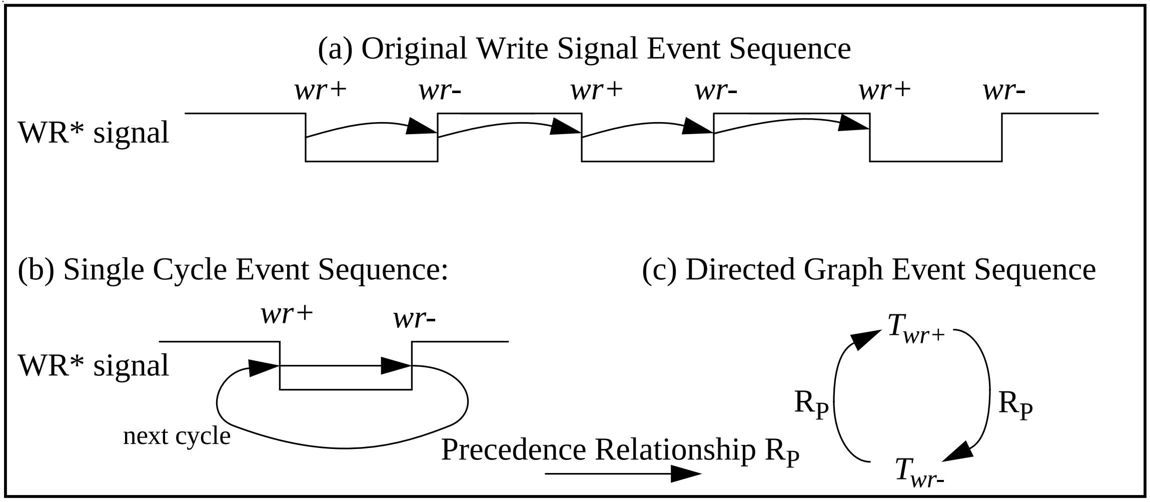

(a) Original Write Signal Event Sequence
wr+
wr-
wr+
wr-
wr+
wr-
WR* signal
(b) Single Cycle Event Sequence:
wr+
wr-
WR* signal
next cycle
(c) Directed Graph Event Sequence
T_{wr+}
R_P
R_P
T_{wr-}
Precedence Relationship R_P

FIGURE 4-6. Repeated Event Sequence Representation

(Section 4.5.4) in a sequence are given names with a + or - symbol attached to the end of the name. For example event *name+* has complementary event *name-*. Using the +/- convention in naming the complementary events helps in the understanding of diagrams showing temporal ordering between events. The + will be associated with the earlier, while the - will be associated with the later complementary event. In most circumstances, the + event is associated with the more 'positive' or 'beginning' event such as an event initiating something, a signal going from INVALIDO to VALIDO or a signal going from negated to asserted, while the - event is associated with the more 'negative' or 'end' event such as an event terminating something, a signal going from VALIDO to INVALIDO or a signal going from asserted to negated.

The timing diagram for a repeated event sequence can be redrawn for a single cycle of the repeated sequence as shown in Figure 4-6(b). A directed graph representation of the repeated event sequence is shown in Figure 4-6(c). In the directed graph, a subtle property for the signal behavior shown in the timing diagrams is represented by the graph links: The signals that generate the events are not allowed to change between the events. If this property is attached to the directed graph links, then the original event sequence shown in Figure 4-6(a) can be obtained from the directed graph Figure 4-6(c).

4.6.3 Properties of Timing Links

Timing links are used to represent different time relationships between events. This section describes the time relations represented by the timing links. Timing links have properties such as the precedence between the head and tail events and the time of occurrence of the tail event relative to the head event. For example, a timing link can represent a simple precedence relationship:

Event B is always after event A,

or the timing link can give the guaranteed time of an event relative to another event:

Event B always occurs between 10nsec and 20nsec before event A.

The interval 10nsec to 20nsec is called the *timing parameter* of the timing link. A timing parameter is a time interval with a lower and upper limit. The timing parameter represents a range of time values of when an event can occur. A time interval is written by enclosing the interval limits, in nano seconds, in brackets. In the above example, the timing parameter would be written as (-20 -10) since negative time values are used to indicate that event B occurs before event A. An interval (a b) is written so that $a \leq b$. A timing parameter that is an exact time such as 20nsec, is considered an interval of zero length and is written as either a single value (20) or as a range with the same lower and upper limits (20 20). Using the notation for intervals, the previous timing link description could be written as:

Event B always occurs at time (-20 -10) relative to event A.

An event may not always occur with another event, but it may still be related to the other event as in:

If Event B occurs, it will occur at time (-20 -10) relative to event A.

Timing links between events on the same signals can also convey information about the signals' behavior between the events as in:

Event B will occur at a time (50 70) relative to event A and the signals involved in events A and B will not change state between events A and B.

As these examples show, timing links have several properties and characteristics, which include:

- The direction of the signals involved in the events: Input to Input, Output to Output, Output to Input or Input to Output.
- A timing parameter
- If there is precedence between the events
- If an event always occurs with another event, or only sometimes
- If the signals change state between the events (if both events involve the same signals).

There are a total of six timing links developed that are sufficient to represent all practical cases and are presented in the next sections.

4.6.4 Timing Links Between Events

Four possible timing links based on the direction can be found in the specification of time relationships between two events as illustrated in Figure 4-7.

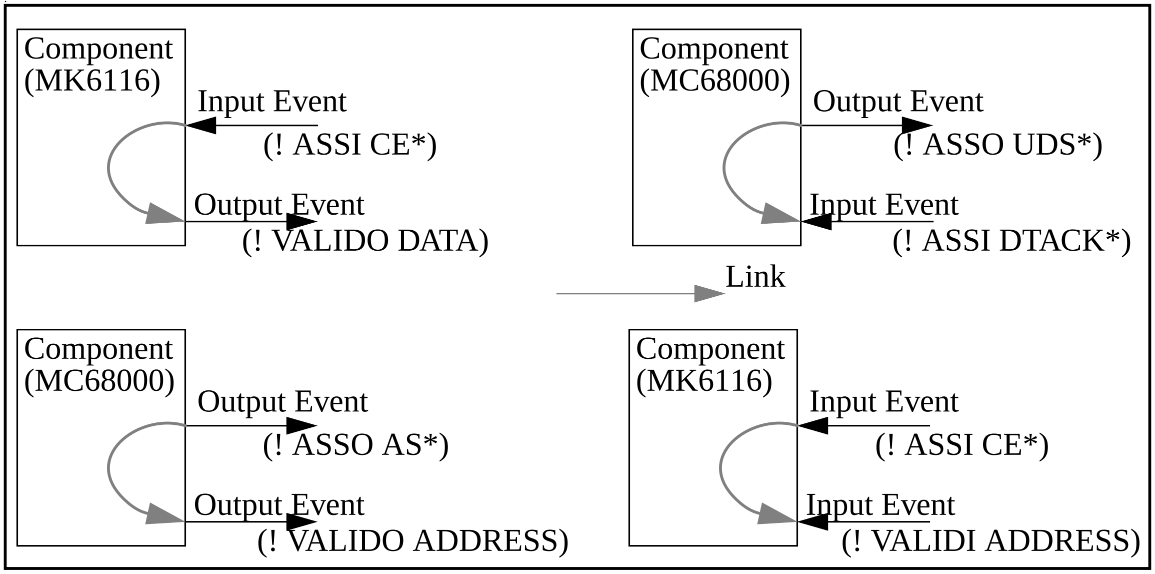


FIGURE 4-7. Possible Event Relationships

The input to output timing links and the output to input timing links are called *causal* timing links since there is a direct cause and effect relationship between the head and tail events. A cause and effect relationship implies a precedence between the head and tail events. For example, for a MC68000, the asserted DTACK* signal will cause the UDS* signal to become negated. This means that the (! ASSI DTACK*) event will have a causal timing link to the (! NEGO UDS*) event, and the (! ASSI DTACK*) event must precede the (! NEGO UDS*).

The input to input and output to output event relationships are called *non-causal* timing links since there is no cause and effect relationship between the events. The precedence of the events related by a non-causal timing link is not known. For a non-causal timing link the tail event could occur at any time relative to the head event.

If the tail event of a timing link is an input event, the timing link specifies an *input requirement*, since the link dictates how the input must behave. If the tail event is an output event, the timing link specifies an *output specification*, since it dictates how the output behaves.

If all timing links for the events of a device have strict precedence between events a petri net called a *signal transition graph* (STG) can be used to represent the timing behavior. STGs are directed graphs consisting of the events **T** and the precedence relation **P** between the event nodes. When state transition graphs were developed they were applied to designs which were delay insensitive (i.e. unbounded positive delays), but they have been extended to include timed delays [69][27].

Instead of a petri net based approach this work develops its own specialized timing links to represent the timing behavior patterns for several reasons:

- This work only requires a representation that gives the general pattern of a timing behavior, not the detailed relationships between all signals. STGs provide too much detail, thus adding unnecessary complexity.
- This work requires that links have certain properties in addition to a time value. For example such a property may be a link that specifies that a signal will not change state between two specified events.
- This work requires that the links in timing patterns can have a general behavior associated with them such as: The setup time of signal A relative to a reference event is usually negative or close to zero. This type of behavior can then be used to develop the concept of a propagation delay invariant timing templates as explained in Section 4.7.2.

There are some similarities between STGs and the timing behavior model developed for this work, and it may be possible to extend STGs to include some of the special concepts developed here.

The following sections describe the timing links developed for this work and their associated properties.

4.6.4.1 Causal Timing Links

A timing link between either an input event and an output event or an output event and an input event is called a causal timing link.

The input to output event link represents the response output event to some input event and is called a *responds-with* timing link. Due to the cause and effect relationship between the events (i.e. an input event will cause the output event), there is strict precedence and the timing parameter (a b) associated with the responds-with timing link is restricted to $a \leq b$, $a \geq 0$ and $b \geq 0$.

The output to input event link expects an event on an input port in response to the event on an output port, hence it is called an *expects* timing link. Normally, there is strict precedence between the output and input events. It was found however that when describing the timing behavior of more than two events as done later in this work, that an expects timing link is required that can take on a negative timing parameter value. For this reason, an expects link is defined with no restrictions on precedence. The timing parameter (a b) of the expects link is a ≤ b, -∞ ≤ a ≤ +∞, and -∞ ≤ b ≤ +∞.

4.6.4.2 Non-Causal Timing Links

A non-causal timing relationship is one between two output events or two input events. For two output events or two input events that are related, the important knowledge about the events is not the order in which they occur, since that may change, but which event always occurs with the other event, and what the relative time between the events is.

The *always-accompanied-by* timing link represents a non-causal timing link between two input events or two output events that always holds true. A typical example of this is the time relation between a transfer request signal AS* event and the ADDRESS signal event of a MC68000 microprocessor as shown in Figure 4-8. It is known that if an *as+* event occurs on the AS* signal, it will always be preceded by the *add+* address event on the ADDRESS signals at a certain time. Similarly, if an *as-* event occurs on the AS* signal, it will be followed by the *add-* address event on the ADDRESS signals at a certain time.

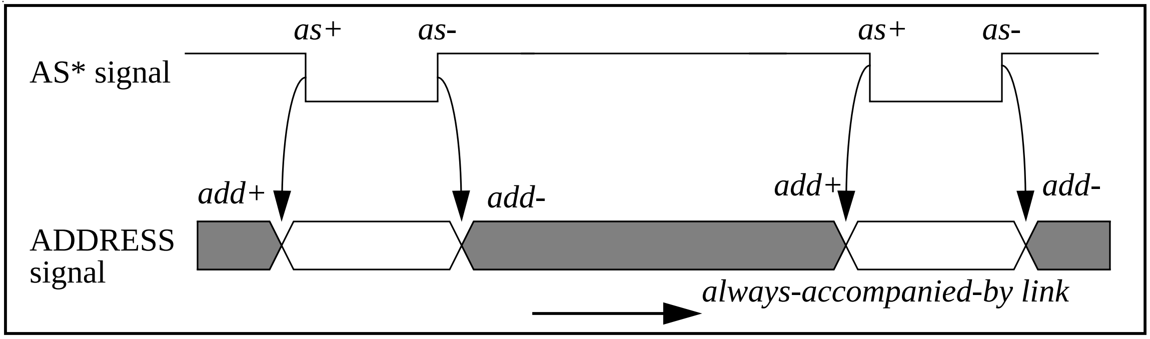


FIGURE 4-8. Example of the Always-Accompanied-by Link

Sometimes an event is not always accompanied by another event, but if it is accompanied by another event, a certain timing relationship exists between the events. This timing link is called an *accompanied-by* timing link. An example of the accompanied-by link is shown in Figure 4-9. The DS* signal indicates that a data transfer operation will occur. It is sometimes accompanied by a WR* signal as shown in the figure with a certain timing relationship to the DS* signal. If a *ds+* and *ds-* event sequence occurs, the *wr+* and *wr-*

event sequence may or may not occur, and if it does occur it will have a given timing relationship as specified by the accompanied-by timing parameter.

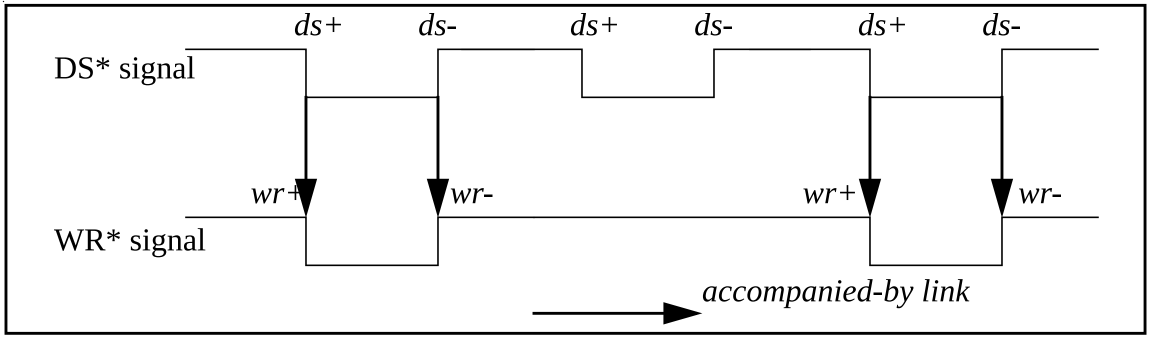


FIGURE 4-9. Example of the Accompanied-by Link

The timing parameter (a b) associated with the always-accompanied-by and accompanied-by timings link is a ≤ b, -∞ ≤ a ≤ +∞, and -∞ ≤ b ≤ +∞.

4.6.5 Timing Links Between Complementary Events

Figure 4-10 shows a typical write data transfer operation for a microprocessor. Always-accompanied-by can be used to specify links between the wr+ and dat+ events and the wr- and dat- events. The timing diagram shown in Figure 4-10 conveys more information about its behavior to the designer than that given by the two always-accompanied-by links. The timing diagram also indicates that the WR* signal does not change between the wr+ and wr- events, and that the DATA signal does not change between the dat+ and dat- events. Special always-accompanied-by links have been developed to represent the behavior of signals between complementary events.

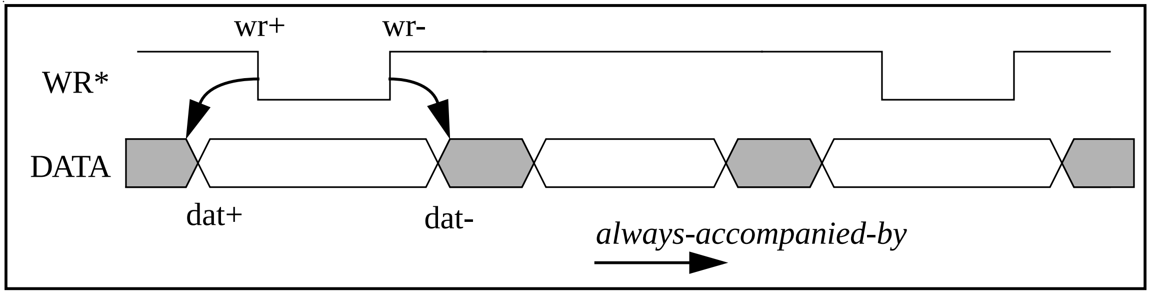


FIGURE 4-10. Typical Data Write Operation Timing Diagram

The always-accompanied-by link between complementary events can be split into two types: a *complementary-precedes* timing link where the signals involved in the events will not change until the complementary event occurs called, and an *eventually-precedes* timing link where the signals are allowed to changed between complementary events. This is illustrated in Figure 4-11 (a) for a typical data write operation. Link 5 is a complementary-precedes link which indicates that the DATA signals will not change between the dat+ and dat- events and that the dat+ event occurs before the dat- event. Link 6 is an eventually-precedes link where the DATA can change between the dat- and dat+ events.

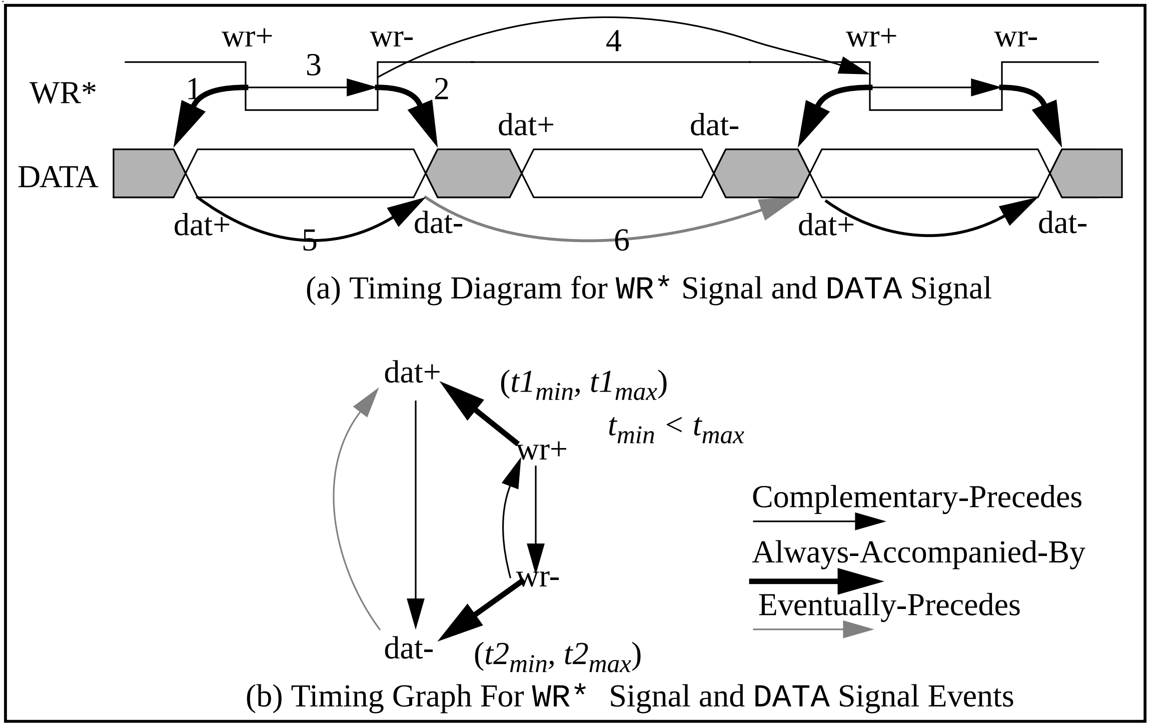


FIGURE 4-11. Typical Data Write Operation Timing Links

The complementary-precedes and eventually-precedes links always have precedence: the head event will always occur before the tail event. This means that their timing parameter (a b) is a ≤ b, a ≥ 0 and b ≥ 0.

Using the timing links developed, the behavior of microprocessor system signals can now be specified. For the timing behavior of the data signal in Figure 4-11(a), link 3, 4 and 5 are complementary-precedes links while link 6 is an eventually-precedes link. Links 1 and 2 are always-accompanied-by links. The links developed make it possible to give a graphical representation of the data write timing behavior as shown in Figure 4-11 (b). Using the graphical representation given in Figure 4-11(b), it is possible to reconstruct the timing diagram representation shown in Figure 4-11(a).

4.6.6 Timing Link Summary

This section developed a method to represent the time relationships between events in the form of links. Table 4-3 shows a summary of all the timing links and their properties.

Timing Link	**Head to Tail**	**Specification or Requirement**	**Classification**	**Precedence**	**Timing Parameter (a b), b>a**
Responds-with	in to out	Specification	Causal	Yes	$a, b \geq 0$
Expects	out to In	Requirement	Causal	No	$-\infty \leq a \leq +\infty$ $-\infty \leq b \leq +\infty$
Always-Accompanied-by	out to out in to in	Specification Requirement	Non-Causal	No	$-\infty \leq a \leq +\infty,$ $-\infty \leq b \leq +\infty$
Accompanied-by	out to out in to in	Specification Requirement	Non-Causal	No	$-\infty \leq a \leq +\infty,$ $-\infty \leq b \leq +\infty$
Complementary-precedes	out to out in to in	Specification Requirement	Non-Causal	Yes	$a, b \geq 0$
Eventually-precedes	out to out in to in	Specification Requirement	Non-Causal	Yes	$a, b \geq 0$

TABLE 4-3. Component Timing Links

It should be pointed out that the timing links were developed with foresight as to how they will be used to represent the timing behavior of signals in the Interface Designer. Specifically, the timing behavior of signals are often similar but not identical, and the timing links will allow the similarities to be extracted as patterns which the Interface Designer can use to perform design. For example the behavior of the `DATA` signal relative to the `WR*` signal in Figure 4-11(b) is the typical behavior of a non-multiplexed signal in a microprocessor system. By giving this timing behavior a name such as XYZ, any similar timing behavior can be simply described by stating that it is of type XYZ and giving the specific timing intervals associated with each link. This type of representation integrates very nicely with frame based semantic networks used for this work: the timing behavior of every signal is based on a timing template (such as pattern XYZ) with specific timing parameters for each link specified in an instantiation of the timing template. This will be discussed in more detailed in the section on modeling signal timings.

4.6.7 Notation Used to Represent Timing Links Between Events

A timing link represents the time relationship between two events. A *timing link expression* is used to represent the relation between the events and the associated timing parameters. A timing link expression is given using the following notation:

<*timing link expression*> ::== <*head event*> '->' <*tail event*> {<*tail event*>}'@' <*time*>
<*head event*> ::== <*event*>
<*tail event*> ::== <*event*> | <*port transition*>
<*time*> ::== '(' <*time value*> [, <*time value*>] ')'
<*time value*> ::== <*numeric constant*> | '-~' | '+~'
'-~' represents a time of negative infinity
'+~' represents a time of positive infinity
<*numeric constant*> ::== character string representing time in nano seconds.

The RHS of a timing link expression consists of a set of one or more events. (A port transition can be expanded into a set of transitions). The <*time*> gives the timing parameter that indicates when the tail event will occur relative to the head event. For example,

(! (OR (ASSO UDS*) (ASSO LDS*))) -> (IVALIDO ! VALIDO A0) @ (-~ -10)

states that the address signal `A0` will go VALIDO 10 nsec or earlier before the occurrence of either an asserted `UDS*` or asserted `LDS*` signal. This link expresses what is often called the setup time of the `A0` relative to the asserted `UDS*` or `LDS*` signals: The A0 signal becomes stable and valid 10 nsec before the asserted `UDS*` or `LDS*` signals.

In this work, timing links are normally used to indicate when information transfer takes place. This means that the head event is used as a relative time reference point for the time of occurrence of the tail event. For practical design reasons, an event that is used as a time reference should be a detectable event. The head event for timing link expression developed for this work is therefore normally a detectable event.

4.7 Modeling Signal Timings

Section 4.5 and Section 4.6 showed how state changes for signals are modeled as events and how timing links represent time relations between events. This section builds on the concepts of events and timing links to develop *signal timings*. A signal timing can be thought of as a method of specifying when and how information is transferred between two components in a digital system relative to one or more detectable reference events.

4.7.1 Developing the Concept of Timing Templates

One of the difficulties the interface designer faces when connecting two components is the large variety of signal timing behavior that can be encountered. Often the general aspects of signal timing behavior are similar between components, while the detailed signal timing specifics are different. To overcome this problem the designer may use some heuristics by recognizing the general timing behavior patterns of a signal timing. For example, the designer may look at a timing diagram and recognize that the output address signal of one device has the general behavior of a multiplexed signal, while the input address signal of another device has the general behavior of a non-multiplexed signal. The designer recognizes the general signal interrelationships and knows that he must store the information carried by the multiplexed signals before connecting it to the non-multiplexed input signals. The designer knows that this storage can be accomplished with a D-Latch. The designer inserts the D-Latch into the interface and then verifies that none of the specific setup and hold times of the input are violated. This work takes a similar approach in organizing the knowledge about the timing behavior of a signal. The timing behavior knowledge is split into ‘a pattern’, which represents the general aspects of the timing

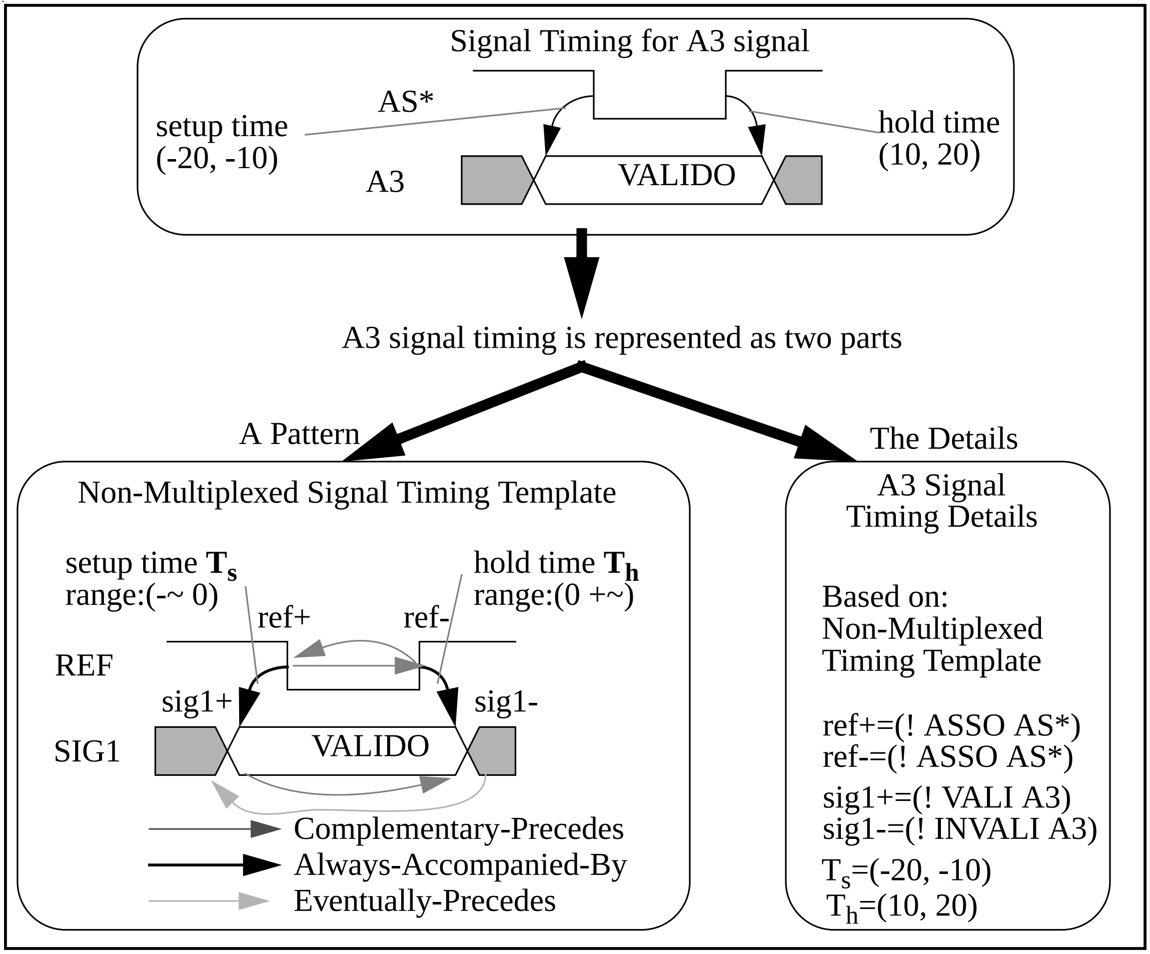


FIGURE 4-12. Representation of Signal Timing of Non-Multiplexed Signal A3

behavior, and ‘the details’ which represent the exact timing parameters such as setup and hold times, as shown for the non-multiplexed signal timing in Figure 4-12.

The general behavior pattern of a signal timing is called its *timing template*. A signal timing template classifies the characteristics, properties and behavior of a signal timing including:

- A description of the different events involved in the signal timing
- A description of each of the timing links and the associated events
- A range of allowed values for the timing parameters for each of the timing links

Signal timings that have similar characteristics, properties and behavior belong to the same class of timing templates.

After studying the signal timing behavior of many components, it was found that the setup timing parameter was almost always less than or equal to zero, while the hold time almost always was greater than or equal than zero. This property of the timing template is included as the *allowed range of values* for the timing parameters of a timing link. For example, (-~ 0) for the setup time and (0 +~) for the hold time in Figure 4-12.

The allowed range for a timing parameter in a timing template allows representation of heuristics used in interface design. It allows assumptions to be made about the timing behavior of a signal by simply looking at the timing template. In the non-multiplexed example of Figure 4-12, the setup and hold time allowed ranges enable us to assume that the `SIG1` signal will become valid before the ref+ event, and it will remain valid until after the ref- event, without having to consider the details of the timing.

The details for a timing are given by specifying the events and timing parameters. In the example, the ref+ and ref- events are specified as (!ASSO `AS*`) and (!NEGO `AS*`).

The timings and their templates integrate well with frame based semantic networks. A set of frames representing the templates for all possible signal timings is created. Then, when a component is entered into the database, the appropriate signal timing template frames are instantiated for each signal timing and the details for the specific signal timings are filled in.

4.7.2 Propagation Delay Invariance of Timing Templates

Timing templates were developed to provide a method that allows signals with similar timing behavior to be represented by the same timing template. Conceptually, if a signal is based on a given timing template class at one point in a circuit, it will be based on the same timing template class at any other point in the circuit.

The concept of *propagation delay invariance* was developed to provide a method to represent the design engineer's knowledge of the effect of circuit elements, such as wires, on the behavior of a signal timing. A design engineer knows that conceptually, if a signal has a given timing behavior at one end of a circuit wire, it will have the same timing behavior at the other end of a wire. However, physically, any two points of the same signal in a circuit will be separated by an inherent delay determined by the distance between the two points. Therefore the timing templates must have a property that preserves the behavior of the timing template class even in the presence of inherent circuit delays which are unknown until the design is implemented. The Interface Designer represents this heuristic as the propagation delay invariance property of the timing templates.

The concept of propagation delay invariance is extended to other simple electronic devices such as buffers. A buffer is an active device that is used to restore the voltage level of a signal and/or increase the drive capability of a signal. Conceptually the timing behavior of a signal before and after a buffer should be similar. In other words, the delay introduced by a buffer can be treated as a simple inherent delay of the circuit and should not fundamentally change the timing behavior of a signal.

The utility of propagation delay invariance can be seen from a simple example. Assume there exist two microprocessor components: for component A, a silicon die is packaged in a plastic dual in-line package with signal pins, while component B uses the same silicon die which is mounted on a printed circuit board with the signals going to an edge connector. Additionally, the address signals of component B have a buffer inserted to increase the current drive capability. The propagation delay invariance property allows the timing behavior of the signal pins of component A and the edge connector signal pins of component B to be represented by the same timing templates. This provides three important advantages: First, only one single model for the timing behavior for both components has to be developed. Second, rules representing design heuristics used for interface design of component A can also be used for component B. Third, the timings of signals within an interface can be based on the same timing templates as those used for the components.

It is known that the propagation delay is greater than zero, finite and generally on the order of magnitude of the propagation delay of the technology used to implement the design. However, the exact value is not known until a technology is chosen and the design is implemented. We shall call this type of delay an *omp* delay (for Order of Magnitude Propagation delay). Typical values for an omp delay in LS TTL technology are 7-10ns.

An example of the effect of propagation delay on the signal timing, if the signal is delayed, is shown in Figure 4-13. Signal A3, based on a non-multiplexed timing template, is delayed by an amount δ due to either physical separation or a buffer. Even though the resulting signal A3' has different setup and hold timing parameters relative to the same reference WR*, and the setup time changed sign ($T_s < 0$ and $T_s' > 0$), the A3 and A3' signals should be based on the same class of timing template.

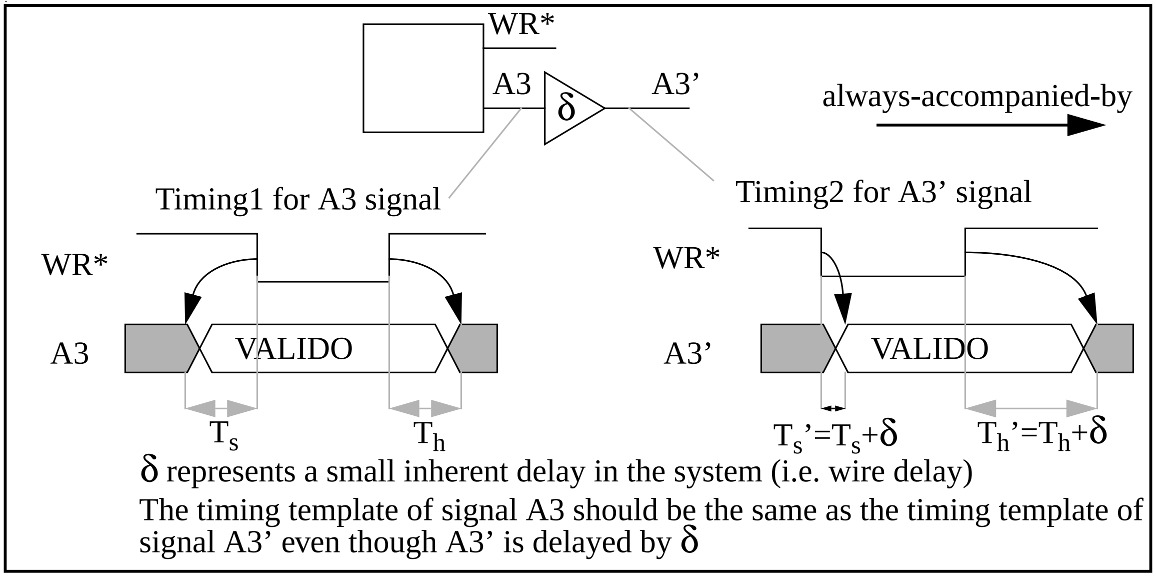

WR*
A3
δ
A3'
always-accompanied-by
Timing1 for A3 signal
Timing2 for A3' signal
WR*
A3
VALIDO
T_s
T_h
WR*
A3'
VALIDO
$T_s'=T_s+\delta$
$T_h'=T_h+\delta$
δ represents a small inherent delay in the system (i.e. wire delay)
The timing template of signal A3 should be the same as the timing template of signal A3' even though A3' is delayed by δ

FIGURE 4-13. Propagation Delay Invariance of Timing Template (Signal is Delayed)

Similarly, an example of the effect of a propagation delay on the signal timing if the reference is delayed is shown in Figure 4-14. The reference WR*, is delayed by an amount δ due to either physical separation or a buffer to produce WR*'. Even though the signal A3 has different setup and hold timing parameters relative to the two reference signals WR* and WR*', and the hold time changed sign ($T_h < 0$ and $T_h' > 0$), the signal timings relative to the WR* and WR*' signals should be based on the same class of timing template.

4.7.3 Developing Propagation Delay Invariant Timing Templates

The last section established that signal timing templates must be propagation delay invariant to assure that a timing template is unaffected by small delays inherent in a microprocessor system. This section presents the methods used to make the signal timing templates propagation delay invariant.

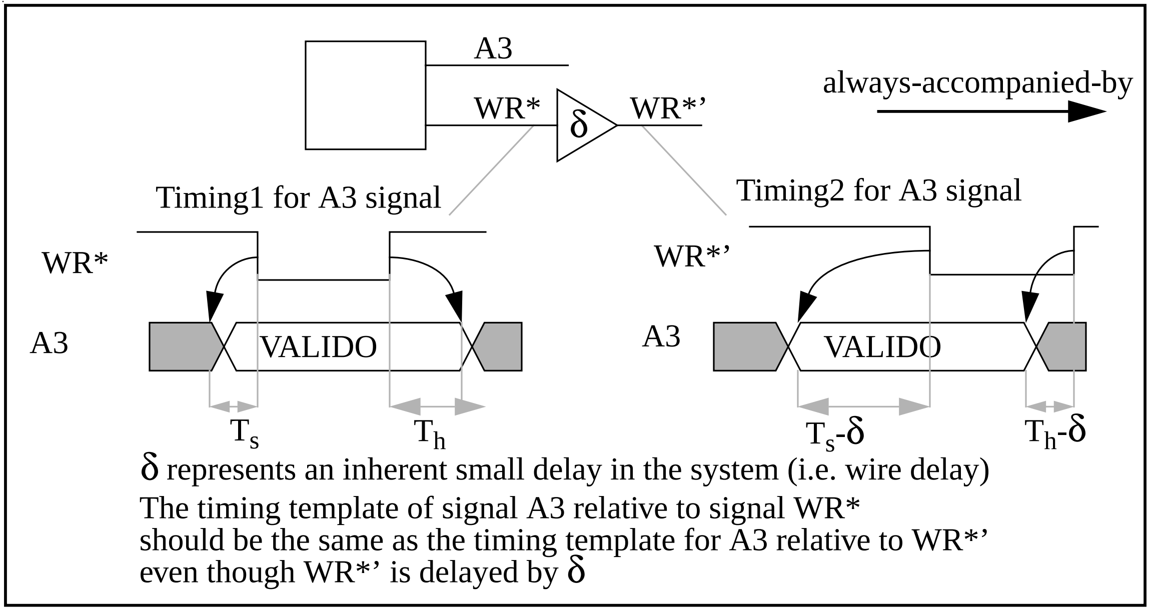

A3
WR*
δ
WR*'
always-accompanied-by
Timing1 for A3 signal
Timing2 for A3 signal
WR*
WR*'
A3
VALIDO
A3
VALIDO
T_s
T_h
T_s-δ
T_h-δ
δ represents an inherent small delay in the system (i.e. wire delay)
The timing template of signal A3 relative to signal WR*
should be the same as the timing template for A3 relative to WR*'
even though WR*' is delayed by δ

FIGURE 4-14. Propagation Delay Invariance of Timing Template (Reference is Delayed)

So far the timing templates presented consist of events, links between events and allowed range for the timing parameters for the links in the timing template. An example of a timing template was given in Figure 4-12 for the non-multiplexed signal timing template. Is the timing template shown in Figure 4-12 propagation delay invariant? The answer is no, which can be seen from a simple example illustrated in Figure 4-15. Assume

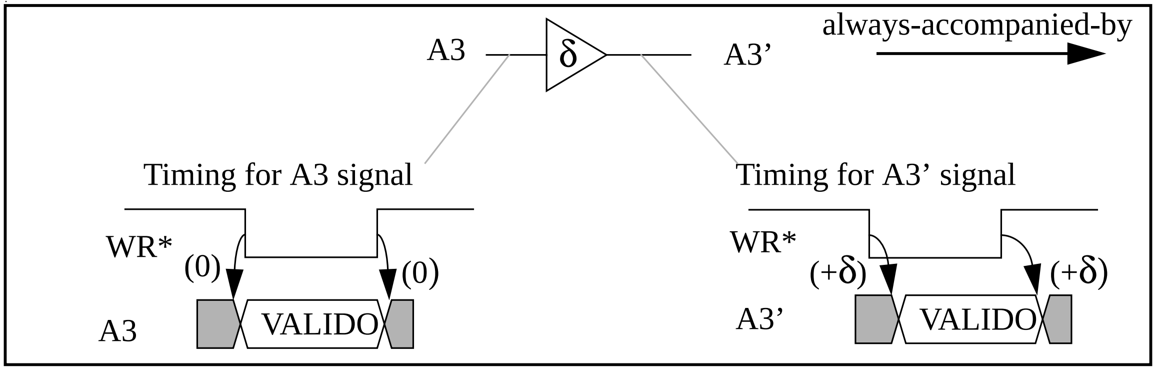


FIGURE 4-15. Simple Setup and Hold Time Example

there is a signal `A3` with a signal timing that has a setup and hold timing parameter of 0 relative to the timing reference `WR*`. Since 0 is included in the timing parameter range for the template setup and hold times of Figure 4-12, this timing does indeed follow the non-multiplexed timing template of Figure 4-12. If `A3` is delayed by a finite delay δ, the setup and hold times of the resulting `A3'` signal will change to +δ. A setup time of +δ violates the allowed range imposed on the timing template of Figure 4-12 on the setup time of -~ to

0 (i.e. +δ does not fall within the interval (-~ 0)). Due to this violation, the signal timing of the A3’ signal can not be based on the timing template of Figure 4-12.

To make the timing template presented in Figure 4-12 propagation delay invariant, it must be modified slightly as illustrated in Figure 4-16. To allow for the inherent delays in the system, the limits for the setup and hold time must be extended by an omp delay (written as -omp and +omp). This is shown with the special ◁——┤ symbol. The ——┤ indicates a range from the omp delay and the ◁—— indicates the range to negative infinity.

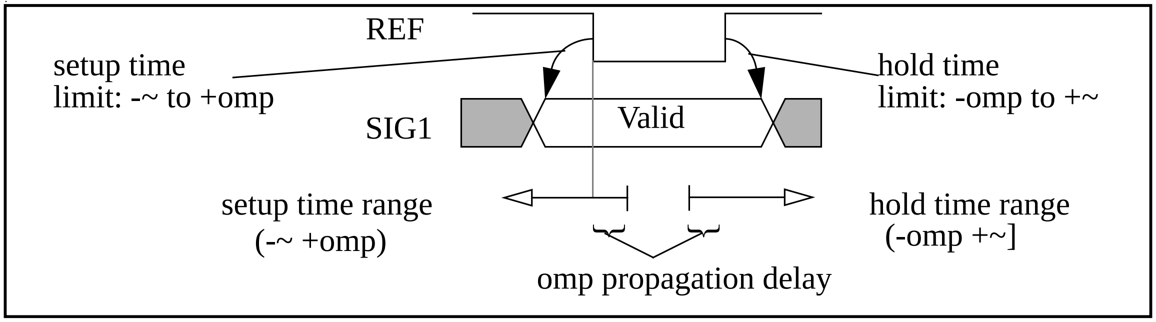


FIGURE 4-16. Updated Non Multiplexed Signal Timing Template

This section showed how a non-multiplexed timing template using always-accompanied-by links can be made propagation delay invariant by adjusting the limits of the allowed timing parameter range. When developing the component model timings presented in Section 4.8, all allowed timing parameters were investigated and adjusted to allow for propagation delay invariance. Non-causal timing links are adjusted by extending the allowed timing parameter limits by an omp delay, while causal timing links normally do not require any adjustment since their allowed range due to causality must be (0 +~).

4.7.4 The Component Model Timings

The signal timings developed for the component model fully specify the behavior of signals relative to one or more reference events. The *reference events* are detectable events that are fundamental to the operation of a capability. For example in data transfer there is always an event indicating that a data transfer operation has started and an event indicating that a data transfer operation is about to complete. *Fundamental to the operation of a capability* means that if the signals used to transfer some information are connected, the signals generating the reference events must also be connected. For example consider the connection of the address signals on a microprocessor and a memory device. Connection of the address signals alone is not enough to transfer the address information: some other signals (i.e. the signals that can be used for timing reference such as a data strobe) must also be connected. In this case, the signals that generate the timing reference events are considered fundamental to the operation of the capability.

4.7.5 Two Reference Event Timings for Data Transfer

This work develops a set of timing templates that can be used to represent the timing behavior of any information signal involved in data transfer. The signal timings developed are based on two reference events, the first event, ref+, represents the initiation of the data transfer, while the second event, ref-, represents the termination of the data transfer. The reference events are illustrated in timing diagrams as transitions on a *reference signal*. The reference is a virtual reference signal since the reference events are often generated by several signals, but only one signal is shown. The signal timing of any information signal is given relative to the two reference events. The reference consists of two complementary detectable events ref+ and ref-, while the information signal consists of complementary events sig+ and sig-.

The data transfer signal timings encountered in the microprocessor system components investigated are divided into two groups. *Non-interactive timings* give the timing behavior of an information signal relative to the two reference events while *interactive timings* give the timing behavior of an information signal relative to the reference events, and also the behavior of the reference events relative to the information signal events. Interactive timings are used to specify the timings of signals that are used in the overall control discussed in Section 3.4.2.2 on page 41. The difference between the two groups of timings can be seen in the direction of the timing links between the reference and information signal events: If there is a timing link from an information signal event to a reference event, the timing is an interactive timing, otherwise the timing is a non-interactive timing.

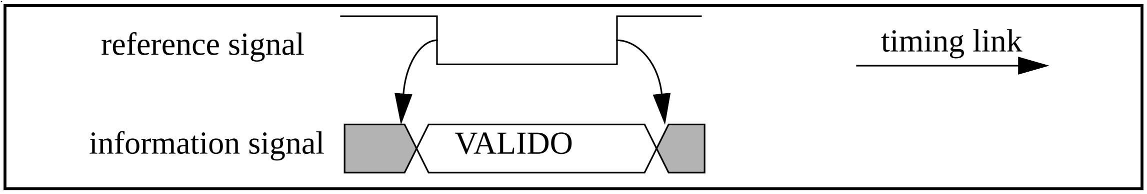


FIGURE 4-17. Non-interactive Timing Example

Figure 4-17 shows an example of a non-interactive timing, where the reference events have a timing link to the information signal events. This type of timing is typically found for address signals in microprocessor systems such as the `A1` signal of a MC68000 microprocessor. There are no timing links from the information signal events to the reference events.

Figure 4-18 shows an example of an interactive timing where the reference events have a timing link to the information signal events as in the non-interactive timing, and where an information signal event has a timing link to a reference event. A typical example for an interactive timing information signal is the `DTACK*` signal of a MC68000 microprocessor.

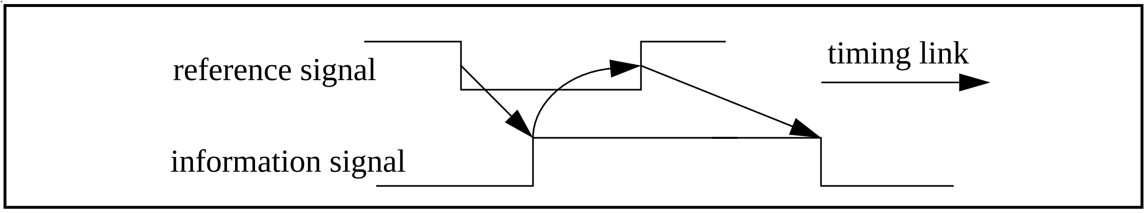


FIGURE 4-18. Interactive Timing Example

4.8 The Data Transfer Signal Timings

A signal timing for data transfer describes the relationship between two complementary events of an information signal (sig+ and sig-) relative to two timing reference events (ref+ and ref-). Figure 4-19 shows the timing links that are always assumed to be present between the ref+ and ref- events and the sig+ and sig- events unless otherwise specified.

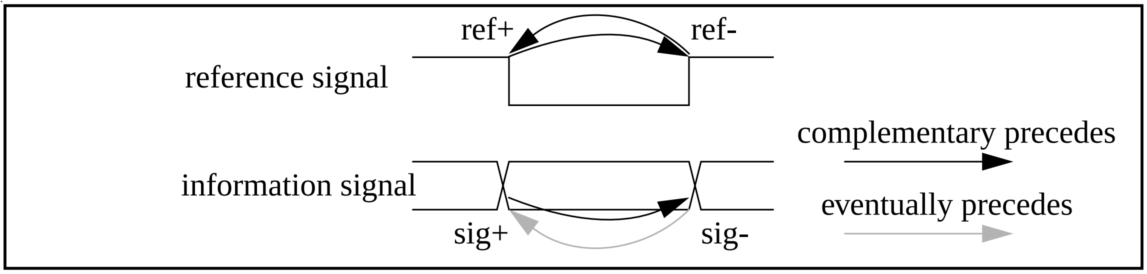


FIGURE 4-19. Theoretical Timing Relations

The signal timings described in this section are illustrated by showing the timing links between events other than those shown in Figure 4-19. The range of the allowed values for the timing parameter of the timing links are shown using the symbol ◁——┤ or ├——▷ with respect to the reference event, as explained in Section 4.7.3, if the timing parameter is bounded by infinity on one side and an omp delay on the other. For a

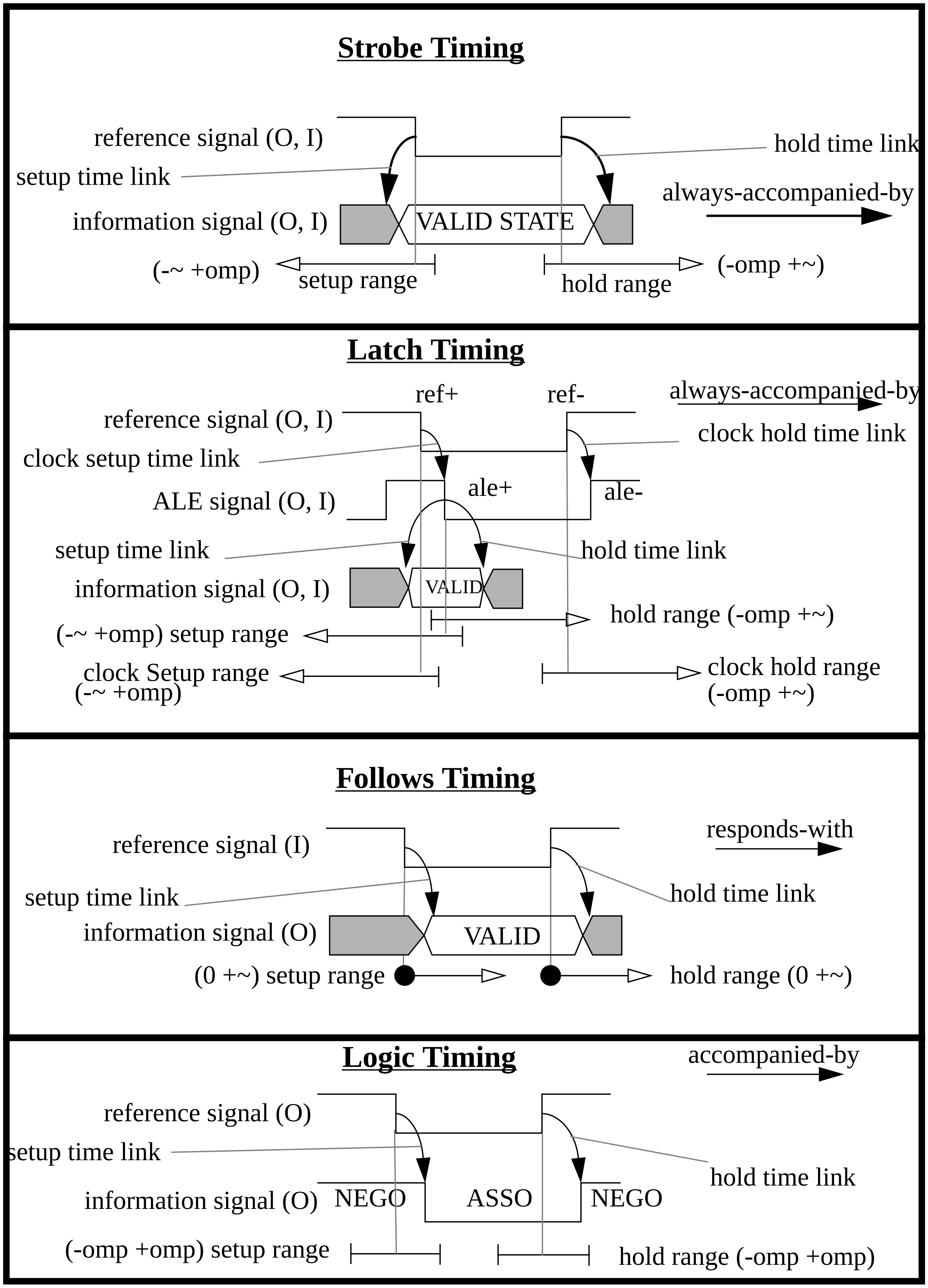
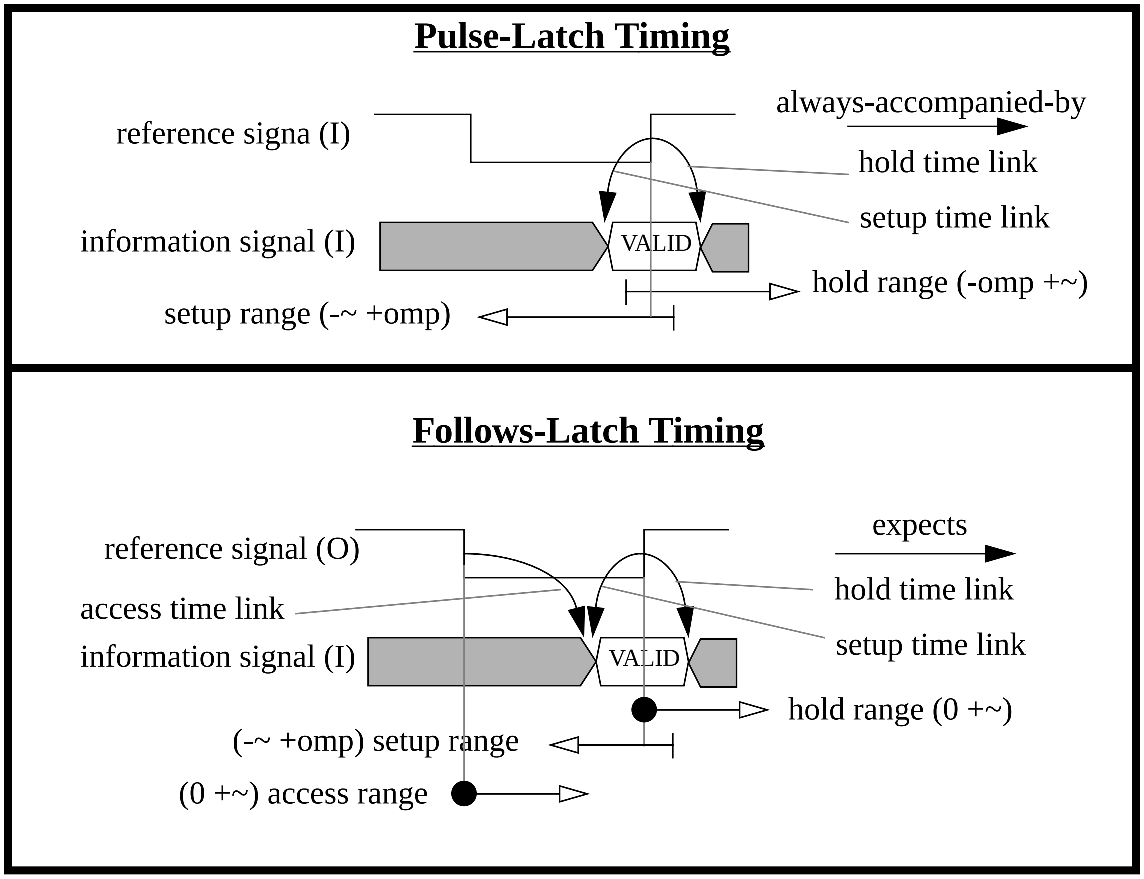


FIGURE 4-20. Non-Interactive Timing Templates - Part 1

causal timing link, where the timing parameter value is bounded by 0 one side and + infinity on the other, the symbol ●——▷ is used.

All signals shown in a signal timing template are marked with an O for output or an I for input, indicating the allowed direction of the signals with respect to the component.

For discussion purposes, all timing links in the signal timings presented are given names such as ‘setup time link’, ‘hold time link’, ‘response time link’, ‘acknowledge time link’ or ‘access time link’. The timing link names have slightly different meanings for the different timing templates, and must be discussed in the context of the timing template in which they are used. Figure 4-20 and Figure 4-21 present the non-interactive Strobe, Latch, Follows Logic, Pulse-Latch and Follows-Latch timing templates, while Figure 4-22 presents the interactive Handshake, Wait and Pulse timing templates. A detailed description of the different timing templates can be found in Appendix A.


Pulse-Latch Timing

Follows-Latch Timing

FIGURE 4-21. Non-Interactive Timing Templates - Part 2

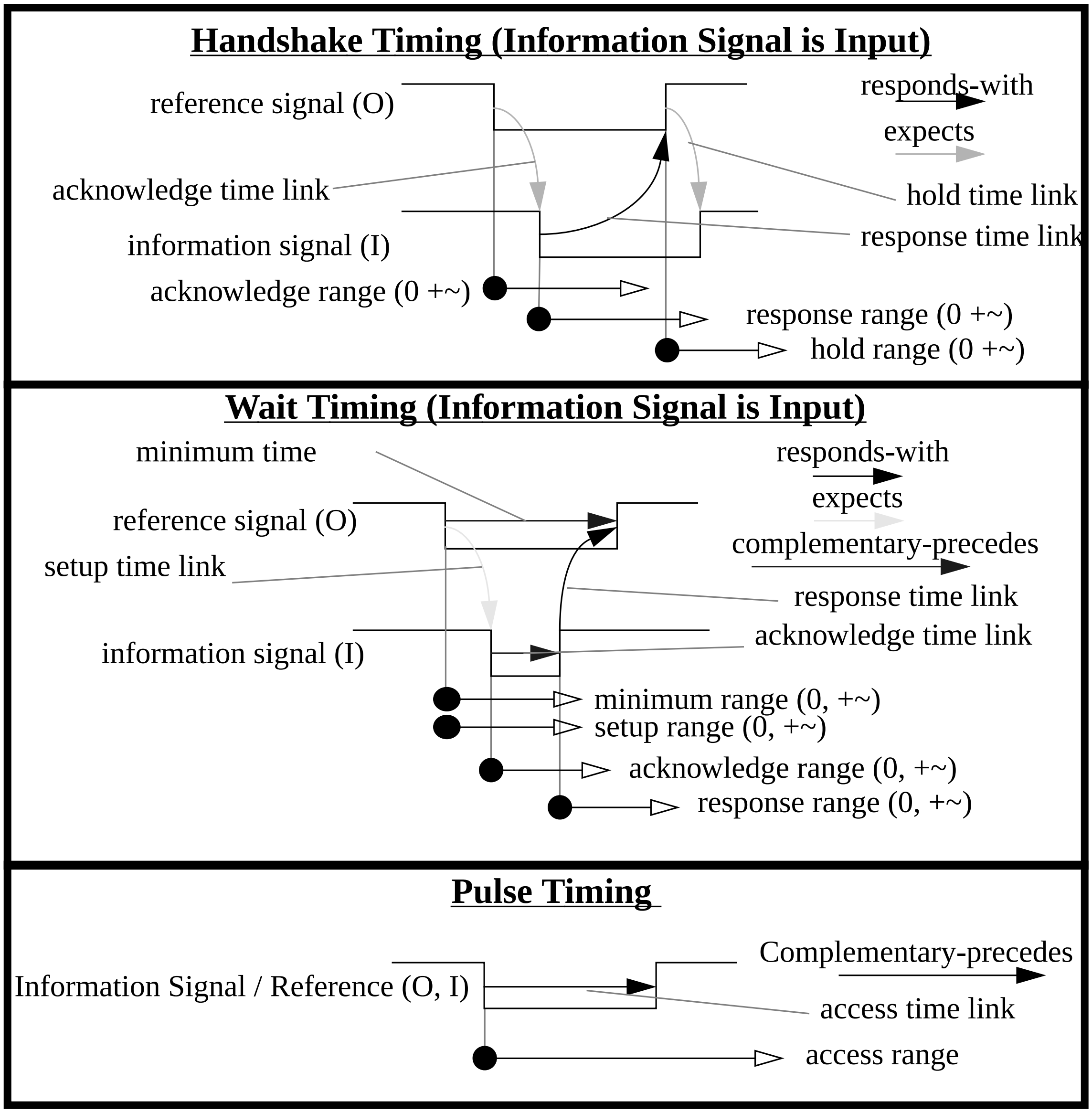


FIGURE 4-22. Interactive Timing Templates

4.8.1 Interactive Timings and the Initiate to Terminate Time Interval

The three interactive timings represent three fundamentally different methods used to adjust the time interval between the initiate and terminate events of the reference. Isolating the method of adjusting the initiate to terminate interval has two important advantages for modeling the signal timing of components and interface design:

First, it allows the concept of delay information to be developed. Delay information is information transferred between components that is used to adjust the initiate to terminate interval. For example on a MC68000, the DTACK signal is used to pass information to the MC68000 that indicates when it should terminate a data transfer. This work later develops universal techniques to connect information transfer, which can also be used to connect delay information.

Second, it allows separation of the interactive and non-interactive aspects of an information transfer. The separation provides a method of abstracting information transfer into more primitive and therefore simpler information transfers. For example, the description of a read data transfer of a MC68000 is often presented relative to the UDS* and the DTACK* signal as shown in Figure 4-23. The asserted UDS* event causes the memory device to supply the valid data after an interval (A). Once the memory device supplies the data on D15, DTACK* is asserted after interval (B). An interval (C) later, the UDS* signal is negated. Instead of a single timing behavior involving three signals, the timing behavior of the read data transfer is modeled as a Handshake Timing between the UDS* and DTACK* signals and as a Follows-Latch Timing between the UDS* and the D15 signal.

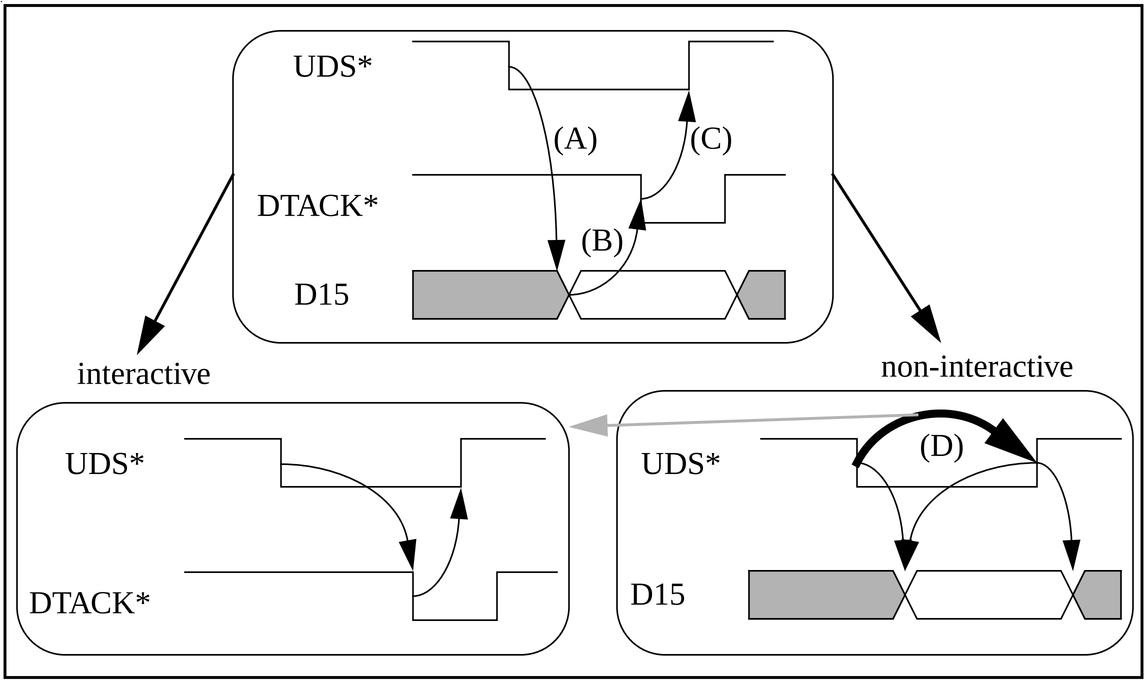


FIGURE 4-23. MC68000 Read Data Transfer

The three interactive timings represent different methods of specifying the initiate to terminate interval of the reference. The Handshake Timing specifies how the interval can be increased (from a lower limit), by delaying the time of occurrence of the asserted information signal event as shown in Figure 4-24(b). The Wait Timing specifies what the interval is if no information signal transition occurs as shown in Figure 4-24(c), and it specifies how the interval can be increased from a lower limit by delaying the time of occurrence of

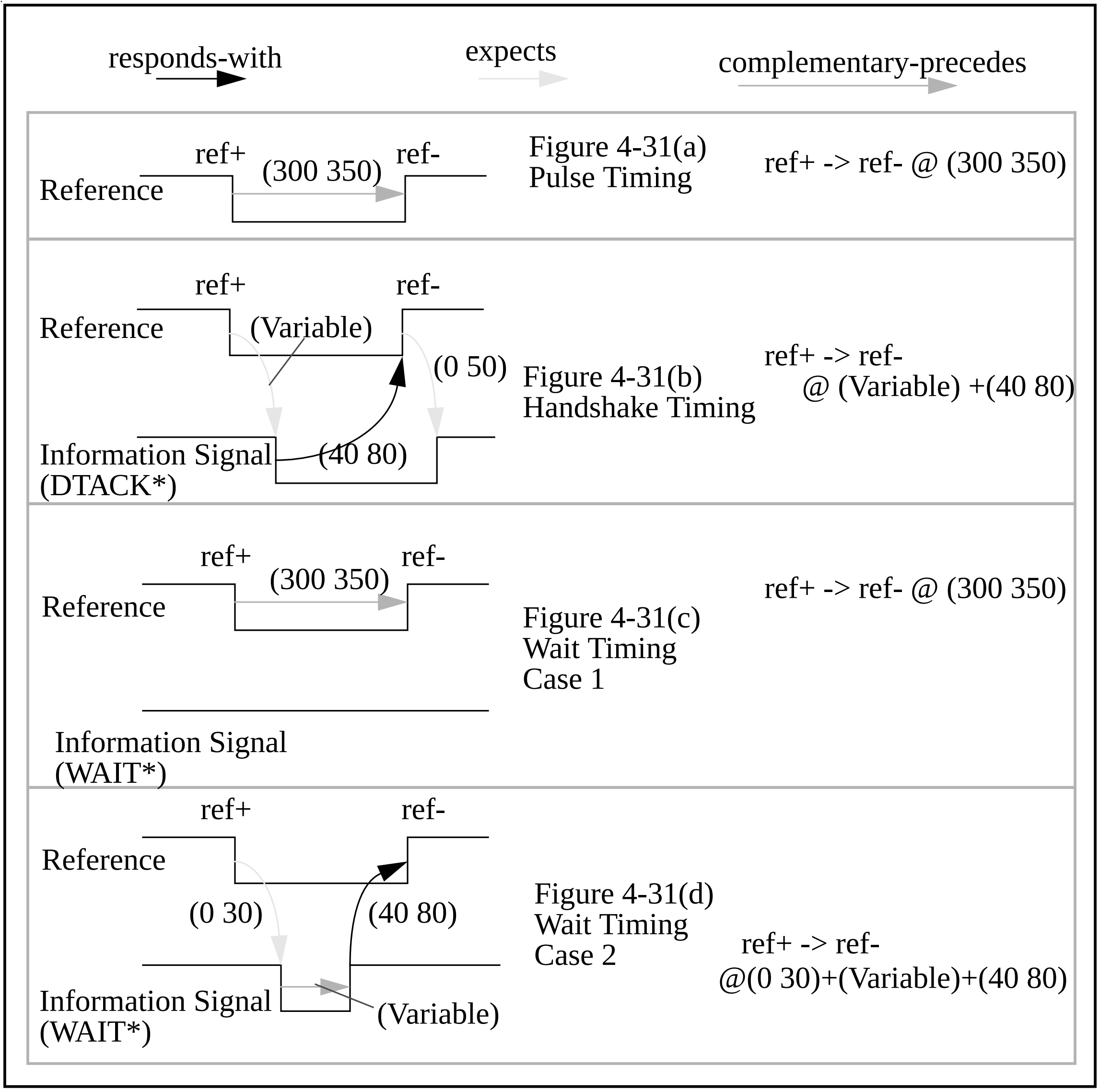


FIGURE 4-24. Initiate to Terminate Timing Link Example

the negated information signal event as shown in Figure 4-24(d). The non-interactive Pulse Timing simply specifies what the interval is, as shown in Figure 4-24(a).

4.8.2 Multiple Reference Signal Timings

The input reference on a device can consist of several signals. Often the timing parameters of timing links to information signals are given relative to initiate and terminate events of each of the reference signals. For example, Figure 4-25 shows the data access time T_1 and T_2 for a typical EPROM memory device relative to two signals, OE* and CE*. How is the reference consisting of multiple signals given? And how are the other signal timings given relative to this reference?

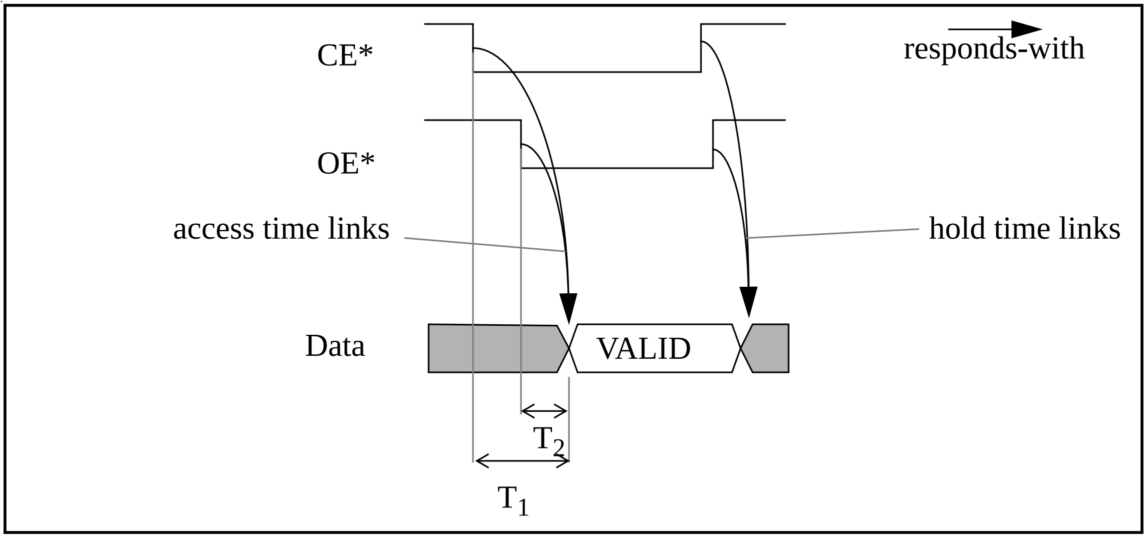


FIGURE 4-25. Data Access Timing for a Typical Slave Device

By investigating a signal timing such as the one shown in Figure 4-25, it was realized that the reference consists of the logical AND of the signals involved as shown in Figure 4-26. This resulted in the development of the concept of the AND signal timing:

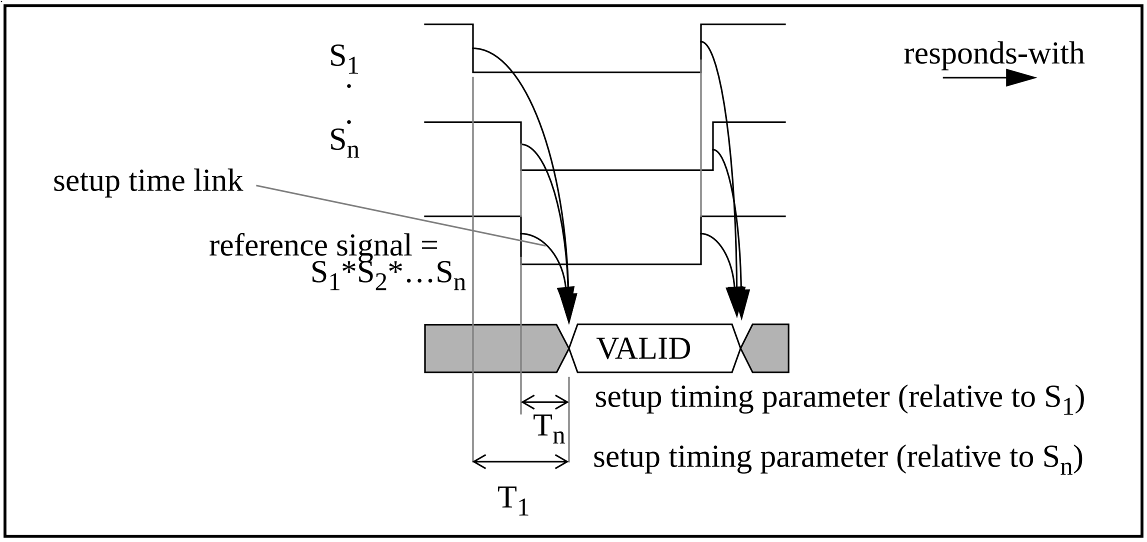


FIGURE 4-26. AND-Follows Timing

An AND signal timing provides a separate timing parameter for the timing links from each of the signals involved in the reference as shown. For example, the AND-Follows Timing given in Figure 4-26 indicates that the timing is a Follows Timing, with different access timing parameters supplied for each of the reference signals.

The following AND timings were found to exist: AND-Follows Timing (Figure 4-26), AND-Pulse Timing, AND-Strobe Timing, AND-Pulse-Latch Timing, AND-Latch Timing, AND-Handshake Timing, AND-Wait Timing.

AND timings are similar to the other timings described in this section except they have separate timing values associated with each reference signal. If one of the timings of a data transfer capability is given as an AND timing, all other data transfer capability timings for that device are also given as AND Timings.

4.8.3 Signal Timing Summary

A total of seven output signal timings (Table 4-4) and seven input signal timings (Table 4-5) were developed to represent the timing behavior of an information signal for data transfer. If the input timing reference consists of more than one signal, the input timings will be AND timings, which means separate timing parameter values are provided for events on each of the signals used for the reference. Of the total number of nine signal timings, six are non-interactive timings, while three are interactive timings which can be used to adjust and/or specify the initiate to terminate interval of the reference.

Strobe Timing
Follows Timing
Latch Timing
Logic Timing
Pulse Timing
Handshake Timing
Wait Timing

TABLE 4-4. Output Specification Timings

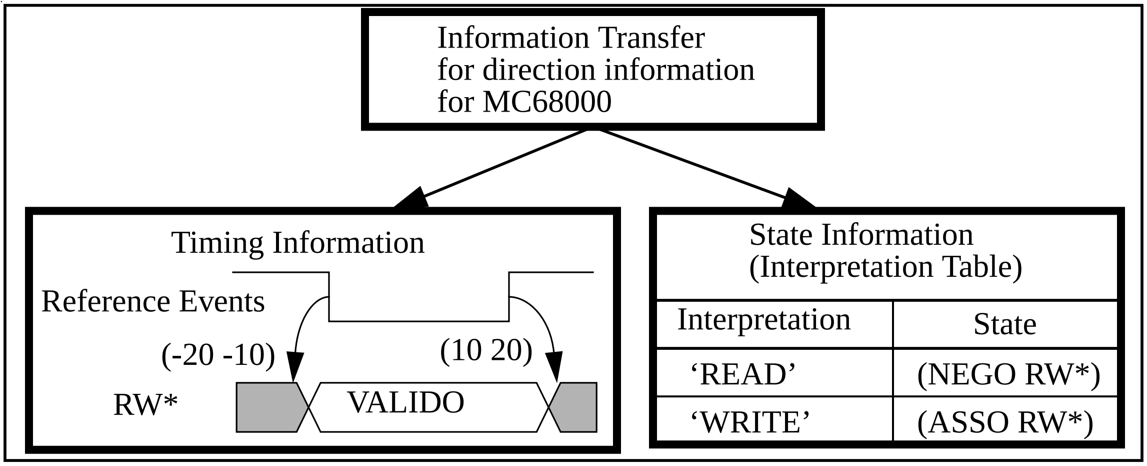
Strobe Timing
Latch Timing
Pulse-Latch Timing
Follows-Latch Timing
Pulse Timing
Handshake Timing
Wait Timing

TABLE 4-5. Input Requirement Timings

4.9 Modeling Information Transfer

Information transfer is the conveyance of information over signal wires. Information transfer over electrical signal wires requires the interpretation of some state that can be found on the signal wires and an indication of when the state can be interpreted. Thus, an information transfer is divided into two parts: the *timing information* indicating when certain states are present on the signal and a *state information* indicating the meaning or interpretation of the state. The models developed so far allow us to give the timing information of signals in the form of signal timings. The state information of a signal requires the attachment of an interpretation or meaning to a set of states.

The concept of attaching an interpretation to a set of states is best explained using an example. The information transfer associated with the direction of a data transfer for a MC68000 microprocessor is given in Figure 4-27. The timing information for this example is given by a signal timing that is based on a Strobe Timing template with a setup time of (-20 -10) and hold time of (10 20) as shown Figure 4-27. The timing information is used to indicate when the state information is transferred. The RW* signal is used to indicate a read if asserted and a write if negated. The state information for this information transfer is given by associating two keywords, ‘READ’ and ‘WRITE’, representing the concepts of reading from and writing to a component, with the appropriate states of the RW* signal.



Interpretation	State
‘READ’	(NEGO RW*)
‘WRITE’	(ASSO RW*)

FIGURE 4-27. Information Transfer Example

The division of the information transfer into timing and state information integrates well with the frame based data structures used for component representation. An information transfer frame has slots containing the names of the timing information frame and the state information frame. The state information frame is a table that associates some keywords with signal states. The Interface Designer will look for the keywords to perform a specific task, such as determining the state of a signal during a read operation.

4.10 Modeling the Data Transfer Capability

Data transfer is a capability whose specific purpose is to move specific information from one place to another in the microprocessor system. This well defined information will be called the *data* information.

To complete the transfer of the *data* information, there must be other information transfers associated with it that indicate when the *data* information should be transferred, how it should be transferred and where it should be transferred to/from. The description of how the different information transfers involved accomplish the data transfer is called the *protocol of the capability.* This section discusses a method of representing the protocol of the data transfer capability as a set of information transfers.

4.10.1 Organization of Data Transfer in a Microprocessor Systems

For data transfer to occur, requires a device to transmit the data, a device to receive the data, and a device to initiate and terminate the data transfer. For data transfer it was found that all devices that are capable of initiating data transfer also terminate the data transfer, thus the initiator and terminator are the same device.

Once the data transfer is initiated, the *data* information will eventually be transferred between two or more devices. There always will be one source for the *data* information and one or more destinations for the *data* information. To initiate a data transfer the initiator / terminator indicates to the transmitter to start the process of sending the *data* information to the receiver, while at the same time indicating to the receiver to be prepared to accept the *data* information from the transmitter. To terminate the data transfer, the initiator / terminator indicates to the transmitter and receiver that the *data* information transfer is about to be completed. It should be noted that the termination as discussed here does not refer to some indication that the transmitter has transmitted the *data* information or that the receiver has accepted/received the *data* information (i.e. an acknowledge). It simply indicates that the data transfer process is about to be completed. The sequence of initiation, transfer and the termination is called a *data transfer cycle*. The initiation and termination of the data transfer cycle can be recognized by events on the control signals. The initiation and termination events are used as the two reference events for all data transfer signal timings.

In most practical cases either the transmitter or receiver will also be the initiator/terminator of the data transfer. A device that can receive and transmit `data` information and is also the initiator/terminator is commonly called a *master.* A device that can receive and transmit `data` information but can not initiate and terminate the data transfer is commonly called a *slave*.

In this work we will only consider the data transfer between master and slave components. The techniques developed for interface design can be extended to data transfer for the more general receiver/transmitter model with a third device as the initiator/terminator.

4.10.2 Classification of the Data Transfer Information Transfers

In classical treatment of microprocessor systems [18][65][35], data transfer is partitioned into the `data` information itself, the address information which indicates where the data is transferred to/from, and the control information which includes any other information required to complete the data transfer.

When investigating data transfer for microprocessors it was found that often part of the information that indicates where data is transferred to/from consists of the address information plus some other information, such as information about the type of data space (e.g. supervisor or user). For this reason it was decided to include the address information with the control information.

The control information is classified into information sub-classes: request, direction, address, type, size, width and delay. It should be noted that the same physical signal can be used to transfer two or more different sub-classes of information.

The information supplied by the master that provides an indication of the direction of data transfer is the `direction` information, such as the `R/W` signal of the MC68000.

The location of `data` information is provided by using a linear address that represents an index to a location. This information is called `address` information, such as the `A1-A23` signals of the MC68000.

The `data` information transferred during a data transfer may be classified according to the type of information it represents. The classification of the information represented by the `data` information is indicated using `type` information supplied by the master such as the `FC0`, `FC1` and `FC2` signals of the MC68000.

The `data` information requested to be transferred often has different size from one transfer to the next, such as 8-, 16- or 32-bit words. The information associated with the size of the transfer is called `size` information and is supplied by the master such as the `SIZE0` and `SIZE1` signals of the MC68020.

There often can be more than one data path between two devices. For example, a 32-bit microprocessor has 4 separate 8-bit wide data paths available to transmit an 8-bit information word. The actual data path used to transfer data is selected by the slave in the form of `width` information, such as the `DSACK0* DSACK1*` signals of the MC68000. The `width` information indicates to the master how wide the actual `data` information path is and which path is used.

The `size` and `width` information is used to completely specify dynamically sized `data` information transfers. For example, the `size` information of a MC68020 microprocessor might indicate a 16-bit word size data transfer request (of a possible 8-, 16- or 32-bit word). The device that responds to the data transfer may only be able to transfer data in 8-bit words. The slave then indicates to the MC68020 that the transfer is only 8 bits using the `DSACK0*` and `DSACK1*` `width` information signals. To complete the transfer of the 16-bit word the MC68020 will then request the transfer of the second 8-bit word.

4.10.3 The Request Information

Conceptually, the initiation and termination events can be treated simply as information transferred between components. This information is called the *request information*. A design engineer will often connect the `request` information implicitly whenever signals are connected between components, since the `request` information is embedded in signals which have functions other than `request` information transfer. For example the `UDS*` signal on a MC68000 contains both the `size` information and `request` information.

Making the `request` information explicit provides a major advantage. It allows us to formalize the behavior of the `request` information in a manner similar to all the other information transfers as state and timing information. This makes it possible to manipulate and connect the `request` information explicitly using the same method developed for the other information transfers. Furthermore, the Interface Designer will be able to proceed with the connection of the other information transfer signals with the knowledge that the `request` information will always be connected.

A Logic Timing with a setup and hold time timing parameter of value (0) provides us with a simple method to give the timing of the *`request`* information. For example, in the MC68000, the signals involved in initiate and terminate events are the `UDS*` and `LDS*` signals. The MC68000 *`request`* information consists of the MC68000 *`request`* timing information and the MC68000 *`request`* state information as shown in Figure 4-28.

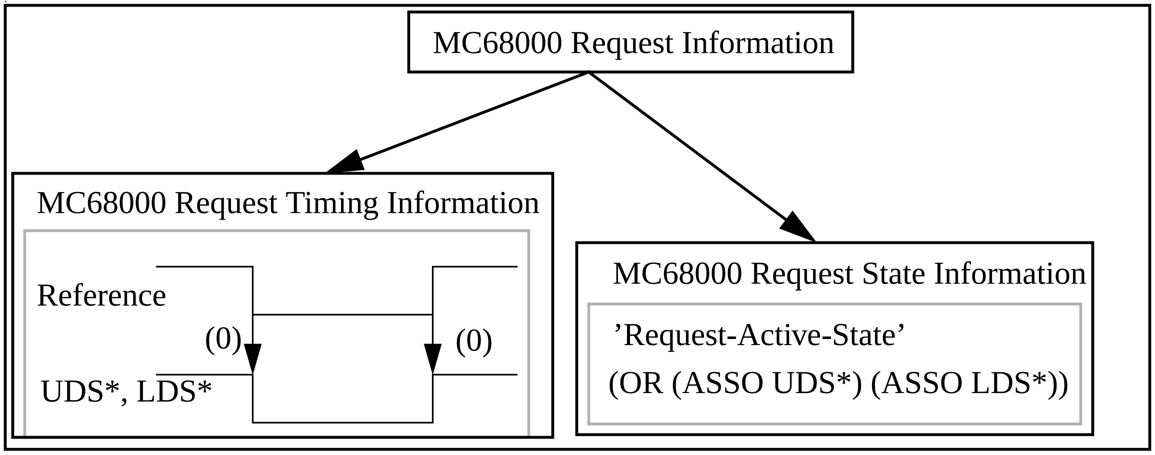

MC68000 Request Information
MC68000 Request Timing Information
Reference
(0)
(0)
UDS*, LDS*
MC68000 Request State Information
'Request-Active-State'
(OR (ASSO UDS*) (ASSO LDS*))

FIGURE 4-28. Request Information Example

4.10.4 The Delay Information

The fundamental events underlying all data transfer operations are the initiate and terminate events of the reference generated by the master. The method to control the time period from the initiate to the terminate event is called the *overall control*. The information transfer associated with the overall control is called the *`delay`* information. The concept of *`delay`* information arose when investigating the interactive and non-interactive aspect of an information transfer as explained in Section 4.8.1. The delay information represents the interactive aspect of the information transfer and is used to specify the time interval from the initiate to terminate event of the reference.

The *`delay`* information is classified into two types: *Overall asynchronous control* and *overall synchronous control*.

4.10.4.1 Overall Asynchronous Control

For overall asynchronous control the time between the initiate and terminate events of the reference is adjustable. The terminate event of the reference can not occur until after a *`delay`* information signal event occurs as shown in Figure 4-29. The adjustable time period is increased by delaying the `delay` information event (grey circle in Figure 4-29). The `delay` information for this type of overall control will be based on either a Handshake Timing or a Wait Timing. For overall asynchronous control, `delay` information flows from the slave to the master.

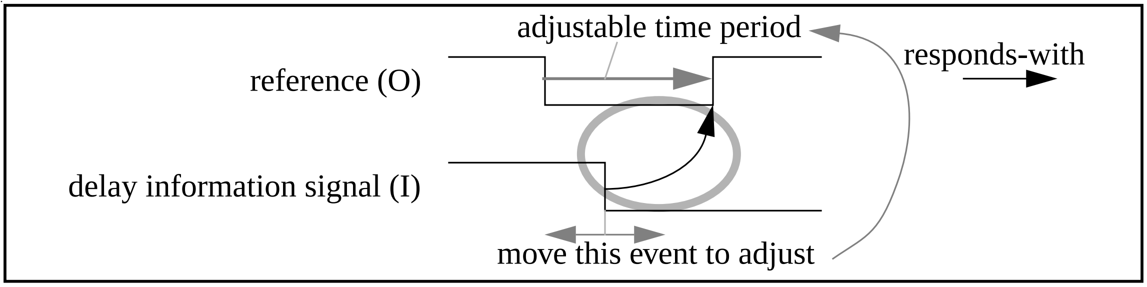


FIGURE 4-29. Overall Asynchronous Control

4.10.4.2 Overall Synchronous Control

For overall synchronous control the time period between the initiate and terminate events of the reference is fixed. Figure 4-30 shows a reference with a fixed time period between the initiate and terminate events. The `delay` information for a data transfer of this type will always have a timing of class Pulse Timing. For overall synchronous control, no information flows from the slave to the master. `Delay` information for overall synchronous control simply specifies the initiate to terminate interval.

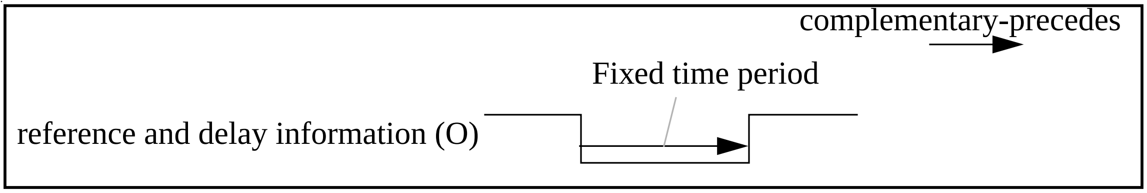


FIGURE 4-30. Overall Synchronous Control

4.10.5 Summary of Information Transfer between Master and Slave

All the control and the data information discussed has an information flow direction associated with it. Each control information either flows out of the master and into the slave or out of the slave and into the master as shown in Figure 4-31.

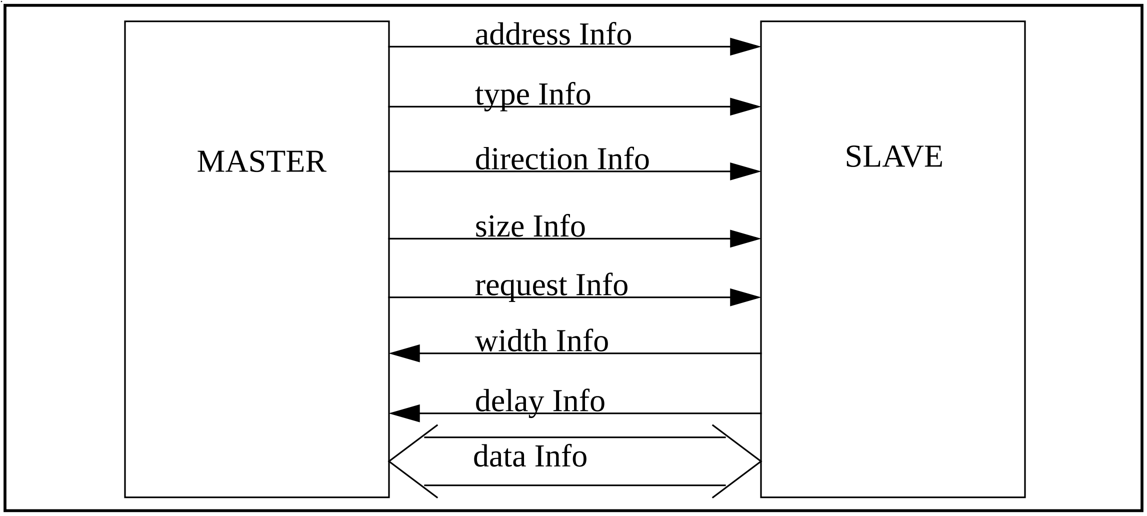


FIGURE 4-31. Information transfer between master and slave

4.11 Conclusions

A component model has been created which allows a hierarchial representation of the component. The protocol of a capability is given as a set of state-timing information transfers. A set of information transfer classes has been developed to represent the data transfer capability of all microprocessor components. For data transfer, the information transfer classes are *data*, *address*, *direction*, *type*, *size*, *width*, *request* and *delay* information. Details of the frames representation of the component can be found in Appendix B.1.

Each information transfer is given as state and timing information. The state information attaches meaning to the states of signals, while the timing information transfer provides information of how and when information is transferred. A method has been developed to represent similarities between timing behavior of different signals in the form of timing templates. All similar signal timings are represented by the same class of timing template. The timing templates are propagation delay invariant: This means that the timing template of a signal will not change from one end of a wire to the other. This allows component signal timings with similar timing behavior to be represented by the same timing template.

Chapter 5

Microprocessor System Interface Model

This chapter develops data structures to represent the interface connecting the components. The information organization is closely related to the hierarchial structure of the component model and the top down design methodology used to accomplish the interface design.

5.1 The Interface Block

An *interface block* (IB) represents the complete digital system that connects a capability of two or more components together. For example, Figure 5-1 shows an IB that connects the x capability of Component1 and Component2. These components could be VLSI devices such as microprocessors, memories, UARTs or any digital systems such as a laser printer. The IB will have all the information signals related to the capability of the devices flowing into and out of it.

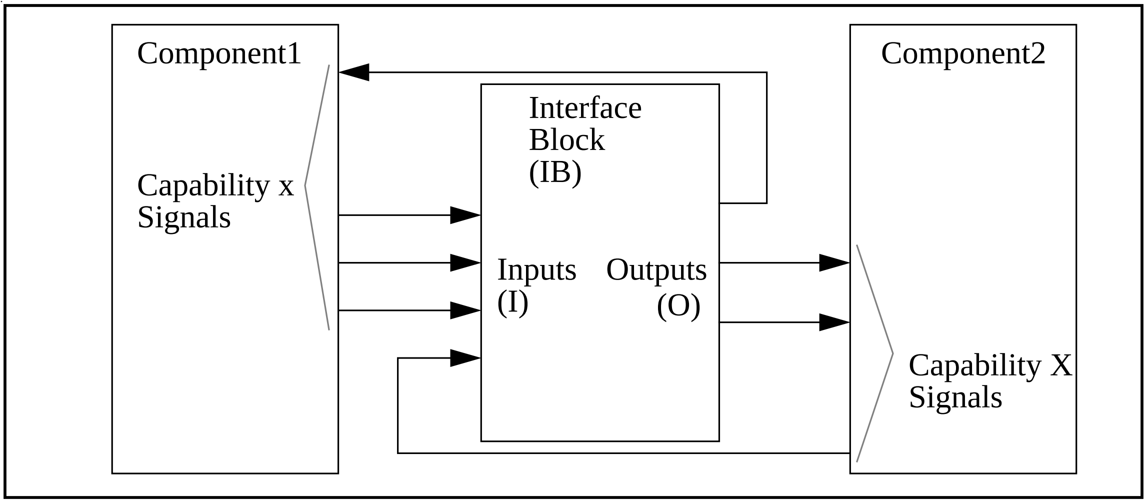


FIGURE 5-1. Interface Block (IB)

5.2 The Information Connection Interface Sub-Blocks

The protocol of a capability is given as a sequence of information transfers over the signal wires that connect to the signal pins of the devices. The IB is divided into interface sub-blocks (ISB) called *Information Connection ISBs* or simply *Info ISB*. Each Info ISB

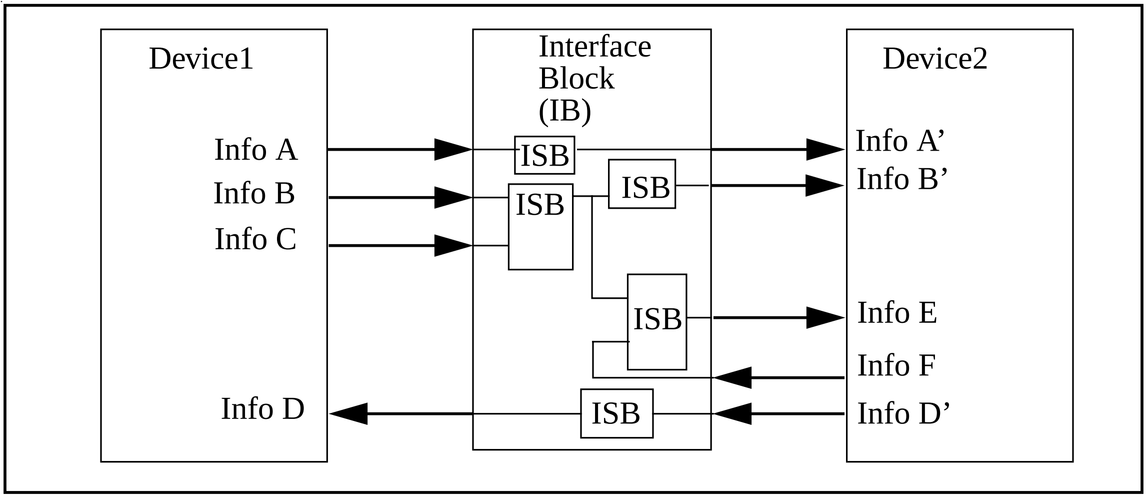


FIGURE 5-2. Information Connection Interface Sub-Blocks (ISB)

can have one or more information input port and a single information output port as shown in Figure 5-2. The protocol of a capability determines which information transfer input or output ports are connected by Info ISBs. The output from an Info ISB can go to the input of another Info ISB or to the input of one of the devices being connected.

5.3 Partitioning the Info ISBs

The intended function of the Info ISB is two-fold:

1. *State Conversion*: The states of the ISB information input port signals are used to generate the correct states on the output of the Info ISB.
2. *Timing Conversion*: The timings of the ISB information input port signals are used to generate the correct timing on the output of the Info ISB.

To accomplish the two functions, the Info ISB is partitioned into ISBs for state conversion and ISBs for timing conversion, called *State ISB* and *Timing ISB* respectively. The choice of using separate State and Timing ISBs is a natural one since the state conversion can be accomplished with a combinatorial circuit, while timing conversion is accomplished with a memory device such as a Flip-Flop. This can be seen with a simple example. A microprocessor with a multiplexed address bus and a memory with a non-multiplexed inverted address bus are to be connected. To accomplish the timing conversion, the microprocessor address bus must be demultiplexed (this is usually done with a transparent D-Latch). To accomplish the state conversion the address signals must be inverted (this is usually done with an inverter).

The State and Timing ISBs will take the input signals of the Info ISB, make the appropriate conversions, and then generate the output signals. Figure 5-3 shows three ways of organizing the state and timing conversion within an Info ISB. In Figure 5-3(a) the timing conversion is performed first, followed by a state conversion. In Figure 5-3(b) the state conversion is performed first, followed by the timing conversion, while in Figure 5-3(c) the timing and state conversion is performed in parallel. Attempting to perform the state conversion and timing conversion in parallel results in a design dilemma: After the timing and state conversion, two signals will exist which must now be used to produce a single output signal as shown in Figure 5-3(c). It is difficult, if not impossible, to define what the block marked ‘Combine?’ in Figure 5-3(c) has to accomplish.

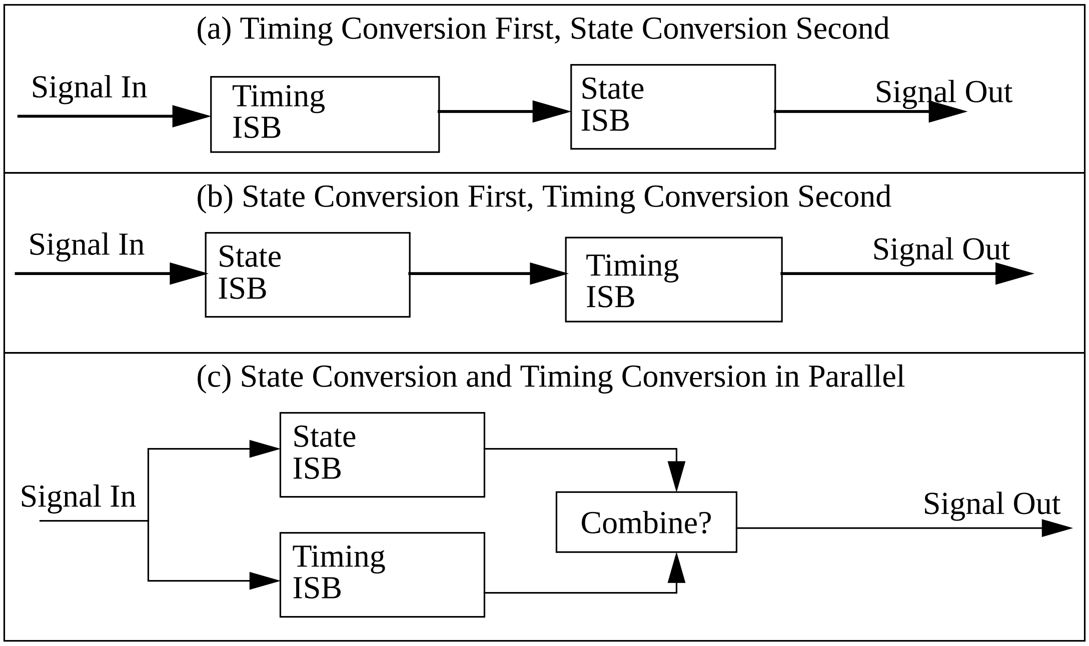

(a) Timing Conversion First, State Conversion Second
Signal In
Timing ISB
State ISB
Signal Out
(b) State Conversion First, Timing Conversion Second
Signal In
State ISB
Timing ISB
Signal Out
(c) State Conversion and Timing Conversion in Parallel
State ISB
Signal In
Combine?
Signal Out
Timing ISB

FIGURE 5-3. Timing and State Conversion Order

Figure 5-4 shows the organization of the Info ISB that was chosen to accomplish the task of timing and state conversion. The timing conversion is performed on each signal entering the Info ISB. The output of each Timing ISB goes into the State ISB.

The order of the State and Timing ISBs was chosen to allow the timing behavior of each signal to be fine-tuned before the state conversion takes place. This imitates the way a designer will normally accomplish the task of timing and state conversion. For example, assume a designer must generate a decoded address signal from a multiplexed address bus. Normally the designer will demultiplex the address signals (timing conversion) and then pass the demultiplexed address signals into the address decoder (state conversion).

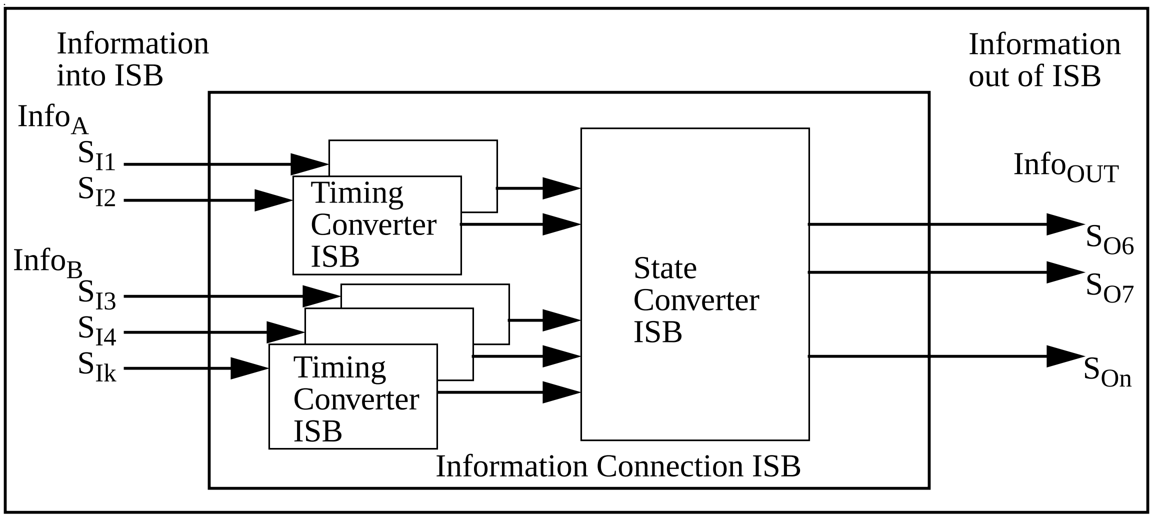


FIGURE 5-4. Details of Information Connection ISB

5.3.1 The Timing ISBs

The output signals of one device will exhibit certain timing behavior, and the other device will require its input signals to have a certain timing behavior. To connect the two devices, the timing behavior of the output information must be translated (modified) using a Timing ISB so that all timing requirements of the device receiving the information are satisfied. There are basically only two methods of changing the timing of a signal: A clocked memory device and a pure delay.

A *clocked memory device* can store the state of a signal until the occurrence of an event called a clock event.

A *pure delay* takes an event and delays it a certain amount of time (e.g. a wire, buffer, delay line or, under some circumstances, a combinatorial circuit). A pure delay will generate an identical version of the event sequence, except delayed in time.

The signal timings developed in this work give the general timing behavior of signals in the form of timing templates and the specific timing behavior of the signals in the form of timing parameters. The pure delay and clocked memory device are used to change different aspects of a signal timing. A clocked memory device such as a D-Latch will normally change the timing template a signal is based on. A pure delay such as that from a wire, buffer or combinatorial circuit will normally only change a signal’s timing parameters, while preserving a signal’s timing template. A signal’s timing template is preserved for a small pure delay, since the timing templates were designed to be small delay invariant as discussed in Section 4.7.2.

For example, Figure 5-5 shows the effect of a clocked memory device and a pure delay on the timing of a signal `SIG1`. Figure 5-5(a) shows the signal `SIG1`, which is based on a Latch Timing template passing into a transparent D-Latch (a clocked memory device) to produce signal `SIG2`. The signal timing of `SIG2` is completely different than that of the `SIG1` signal and is based on a Strobe Timing template. Figure 5-5(b) shows the information signal `SIG1`, passing through a buffer with delay *d* to generate `SIG3`. The timing of `SIG3` is also based on a Latch Timing template and the timing parameters are increased by the amount of the delay *d*.

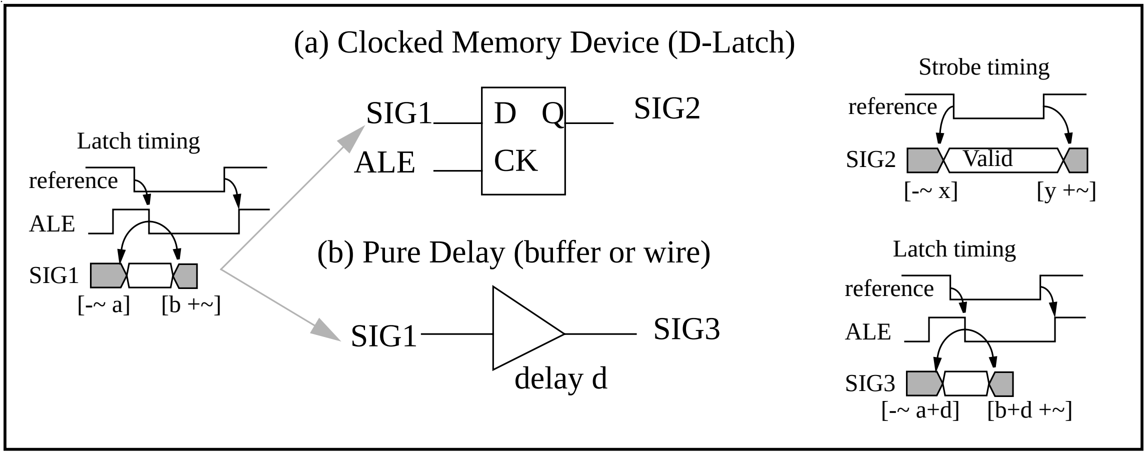


FIGURE 5-5. Effect of Pure Delay and Clocked Memory Device on a Timing

The above discussion of the effect of pure delay and clocked memory devices on the timing behavior of a signal is general and should be used as a guideline. There are exceptions where a pure delay changes the timing template of a signal (for example if the pure delay is longer than an omp delay), or where a clocked memory device preserves the timing template of a signal and only changes the timing parameters.

The Timing ISB is designed by analyzing the required ISB output timing and the available ISB input timing and choosing either a clocked memory device or a pure delay to accomplish the appropriate timing conversion.

5.3.2 The State ISBs

The output signals of one device have certain states, and the other device will require its input signal to have certain states, with a meaning associated with each state. The meaning of a state refers to items such as a specific address for address information, or a specific direction for direction information. To accomplish the connection of the state information, the state information between two devices must be translated using some kind of circuitry, so that they are compatible with each other.

For most information exchanges in microprocessor systems the translation involves only logical operations on asserted or negated signal states, and simple combinatorial logic can be used for the state translation. The combinatorial state translation involves a n-input to m-output combinatorial circuit, which can be generated using m n-input, 1-output combinatorial circuits as shown in Figure 5-6[37].

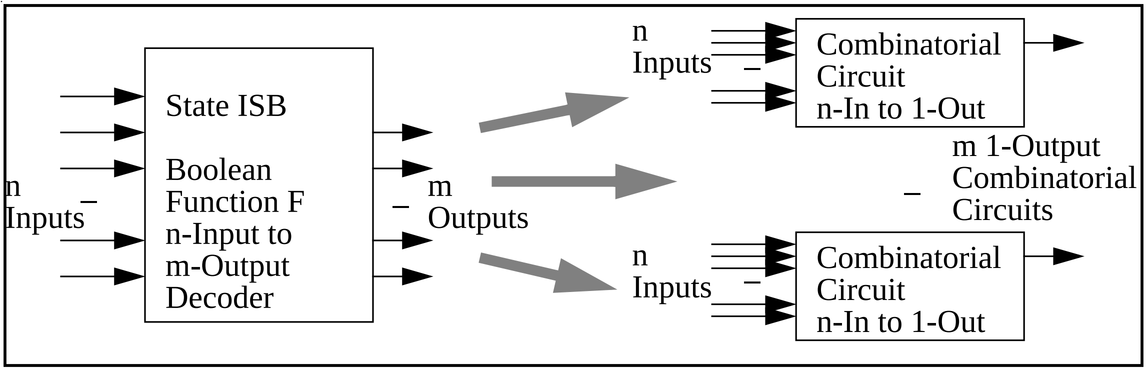

State ISB
Boolean Function F n-Input to m-Output Decoder
n Inputs
m Outputs
n Inputs
Combinatorial Circuit n-In to 1-Out
m 1-Output Combinatorial Circuits
n Inputs
Combinatorial Circuit n-In to 1-Out

FIGURE 5-6. Combinatorial State

Sometimes the state translation involves states other than ASSERTED and NEGATED. In these cases specialized hardware, called *level converters*, must be utilized for the state translation. For example, a tri-state buffer can be used to change the state of a signal from ENABLED to OPEN as shown in Figure 5-7.

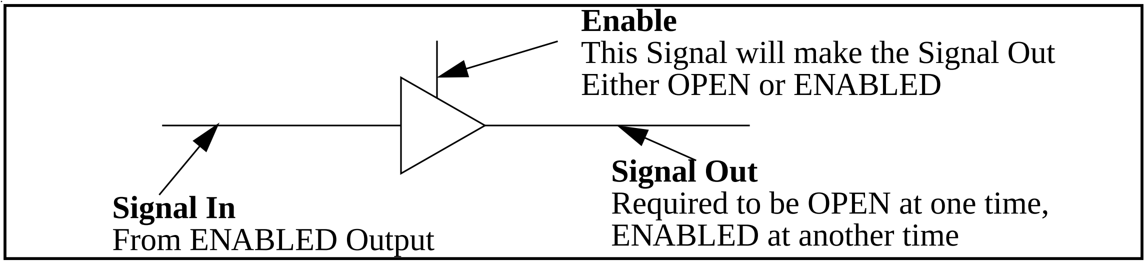


FIGURE 5-7. Tri-state Buffer

The State ISB is designed by analyzing what the required output state of the ISB, and inserting the appropriate State ISB made up of either combinatorial circuits or specialized signal level converters such as tri-state buffers.

5.4 Interface Sub-Block Primitive Circuits

The IB is organized into a hierarchial series of ISBs as shown in Figure 5-8. At each level of abstraction, the ISBs become more and more detailed. At the lowest level, the ISBs are simple digital circuits with known, well defined behavior. These simple digital circuits are devices such as AND gates, OR gates, flip-flops or buffers. They are the building blocks from which the ISBs are ultimately constructed. These digital circuits are called *ISB primitives* (ISBP).

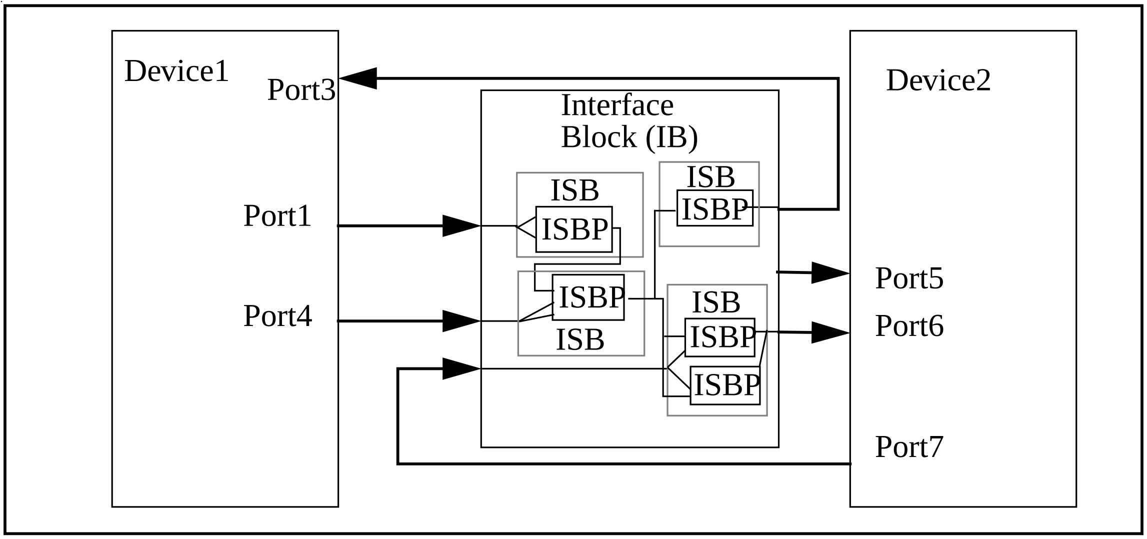


FIGURE 5-8. Interface Block Organization

An ISBP is a simple, well defined digital circuit. The circuit can have any number of input and output signals. It is considered primitive since its input / output behavior is known, and it can not be broken into smaller sub-circuits by the Interface Designer. There are basically three types of ISBPs that are allowed in an ISB and that are primitive:

- *Combinatorial primitive circuit*, which includes any combinatorial circuit without feedback.
- *Memory primitive circuit*, which includes flip-flops, latches and delay elements.
- *Level Conversion Primitives*, which includes tri-state buffers and open collector drivers.

The choice of which ISBPs to use to build up the interface is up to the interface designer. The knowledge about which ISBP to use to manipulate state and timing information is represented with design rules.

5.4.1 Common ISBPs and their Behavior

A set of eight ISBPs was developed for the Interface Designer. The set of ISBPs was chosen to facilitate the design of all possible State and Timing ISBs the Interface Designer may encounter. For example, the state conversion is normally accomplished using a combinatorial circuit, thus the combinatorial circuit was chosen as one of the primitives. Similarly, the timing conversion to convert a multiplexed signal to a non-multiplexed signal is normally accomplished by a D-Latch, thus the D-Latch was chosen as one of the ISBPs. All state and timing conversion possibilities between IB inputs and outputs were examined, resulting in the ISBPs presented in this section. The set of ISBPs chosen is not unique, and it is possible to develop a different set ISBPs to facilitate the design of the State and Timing ISBs.

The ISBP's behavior is divided into two domains: the value domain and the time domain. The *value domain* deals only with the logic function of an ISBP. It represents the transformation of I/O values and ignores the I/O timing relations. The logic function is what a designer generally remembers about a device without looking up specific details in a data book. For example take a D-Flip-Flop: a designer usually will remember that the data input will be transferred to the Q output on a clock transition, but he will often not remember what the specific setup and hold times are. This knowledge represents the logic function. The *time domain* deals with the I/O timing relations such as event propagation delays and signal setup and hold times for any signal going into and coming out of the ISBP. Values associated with the I/O timing relations of an ISBP are referred to as *ISBP parameters*.

In the Interface Designer, the ISBPs are represented using a VHDL [2] behavior description which defines their behavior precisely. There are four advantages to represent the ISBPs using VHDL: First VHDL provides a well developed, standardized method of describing a digital circuit using familiar programming language forms. Second, specifying the ISBPs using VHDL allows the complete structure of the designed interface to be given using VHDL. This means that once the frame based IBs and ISBs are completed they can easily be translated into structured VHDL code which utilizes the VHDL ISBPs. Third, VHDL provides a method of simulating the behavior of a design without the delay and expense of hardware prototyping. And fourth, the VHDL description of the interface can be used for synthesis: The actual hardware can be synthesized using a VHDL synthesis tool. There are other hardware description languages, such as VERILOG, that could be used to describe the ISBPs. VHDL was chosen due to its availability and ever increasing industry support.

The final output of the Interface Designer will be the description of an implementation of the microprocessor system interface. By representing the ISBPs using VHDL several methods of implementing the microprocessor system are possible. The VHDL description can be manually converted to an implementation using discrete logic such as TTL gates, since the ISBPs were chosen to easily map to available TTL devices. The VHDL description can also be used for automatic synthesis of the interface for different target implementation platforms, using field programmable devices such as PALs, XILINX programmable logic devices, Alterra EPLDs, or custom VLSI devices such as gate arrays.

The ISBPs developed for this work are presented in this section. For each ISBP, a circuit diagram logic symbol is provided, showing its inputs and outputs. A timing diagram is used to indicate relationships in the form of events and states between the input and output of the ISBP. Finally, the VHDL code description is given for each ISBP in the form of an entity definition and an architecture for each entity. Following commonly used terminology a device described in VHDL is called an *entity*, while the description of its operation of the device is called its *architecture*.

5.4.1.1 Combinatorial ISBP

Any single output combinatorial circuit of arbitrary complexity, without feedback, is a Combinatorial ISBP. The behavior of the Combinatorial ISBP is represented by its Boolean equation F, which is a function of the inputs $I_1 \ldots I_n$ as shown in Figure 5-9 and the propagation delay T_{PD}. The effect of any changes on the inputs of the Combinatorial ISBP

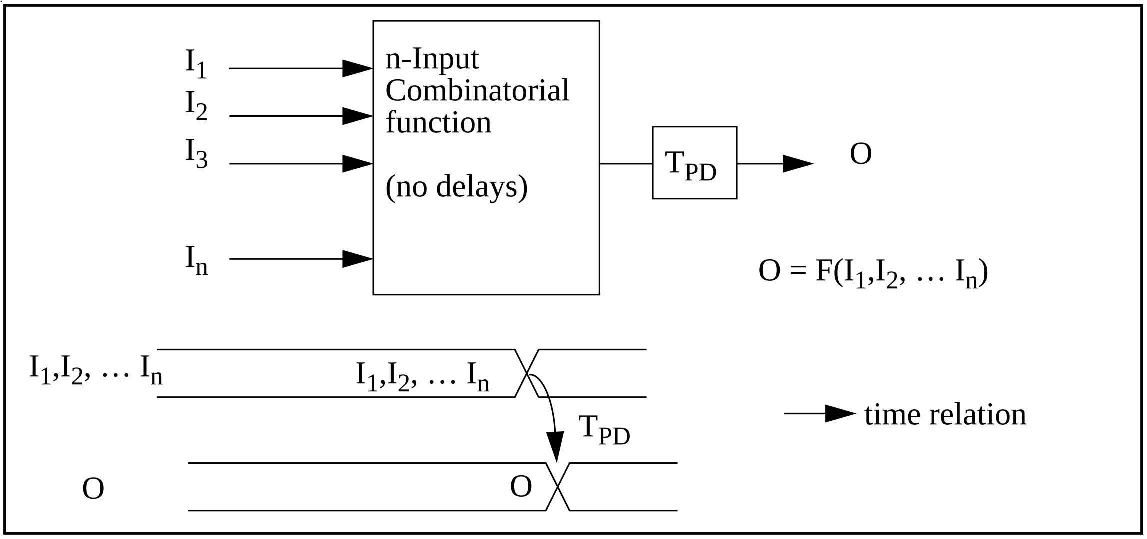


FIGURE 5-9. Behavior Model of Combinatorial ISBP

propagate to the output after a delay T_{PD}. A VHDL behavior model of a 2-input AND Combinatorial ISBP is given in Table 5-1. The Combinatorial ISBP is also used within the description of other ISBPs such as the Leading Edge Delay.

```
entity AND2 is
   generic(tpd : TIME :=10ns);
   port(IN1, IN2: in STD_LOGIC; OUT1: out STD_LOGIC);
end AND2;

architecture BEHAVOIR of AND2 is
begin
      OUT1 <= IN1 and IN2 after tpd;
end BEHAVOIR;
```

TABLE 5-1. VHDL Behavior Model of 2 Input AND ISBP

5.4.1.2 D-Flip-Flop Clocked Memory ISBP

An edge triggered D-Flip-Flop is a simple memory ISBP with two inputs (D and CLK) and one output (Q). It was chosen because it is simple to analyze and it can be used to build other edge triggered Flip-Flops such as J-K-Flip-Flops.

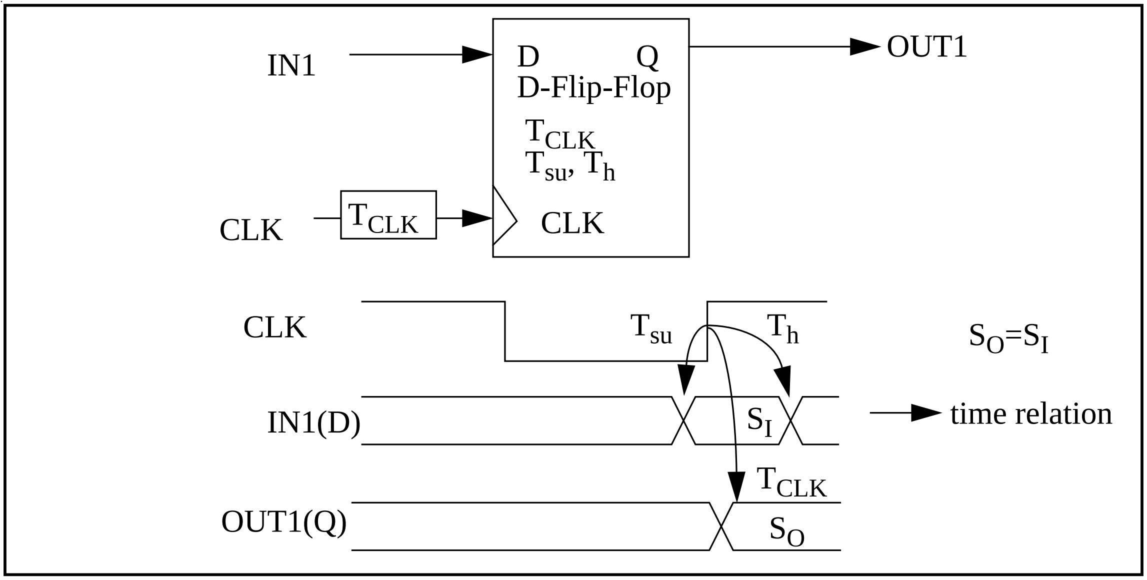


FIGURE 5-10. Behavior Model of Edge Triggered D-Flip-Flop ISBP

An edge triggered D-Flip-Flop latches the input data on the NEGATED to ASSERTED clock edge as shown in Figure 5-10. A VHDL behavior model of the D-Flip-Flop ISBP is given in Table 5-2. The time domain behavior of the D-Flip-Flop includes three time relations shown Figure 5-10: T_{su} and T_h are the input setup and hold time requirements, while T_{CLK} is the clock edge to output delay (tclk in VHDL code). The

```
entity D_Flip_Flop is
  generic(tclk : TIME :=10ns);
  port(IN1, CLK : in STD_LOGIC; OUT1: out STD_LOGIC);
end D-Flip-Flop;

architecture BEHAVOIR of D_Flip_Flop is
begin
  process (CLK)
  begin
    if rising-edge(CLK) then
      OUT1<=IN1 after tclk;
    end if;
  end process;
end BEHAVOIR;
```

TABLE 5-2. VHDL Behavior Model of D-Flip-Flop ISBP

VHDL D-Flip-Flop shown here is a simplified version of the one actually used. The complete VHDL code for the D-Flip-Flop, which also includes a reset signal, is shown in Appendix C.2.6. The current Interface Designer does not use the D-Flip-Flop directly, but uses it to implement Pure Delay ISBPs (discussed below). Once the capabilities of the Interface Designer are expanded, the D-Flip-Flop ISBP could be used to build output registers or to synchronize an asynchronous signal to a system clock.

5.4.1.3 Other ISBPs

The other ISBPs developed are the D-Latch (Figure 5-11), Pure delay (Figure 5-12),

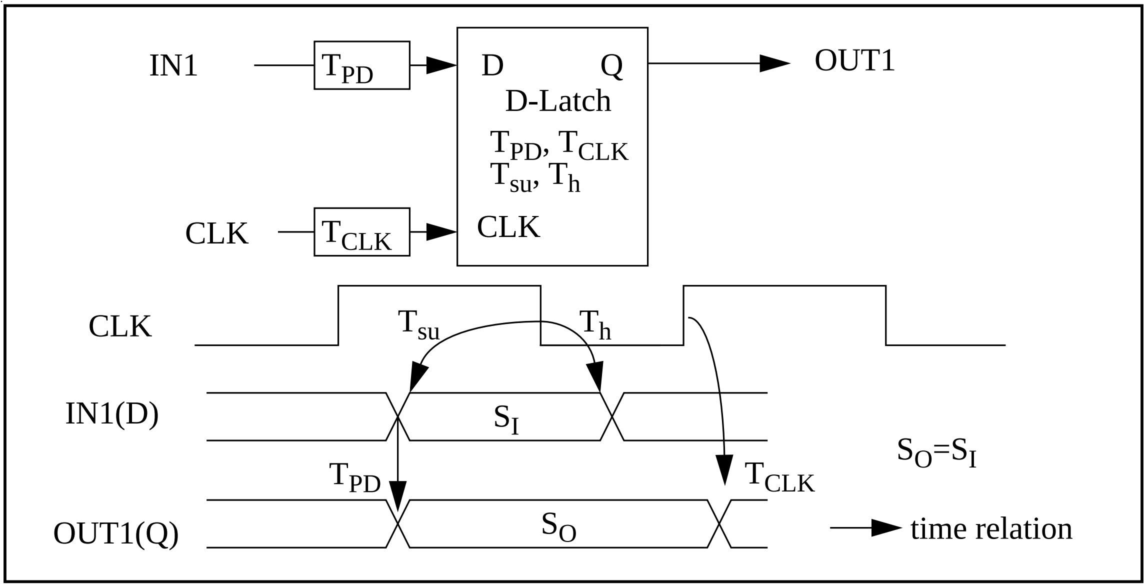


FIGURE 5-11. Behavior Model of D-Latch ISBP

Leading Edge Delay (Figure 5-13), Trailing Edge Delay (Figure 5-14), Tri-State Buffer

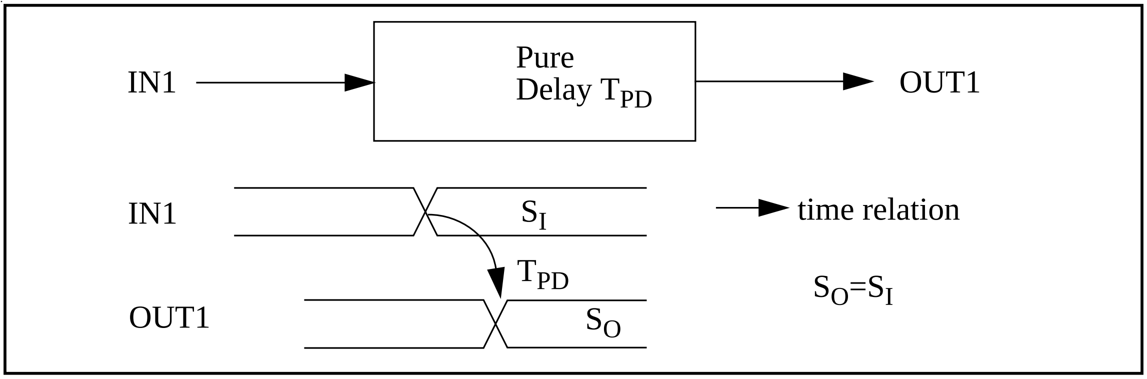


FIGURE 5-12. Behavior Model of Pure Delay ISBP

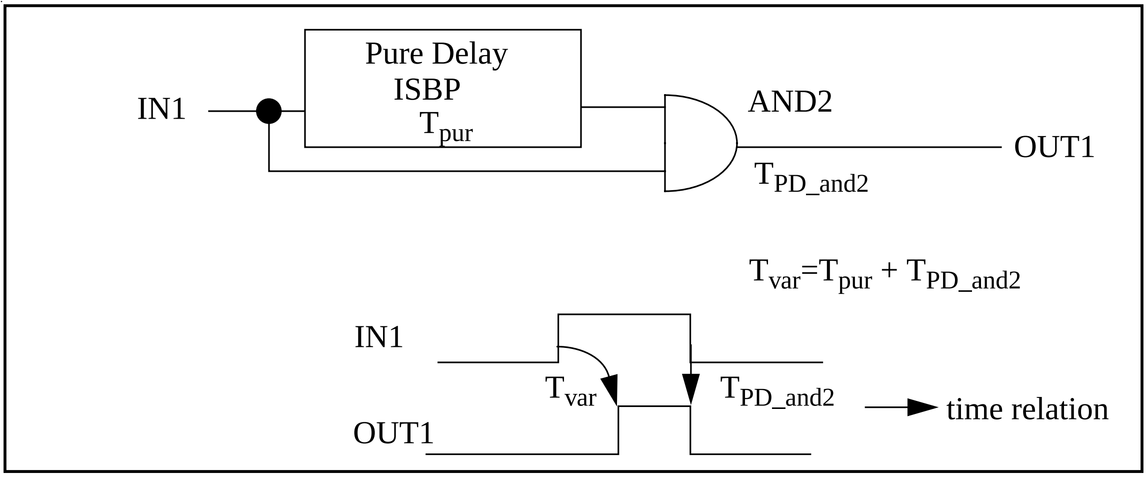


FIGURE 5-13. Behavior Model of Leading Edge Delay Primitive

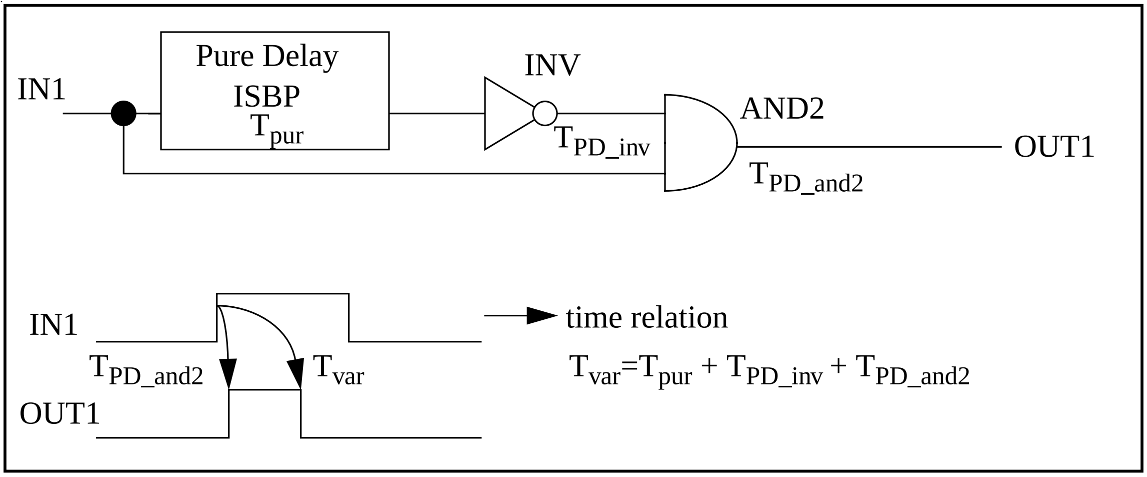


FIGURE 5-14. Behavior Model of Trailing Edge Delay Primitive

(Figure 5-15) and Open Collector Buffer(Figure 5-16). The VHDL behavior model of these ISBPs is given in Appendix C.2.

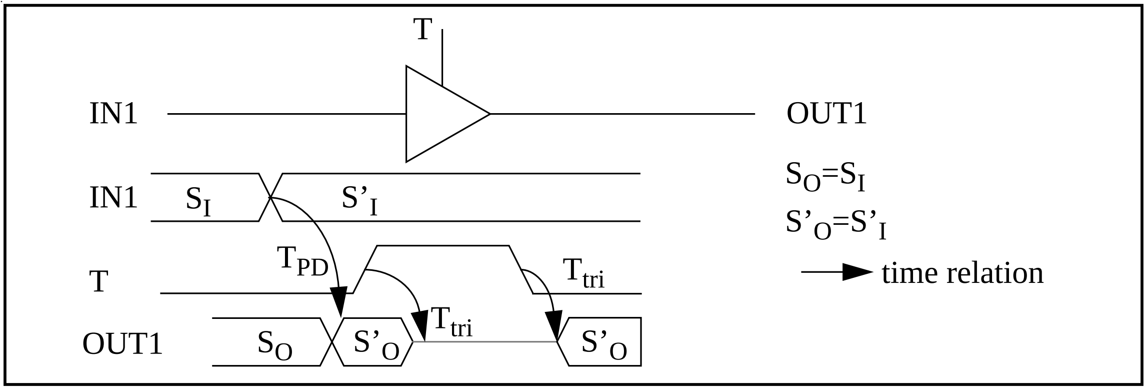


FIGURE 5-15. Behavior Model of Tri-State Buffer Primitive

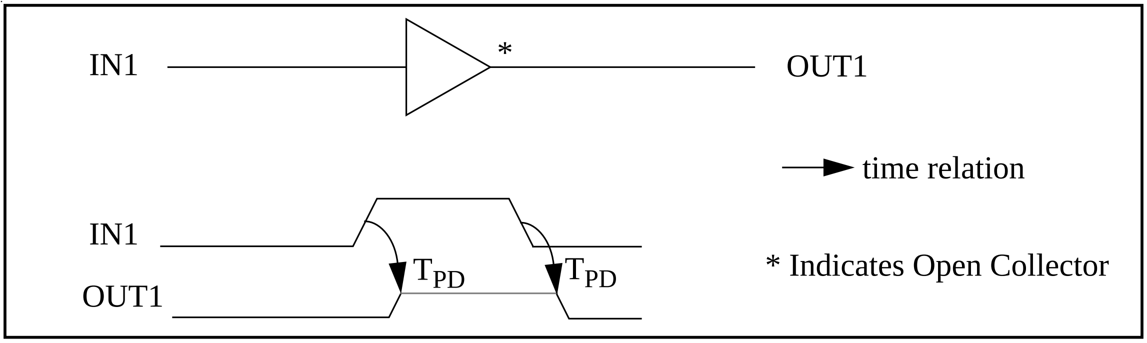


FIGURE 5-16. Logical Model of Open Collector Buffer Primitive

5.4.1.4 ISBP Timing Simulation

A simulation test bench for the VHDL ISBPs was developed to verify their operation. The VHDL test bench was compiled and simulated using Mentor Graphics Corporation VHDL tools. The result of the simulation is shown in Figure 5-17. ‘`in1`’, ‘`in2,`

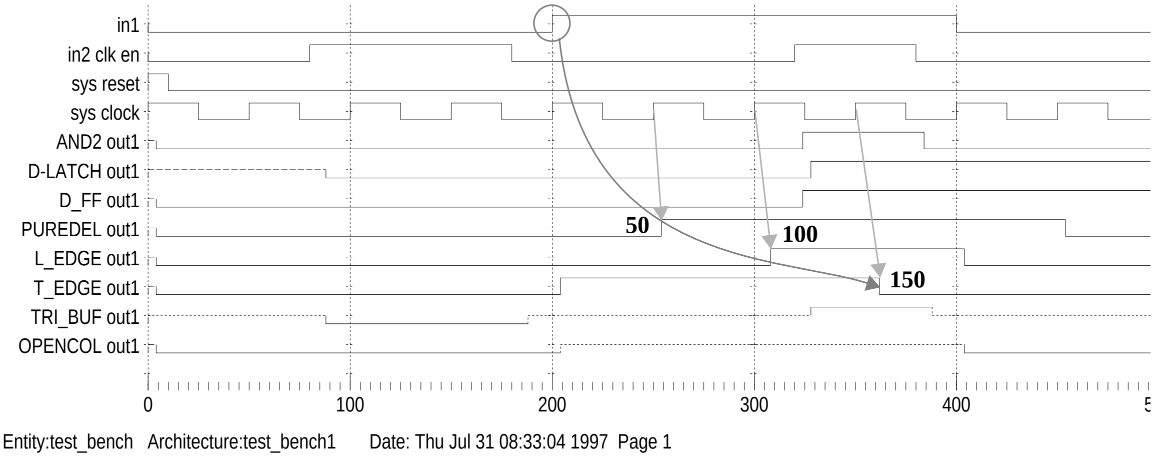


FIGURE 5-17. Simulation of Primitives

clk, en' and 'sys-reset' are three signals generated within the simulation test bench. 'in1' connects to the IN1 input of a device, 'in2, clk, en' connects either to the IN2 input, CLK clock input or T tri-state enable input of a device, while 'sys-reset' connects to the reset input of a device. The out1 output is shown for the 2 input AND gate (AND2), D-Latch (D-LATCH), D-Flip-Flop (D-FF), Pure Delay (PURE-DEL), Leading Edge Delay (L_EDGE), Trailing Edge Delay (T-EDGE), Tri-state Buffer (TRI_BUF) and the Open Collector Buffer (OPENCOL). The test bench used a gate propagation delay tpd of 4ns, a clock to output delay tclk of 5 ns and a tri-state enable t_tri delay of 8 ns. The Pure Delay has a 50 ns propagation delay, while the Leading Edge and the Trailing Edge Delays have a propagation delay of 100 ns and 150 ns respectively. The Pure, Leading and Trailing Edge Delays are implemented using D-Flip-Flops clocked with the 'sys-clock' signal.

The timing diagram verifies the correctness of the VHDL specification of the ISBPs and the operation of the ISBPs. For example, the timing diagram shows the delay of the in1, 0 to 1 edge by approximately 50 ns, 100 ns and 150 ns for the Pure, Leading and Trailing Edge Delays respectively. As expected, the delays are not exactly 50, 100 and 150 ns due to the inherent delays of the ISBPs.

5.5 Interface Representation Summary

This chapter discussed how the interface is represented as a hierarchial data structure that will be created and built up during the interface design process. The interface for a capability is represented using an Interface Block. The IB is sub-divided into Info ISBs. These in turn are sub-divided into Timing and State ISBs. The timing and State ISBs are given using ISBPs. A set of ISBPs was developed to allow for the conversion of commonly found timing and state information formats. The ISBPs include a Combinatorial ISBP, a Tri-state Buffer and Open Collector Buffer ISBP for state information conversion and a D-Latch, D-Flip-Flop, Pure Delay, Leading Edge Delay and Trailing Edge Delay ISBP for timing information conversion. Each ISBP has a VHDL representation which allows testing and implementation of the interface using various technologies. Details of the frame representation of the interface can be found in Appendix B.2.

Chapter 6

The Interface Design Process

6.1 Introduction

This chapter introduces the interface design methodology and its representation in the Interface Designer. Automation of the interface design process is complicated by the fact that often no formal method or precise algorithm exists for all aspects of this design process. A human designer will use heuristics to complete the design. In this work, this heuristic knowledge about interface design is represented in the form of design rules that utilize knowledge at specific abstraction levels of a component's capability. This is one of the primary reasons for the emphasis placed on abstraction in the development of the component and interface models presented in Chapter 4 and Chapter 5.

Figure 6-1 gives an overview of the underlying fundamentals of the design process. A component (Component1) has output information signals with a given state-timing output specification SigSpec1 as specified by the component manufacturer. An output specification gives the state and timing specification of an output signal. Another component (Component2) receives some input information signals with a given state-timing input specification SigSpec2 as specified by the component manufacturer. A state-timing input specification gives the state and timing requirements of an input signal.

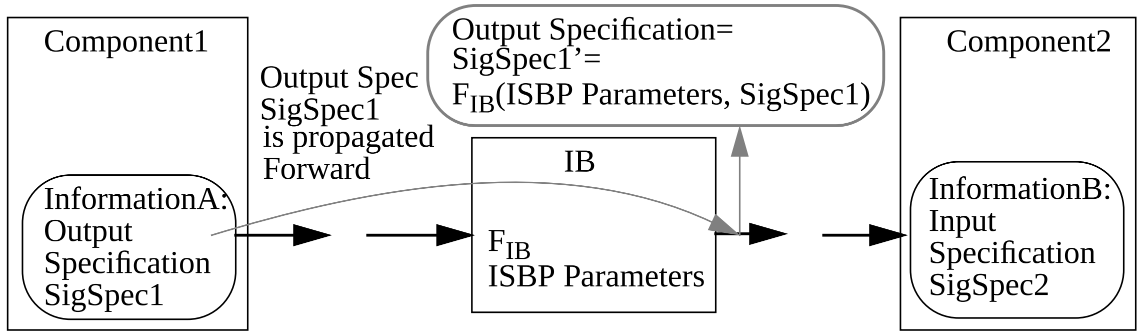


FIGURE 6-1. Interface Design Process

The design of the interface blocks follows a top down design practice. This means that the problem of the interface design is broken down into the design of interconnected sub-systems, which in turn are broken down into the design of more detailed sub-systems until finally at the most detailed level simple, well defined ISBPs are chosen to build the interface. These ISBPs will have ISBP parameters associated with them, such as propagation delays and setup and hold times, whose values will not be known until after the interface design has been completed and an implementation technology has been chosen. This is one of the problems that makes interface design difficult: the design must proceed without knowing the values of the ISBP parameters.

The design of the interface block has one fundamental goal: It must allow the two interfaced components to operate within the limits of the specification supplied by the component manufacturers. As shown in Figure 6-1, the design process developed in this work accomplishes this goal by propagating the output specification of Component1, SigSpec1, forward through the Interface Block (IB), resulting in an output specification SigSpec1'. The output specification of the IB, SigSpec1', is derived as a function F_{IB}, of the ISBP parameters and the output specification of Component1, SigSpec1. Once the design is complete, an implementation technology is chosen with known ISBP parameters, and the timing parameters of SigSpec1' are evaluated. The implementation technology also could be chosen before starting the interface design, and the SigSpec1' timing parameters can be evaluated as SigSpec1 is propagated forward through the IB. The Interface Designer uses the first approach, where the implementation technology is chosen after design completion, to give the user the option of evaluating different implementation technologies for a given design. In any case, once the timing parameters of SigSpec1' are known, they are verified against the SigSpec2 input specification, assuring that the fundamental goal is satisfied.

This chapter first presents the hierarchial tasks used to perform interface design that satisfy the above fundamental goal. This is followed by an overview of the terminology used for the building blocks and signals within the Interface Designer. The rules developed to accomplish each of the Interface Design tasks are then presented in separate sections.

6.2 Abstraction of the Interface Design Tasks

The interface design is divided into several hierarchial tasks which will be accomplished using rules in a production system. The design tasks are organized into the seven layers shown in Figure 6-2. The top level Component Selection design task (Level A) selects components and decides which components must be connected by an interface. The second level Capability Interface design task (Level B) creates an interface block for each capability that must be connected between components. Next, the interface block connecting a capability is broken into a set of Information Connection ISBs (Level C).

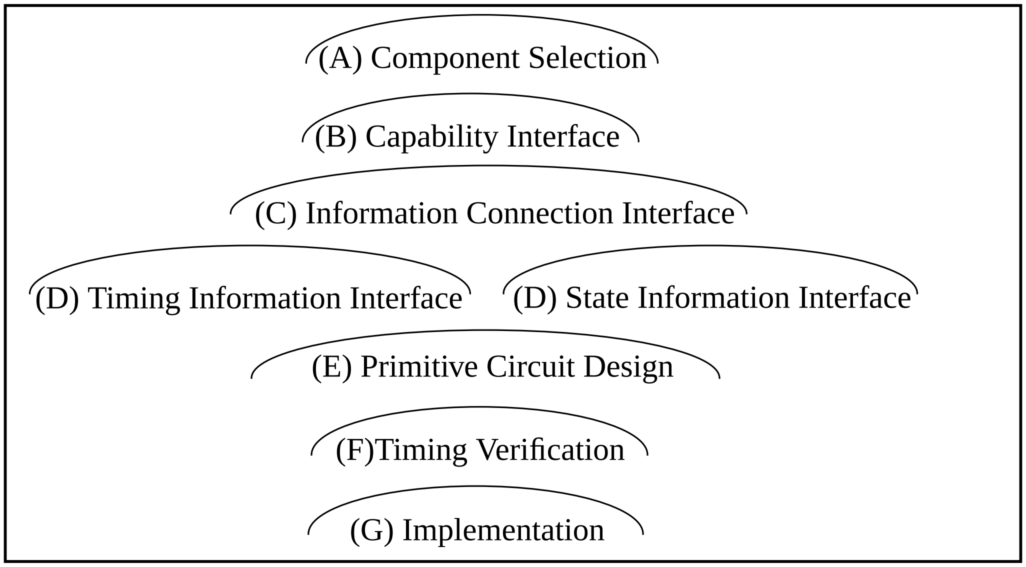

(A) Component Selection
(B) Capability Interface
(C) Information Connection Interface
(D) Timing Information Interface
(D) State Information Interface
(E) Primitive Circuit Design
(F)Timing Verification
(G) Implementation

FIGURE 6-2. Interface Design Task Abstraction Levels

The fourth level of the design hierarchy divides the Information Connection ISB into State and Timing ISBs (Level D). The next task fills the State and Timing ISBs with ISBPs (Level E). After an implementation technology is chosen, timing verification is performed (Level F). Finally the interface is implemented using a VHDL description which allows the design to be simulated or realized using real components (Level G). If problems are found during the timing verification or the implementation phase of the interface design, most likely the components are incompatible and the design process should be repeated using a different selection of components.

This work places emphasis on the development of design rules that can accomplish the design tasks by assembling and connecting building blocks applicable to a particular abstraction level. By using design rules specific to a design abstraction level, task interdependence is reduced. This in turn reduces the number of rules required to achieve each task and it simplifies the development and maintenance of the rules. For example, a set of rules was created to accomplish the division of a generic Information Connection ISB into State and Timing ISBs (Level D). These rules are independent of the actual classification of the information being transferred, found in Level C (i.e. they are independent of the information transfer such as `address` information or `direction` information), and they are independent of the actual timing templates of the signal timing found in Level E (i.e. they are independent of the timing behavior such as multiplexed Latch timing or non-multiplexed Strobe timing). The number of rules is reduced since it is not necessary to provide an Information Connection ISB division rule for each class of information transfer, or for each timing template. If a new class of information transfer has to be added to the capability, the rules dividing an information transfer into state and timing parts do not have to be modified since they are independent of the information class. Thus the maintenance of the rules is also simplified.

This work is primarily concerned with the design of the interface between components (levels B to G), not the selection of the components (level A). Design rules are developed to accomplish the interface design which follow the design task abstraction levels found in Figure 6-2.

6.3 Overview of the Interface Block Design Terminology and Process

Figure 6-3 shows a typical IB. The IB is sub-divided into Information Connection ISBs, also called Info ISBs. The Info ISBs are used to connect information transfer ports between components. Each information transfer port will consist of one or more signals. The *component output signals* (A) will be connected to *IB input signals* (B). The IB input signals are connected to *Info ISB input signals* (C). Each Info ISB may have one or more input signal. In Figure 6-3, ISB1a and ISB1b have one input signal, ISB2 and ISB4 have two input signals, while ISB3 has three input signals. Each Info ISB input signal is connected to the input signal of a Timing ISB. The output signal of the Timing ISB is called the *intermediate signal* (I) and it is connected to the input of the State ISB. The output of the State ISB is connected to the *Info ISB output signal* (D). The Info ISB output signal can be connected to the input of any external component as an *IB output signal* (F), or it can be connected to other Info ISBs as an *internal signal* (E). An internal signal is an Info ISB output signal that only connects to other Info ISB inputs. The only signals connected to the inputs of an Info ISB (C) are IB input signals (B) or internal signals (E). Each signal in Figure 6-3 has an associated timing: The component output timing (A), IB input timing (B), Info ISB input timing (C), Info ISB output timing (D), internal timing (E), IB output timing (F), component input timing (G) and intermediate timing (I).

The interface design process builds up the IB from building blocks specific to each design task abstraction level. Design of the interface commences at the highest abstraction level with the creation of an IB. The IB is created from an IB prototype without specifying internal details, but with all IB input signals (B) and IB output signals (F) specified.

At the next task abstraction level, the IB is filled in with Info ISBs interconnected with Info ISB input and output signals. When each Info ISB is first created, a goal information specification is established for the ISB output. The *goal information* is the desired output state-timing specification of the Info ISB, and the Interface Designer will design

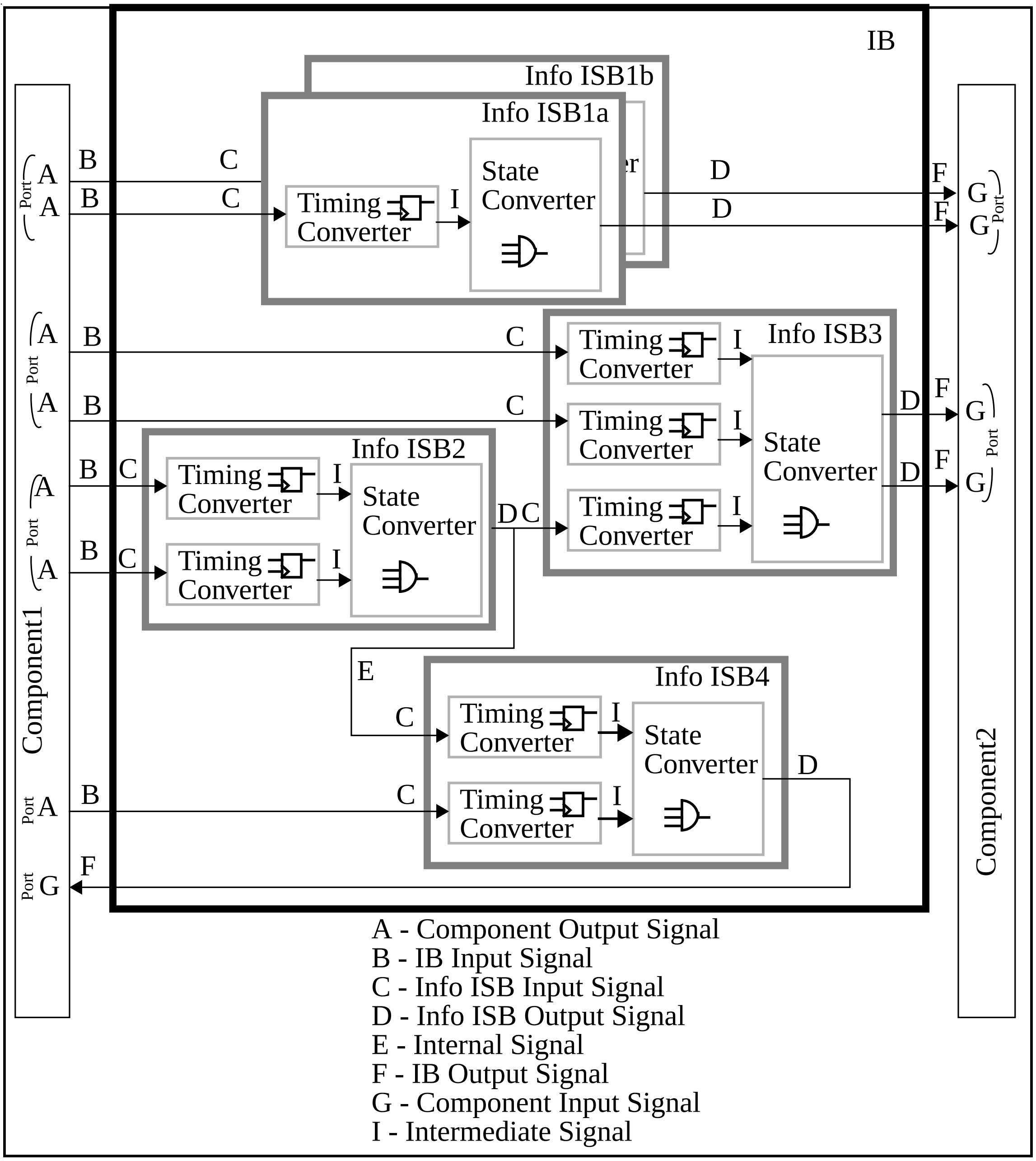


FIGURE 6-3. Design Process Overview and Terminology

the Info ISB so that the output specification of the Info ISB matches the goal information. The design methodology to determine the goal information will be discussed in Section 6.5.5.

The Info ISBs are built up from State and Timing ISB building blocks. Finally the Timing and State ISBs are built up using ISBPs as building blocks. The ISBPs are chosen in a way to generate the desired goal information on the Info ISB output. This is accomplished by first filling in the State ISB with a Combinatorial ISBP, and then filling in each Timing ISB with a Timing ISBP.

An Info ISB can only be designed if its input state and timing specification are known. This means that any Info ISBs whose inputs come from component outputs are designed first, since the signal timings of any component output signals are known from the component library (ISB1a,b and ISB2 in Figure 6-3 are designed first). As an Info ISB is designed, its specific output timing parameters, such as the setup and hold times, will be determined. Any Info ISB that uses the known outputs can then be designed. In the example in Figure 6-3, both Info ISB3 and ISB4 can be designed after ISB2 has been completed since the internal signal (E) comes from the output of ISB2. Since the design of an Info ISB involves the generation of its output signal timing, the output timings of the IB will be known after the completion of all Info ISBs. Once an implementation technology is chosen the IB output timing can then be checked to see whether it satisfies the component input timing, thus verifying that a correct interface has indeed been produced.

The following sections present the rules developed to accomplish each of the Interface Design tasks.

6.4 Creating the Interface Block

The Interface Designer is invoked by a connection request to design a required interface. A connection request specifies the components selected, the class of capability the connection must satisfy and specific information related to the capability.

The data transfer interface connection request contains a list of the components that must be connected through an IB, the address map assigning a unique address to each of the components being connected, the direction of the data transfer, type of data to be transferred and information about the data bus size used for the data transfer. Other information that may be included in the connection request is an indication on what the design priority is: Should the Interface Designer emphasize high speed, low cost, small PC board real estate or low power consumption? The connection request must provide all the information necessary to complete the interface design.

This work is not concerned with how the connection request is generated. It is assumed that the connection request is provided either directly by the user of the design system or through a higher level expert system similar to the one used in the MAPLE [77], MICON [10] and KDMS [45] microprocessor system synthesis systems.

An IB for a capability is created by a rule that is triggered by a connection request frame as shown in Figure 6-4. The signals in and out of the IB are all the signals involved in the capability X.

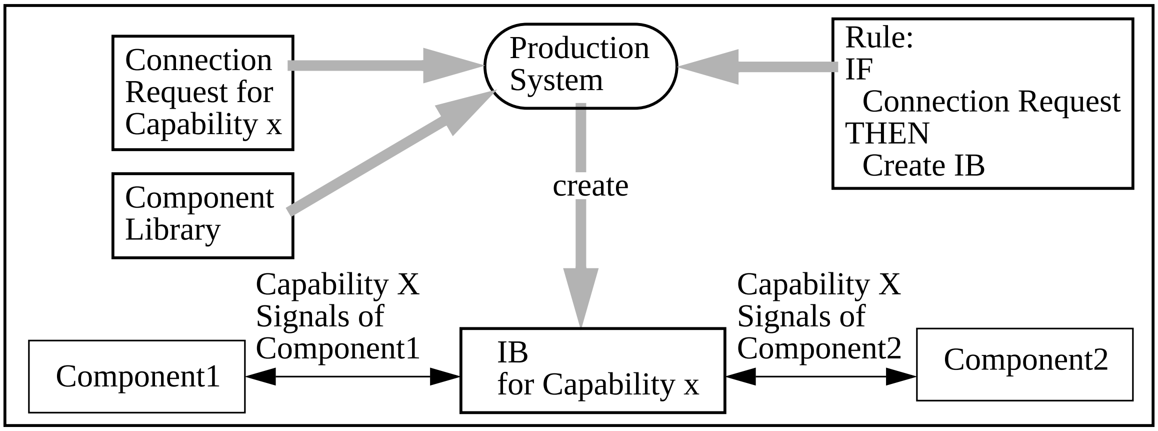


FIGURE 6-4. Capability Connection IB Creation

6.5 Partitioning the IB into Info ISBs

To partition the IB, the protocol of the capability is treated as a series of information transfers which are connected using Info ISBs. Each information transfer is associated with a specific function in the protocol of the capability. For data transfer the protocol requires that *address*, *data*, *type*, *size*, *direction*, *request*, *delay* and *width* information transfer signals be considered for connection using Info ISBs. The decision on which information transfer ports will be connected depends on which information transfer ports exist and the capability being connected. Rules are used to recognize the presence or absence of information transfer ports on a component. Figure 6-5 shows typical Info ISBs created by these rules when they are triggered (in Figure 6-5 an Info ISB is called an ISB).

The primary knowledge for this design task are rules that consider how the different information transfers in a capability should be connected. Figure 6-5 shows a typical example set of information transfers involved in a microprocessor to memory data transfer interface. It should be noted that some of the information of a certain classification can be found on both components being connected (for example *address*, *direction*, *request* and *data* information in Figure 6-5), while others are only found on one component (for example *type* and *delay* information in Figure 6-5).

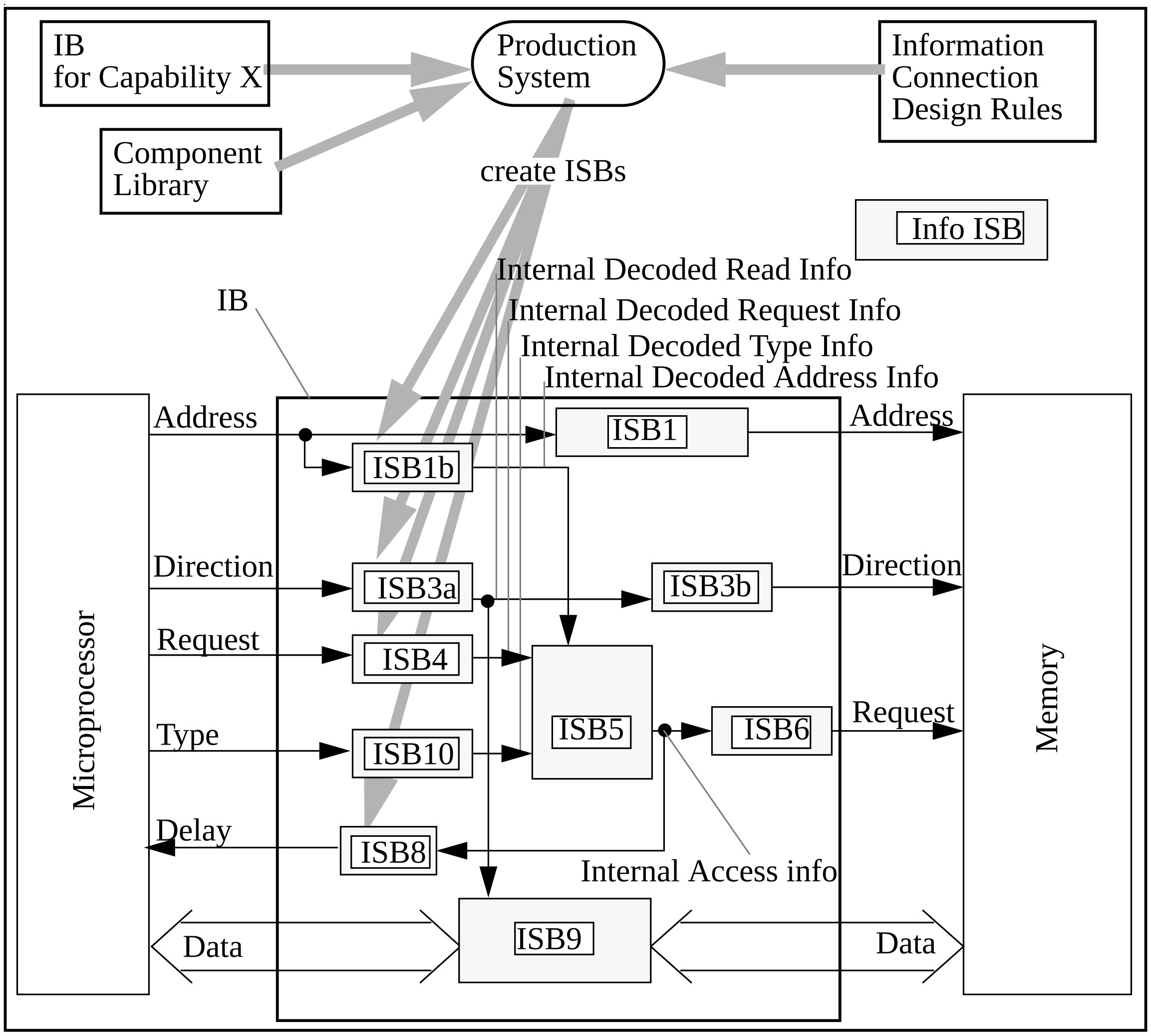


FIGURE 6-5. Example Microprocessor / Memory Interface Info ISBs

The organization of the interface block shown in Figure 6-5 is somewhat dependent on the design style of the designer. Different designers may produce equally correct designs with a different layout of Info ISBs.

Figure 6-5 also shows the generation of internal information ports. An *Internal information port* refers to any port generated within the IB and only connects to the inputs of other Info ISBs within the IB, such as the internal `decoded request` information. Internal information ports are introduced since they provide a flexibility method of representing the knowledge and heuristics a designer uses for interface design. Internal information serves two purposes, which may overlap. First, an internal information port can be used to generate information internal to the interface that is not available directly from the components, such as the internal `access` information in Figure 6-5. Second, an internal information port can be used to provide information signals commonly used in a standard format. For example a designer will normally use an address decoder to generate a decoded address signal. The decoded address signal is then used in conjunction with other signals (such as a data strobe) to generate the chip enable signal for a memory component or to generate the enable signal on a bus transceiver. The internal `decoded address` information generated in ISB1b of Figure 6-5 is equivalent to the decoded address signal. This example also shows that internal information ports provide a powerful tool to separate the utilization of information from the generation of information.

A set of rules is required to generate the Info ISBs in the context of the capability being connected. These rules must be aware of all the possible information input and output ports for a capability and they must include knowledge about how to determine the goal information. These rules are divided into the following categories:

1. Knowledge on how to connect information ports of the same class.
2. Knowledge on how to generate internal information ports.
3. Knowledge on how to use the *extra information* provided by an output port of a component if there is no matching input port on the other component.
4. Knowledge on how to generate the *missing information* required by the input port of a component if there is no corresponding output port on the other component.
5. Knowledge on how to generate the goal information of an Info ISB.

The rules required to represent each of the five categories are discussed in the following sections.

6.5.1 Rules Used for Connecting Information Signals of the Same Class

If information ports with the same class exist between two components, they should be connected. Rules are used to recognize the presence of the same class of information on

If Master has Info	**and Slave has Info**	**Then generate ISB to connect Info**
`data` (In/Out)	`data` (In/Out)	`data` to `data`
`address` (Out)	`address` (In)	`address` to `address`
`direction` (Out)	`direction` (In)	`direction` to `direction`
`type` (Out)	`type` (In)	`type` to `type`
`size` (Out)	`size` (In)	`size` to `size`
`request` (Out)	`request` (In)	`request` to `request`
`width` (In)	`width` (Out)	`width` to `width`
`delay` (In)	`delay` (Out)	`delay` to `delay`

TABLE 6-1. Connections Rules for the Same Information Class

the components that are being connected. These rules, if triggered, will create an Info ISB for each common information class found. For example, in Figure 6-5 ISB1 connects the `address` information ports between Component1 and Component2. Table 6-1 lists all the connection rules developed to connect data transfer ports of the same class.

6.5.2 Rules for Generating Internal Information Ports

The design of the Info ISBs is modularized and simplified through the use of internal information ports with known classification, states and timing. The standardized internal information signals can be used by the Interface Designer during numerous parts of the IB design.

The utility of internal information ports can be illustrated with the simple example design shown in Figure 6-6. The purpose of this Info ISB is to activate the `CE` signal on the memory whenever the address signals have a certain state and whenever either of the `UDS` or `LDS` signals are asserted. By using internal signals the process of generating the `CE` signal can be broken into three simple independent tasks: The generation an internal *`decoded address`* information signal (`SELECT`), the generation of an internal *`decoded request`* signal and the generation of the `CE` signal from the internal *`decoded address`* and *`decoded request`* information signals. The AND function in ISB3 assures that the `CE` signal will only be activated when the address is in the correct range (`A12-A15` are asserted) and when one or more of the *`request`* information signals from the microprocessor is asserted. From another viewpoint, ISB3 combines the internal *`decoded address`* information with the internal *`decoded request`* infor-

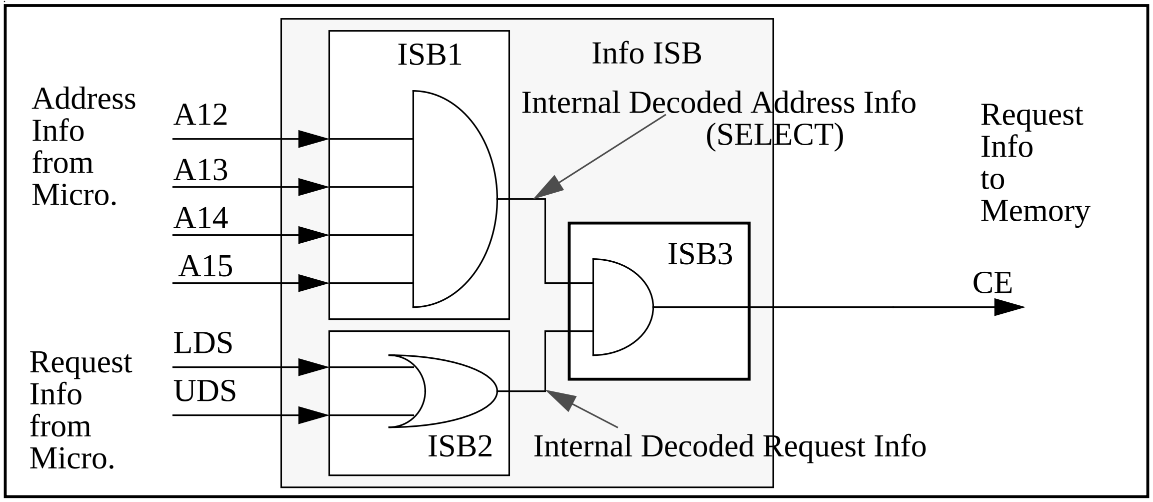


FIGURE 6-6. Example Extra Address Information Merge using three ISBs

mation. Since the combination process involves only standard internal signals generated by the designer, a ‘standardized’ method can be used to generate the CE signal, namely the combination of the internal *decoded address* and *decoded request* signal using the AND gate in ISB3. If the address information from the microprocessor is different (i.e. different address or more address signals), only ISB1 must be changed, and the same ISB3 can be used for the combination process.

By using internal information ports two important advantages are realized. First, rule development becomes simpler since it is usually easier to develop many simple rules that carry out small, well defined independent tasks, than it is to develop one complex rule that carries out a more elaborate task. Second, the rule tasks can be partitioned into groups of independent sub-tasks, which makes the rule base easier to maintain and debug. For example one group of rules generates the internal signals and another group utilizes the internal signals.

Several microprocessor system designs were investigated to see which internal ports are commonly generated, and how these internal ports are utilized. The internal information ports represented in the Interface Designer are:

Internal *decoded request* information	Single signal that specifies events that initiate and terminate data transfer
Internal *decoded address* information	Information indicating when a given memory block in the address space is being accessed.
Internal *decoded type* information	Information indicating when a given memory block in the type space is being accessed.
Internal *decoded size* information	Information indicating what specific data bus signals are used for data transfer.
Internal *decoded read* information	Information indicating when a read data transfer is in progress.
Internal *decoded* access information	Information indicating when a data transfer is in progress in a given address range, type space and for a specific data bus size.

Table 6-2 shows internal information generation rules. These rules were developed to be able to generate all commonly used internal information found in a data transfer interface. As can be seen in Table 6-2, there are two categories of internal information generation. One is the internal *decoded request*, *decoded read* and *access* information which are always generated. The other category has internal information ports generated only if specific extra information is present on a component: *decoded address*, *type* and *size* information. The next section describes how the Interface Designer utilizes extra information.

If	Then Generate Internal
extra `address` information on master	`decoded address` information (A)
extra `type` information on master	`decoded type` information (B)
extra `size` information on master	`decoded size` information (C)
Always	`decoded request` information (D)
Always	`access` information AND of (A) (B) (C) (D) if present
Always	`decoded read` information

TABLE 6-2. Internal Information Generation Rules

6.5.3 Rules Used for Utilizing Extra Information

Often an output port of a certain classification on one component does not have a similar information input port on the other component. This output information port is called an *extra information port*. For example, the microprocessor often provides a `type` information output port, while the memory does not provide `type` information input port.

Design rules are provided that recognize the extra information ports (signals) and subsequently utilize the information if it is required for the correct operation of the interface. For example, in the interface block of Figure 6-5, the extra `type`, `address` and `request` information are used to generate internal `decoded type`, `address` and `request` information which are then applied to ISB5. ISB5 takes all three internal information ports and generates the internal `access` information, which is then sent to the memory `request` information input port after passing through ISB6. ISB6 is generated to allow state and timing conversion of the internal `access` information as required by the `request` information input of the memory.

If Extra Information	Then
`address` on master	utilize the extra `address` information
`direction` on master	error in component library
`type` on master	utilize the extra `type` information
`size` on master	utilize the extra `size` information
`request` on master	error in component library
`data` on master or slave	error in component library
`width` on slave	incompatible component
`delay` on slave	incompatible component

TABLE 6-3. Extra Information Manipulation Rules

Table 6-3 lists all the possible extra information that could be found on either component during data transfer interface design. Since `request` and `direction` information must always be found on both a master and a slave component, extra `request` and `direction` information on the master (but not the slave) indicates that there is an error in the component library and the Interface Designer will stop the design process with an error message. If a slave component has `width` or `delay` information output, but the master component does not have a matching input, it indicates that the components chosen are not compatible with each other. The Interface Designer will not be able to proceed with the design and will stop after printing an error message indicating the components are incompatible.

In this work, the utilization of the extra `address`, `type` or `size` information is accomplished through the use of internal information ports. The Interface Designer will first generate internal decoded information port signals for the extra information as explained in the previous section and then take the decoded extra information which are now in a standard format and logically combine them with the `decoded request` information to generate the internal `access` information.

Extra information is thus handled using two different methods. For extra `request, direction, width, data` and `delay` information an error in the component library or in the selection of components is indicated, while for extra `address, type` and `size` information internal decoded information ports are generated.

6.5.4 Rules Used for Generating Missing Information

The components being connected often do not have an information output port for every information input port. These output information ports are called *missing information ports*. For example, a microprocessor often has a delay information input port, while the memory does not provide a delay information output port.

Design rules are provided to recognize the missing information ports and attempt to utilize available information to generate the missing information. For example, in the interface block of Figure 6-5 the `delay` information output is missing from the memory. The `delay` information is generated from the internal `access` information signal by passing it through ISB8. It will not always be possible to generate the missing information. For example, a microprocessor may have less address signals than are required by a memory component. There is no obvious method of generating the missing address signals directly from the available information. In fact, if `address` or `type` information are found to be missing during interface design, it most likely indicates that incompatible components were chosen during the higher level system design phases. In these cases, the Interface Designer will stop the design process. If the Interface Designer finds missing

If Missing Information Output Port for Matching Information Input Port	Then
`data` master or slave	flag error in component library
`address` on master	flag invalid component selection
`type` on master	flag invalid component selection
`size` on master	generate ISB that supplies `size` information
`request` on master	flag error in component library
`direction` on master	flag error in component library
`width` on slave	generate ISB that supplies `width` information
`delay` on slave	generate `delay` information from `access` information

TABLE 6-4. Missing Information Generation Rules

`data`, `request` or `direction` information on the master, it indicates that an error in the component library, since these information ports must always be present on the master.

Table 6-4 summarizes the rules developed to generate Info ISBs for missing information ports. The Interface Designer will generate missing `size`, `width` and `delay` information ports. The other missing information will generate error messages.

6.5.5 Generating the Goal Information of an Info ISB

When an Info ISB is created, a goal information is determined for its output. The goal information represents the Interface Designer’s understanding of what the output of the Info ISB should be once design has been completed. As such, the Interface Designer will use the goal information as a guideline for designing the Info ISB. The goal information is divided into two parts: the goal state and the goal timing. The goal state of the Info ISB is the state the output of the Info ISB must attain during information transfer.

The goal timing is a timing template and represents the abstract timing behavior of the output of an ISB. This fundamental technique allows rules to be developed that represent the heuristic knowledge a designer uses when designing the interface. This heuristic knowledge allows the design of an ISB to proceed without knowing specific timing parameters of a timing and relying only on the general behavior of the signals (i.e. the timing template of the signals). This is critically important since the timing parameters are a function of the ISBP parameters which are not known until the design is complete and has been implemented in a chosen technology.

Two methods are used by the Interface Designer to determine the goal information. The first method is used when the goal information is for an Info ISB that generates internal information. This method uses simple rules that represent the heuristics a designer would use to generate the internal information signals and are shown in Table 6-5. The

If ISB Generates Internal	Then	Goal State	Goal Timing
`decoded address` information		asserted	Strobe timing
`decoded size` information		asserted	Strobe timing
`decoded type` information		asserted	Strobe timing
`decoded request` information		asserted	Logic timing
`decoded direction` information		asserted	Strobe timing
`access` information		asserted	Logic timing

TABLE 6-5. Internal Information ISB Goal Information

goal states and timings in Table 6-5 were obtained by analyzing many different microprocessor system designs and looking for similarities between the behavior of internal information signals of the same classification. For example, it was found that most microprocessor systems have an internal `decoded address` information signal that is asserted during data transfer and which has a non-multiplexed timing. Therefore, for the internal `decoded address` information, the goal state is chosen as asserted and the goal timing is chosen as a Strobe timing.

The second method is used when the goal information is for an Info ISB output that connects to a component information input port. This method analyzes the information input specification obtained from the component library. The goal timing is determined by finding an output timing template that is compatible with the component input timing template. An output timing template is compatible with an input timing template if the timing parameter range of the output timing falls within the timing parameter range of the input timing template. For example, the setup time parameter range of a Logic timing is (-omp +omp) which falls within the setup time parameter range of the Strobe timing (-~ +omp). This means that the Logic output timing can be a goal timing for a Strobe input timing. There may be more than one output timing template that can satisfy the timing parameter range of an input timing template.

The goal timings for all the possible component input timing templates were determined by considering all the output timing templates whose timing parameter range could satisfy the input parameter range. (The possible input and output timing templates were discussed in Chapter 4). Those output timing templates that have timing parameters that fall outside the input timing parameter ranges are eliminated. For example, Figure 6-7 shows how the setup time parameter range violation is used to eliminate the Follows timing as a goal timing for the Strobe input timing. As shown, the Logic timing and Strobe output timing setup time parameter range satisfies the Strobe input timing setup time

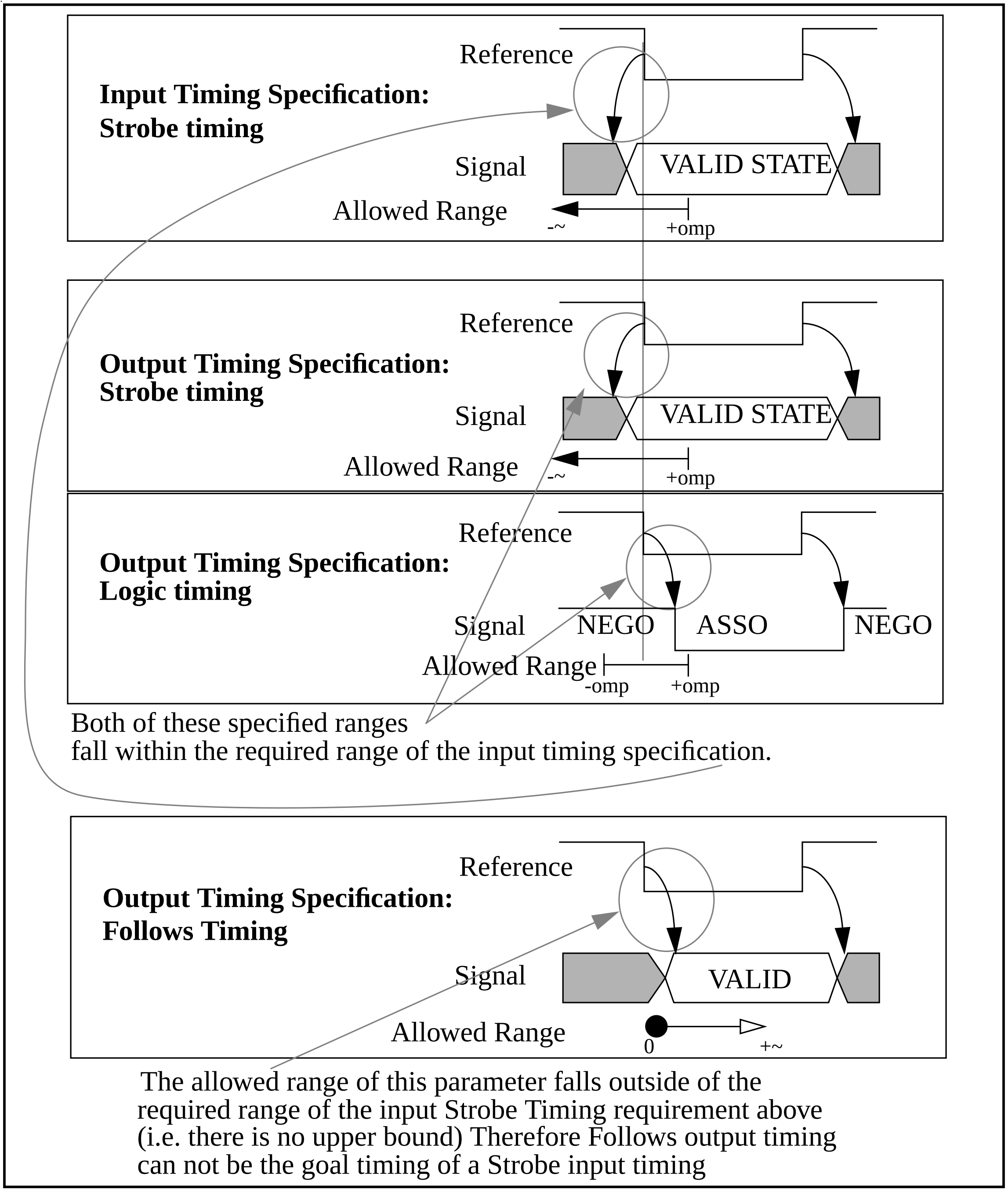

Reference
Input Timing Specification:
Strobe timing
Signal
VALID STATE
Allowed Range
-~
+omp
Reference
Output Timing Specification:
Strobe timing
Signal
VALID STATE
Allowed Range
-~
+omp
Reference
Output Timing Specification:
Logic timing
Signal
NEGO
ASSO
NEGO
Allowed Range
-omp
+omp
Both of these specified ranges
fall within the required range of the input timing specification.
Reference
Output Timing Specification:
Follows Timing
Signal
VALID
Allowed Range
0
+~
The allowed range of this parameter falls outside of the
required range of the input Strobe Timing requirement above
(i.e. there is no upper bound) Therefore Follows output timing
can not be the goal timing of a Strobe input timing

FIGURE 6-7. Strobe Input Timing Specification Goal Timings

parameter range, while the Follows output timing setup time parameter range falls outside that of the Strobe input timing parameter range. A similar conclusion can be drawn for the hold time parameter range for the timings in the example, resulting in the conclusion that Strobe output timing and Logic output timing are both goal timings for an Strobe input timing, while the Follows output timing is not.

A list of possible goal timings for each input timing template is shown in Table 6-6.

Component Input Timing Template	**Goal Timing for Component Input Timing Template**
Strobe	*Strobe*, Logic
Pulse	*Strobe*, Logic
Latch	*Latch*
Pulse-Latch	*Follows*, Strobe, Logic
Follows-Latch	*Follows*, Strobe, Logic
Handshake	*Handshake*
Wait	*Wait*

TABLE 6-6. Goal Timings

Table 6-6 shows that some input timing templates map onto more than one goal timing. The set of timings that can satisfy a component input timing template are called *input specification compatible timings*. From Table 6-6, two sets of input specification compatible timings can be seen: Strobe and Logic timing, and Follows, Strobe and Logic timing. The timings in italics are assigned as the goal timing when an Info ISB is created. During interface design, the Info ISB output timing can be changed to any timing in a set of input specification compatible timings, if necessary. For example, if the input timing template of a component is a Pulse-Latch timing, the set of input compatible timings contains the Follows, Strobe and Logic timing. Table 6-6 shows that the Follows timing is chosen as the goal timing for the Pulse-Latch timing, which means that the Info ISB output timing is set to a Follows timing when the Info ISB is created. During the design of the Info ISB the Interface Designer may realize that the output timing of the Info ISB naturally has a Strobe timing if all Info ISB input timings are Strobe timings. Instead of complicating the interface by adding circuitry that generates a Follows timing output, the Info ISB output timing may be changed to Strobe, since both Follows and Strobe timings are in the same set of input specification compatible timings.

In the timing verification phase of interface design, the ISBP parameters of the interface blocks are assigned values which depend on the implementation technology. A Logic timing will place more restrictions (constraints) on the ISBP parameters than a Strobe timing since the Logic timing has a more restricted range for its timing parameters. A more restricted range on an ISBP parameter means that it will be more difficult to find values that permit correct operation of the IB. In Table 6-6, the timing templates that place the least restrictions on the ISBP parameters are those shown in italics, and are the goal timings chosen for a given component input timing template.

6.6 Creating the State and Timing ISBs

Once an Info ISB is created, a rule is used to partition it into State and Timing ISBs as shown in Figure 6-8. The rule creates the State and Timing ISB building block frames and connects the building blocks with signals. One State ISB is created for each Info ISB, while an individual Timing ISB is created for each signal entering the Info ISB. The output of each Timing ISB is connected to the input of the State ISB.

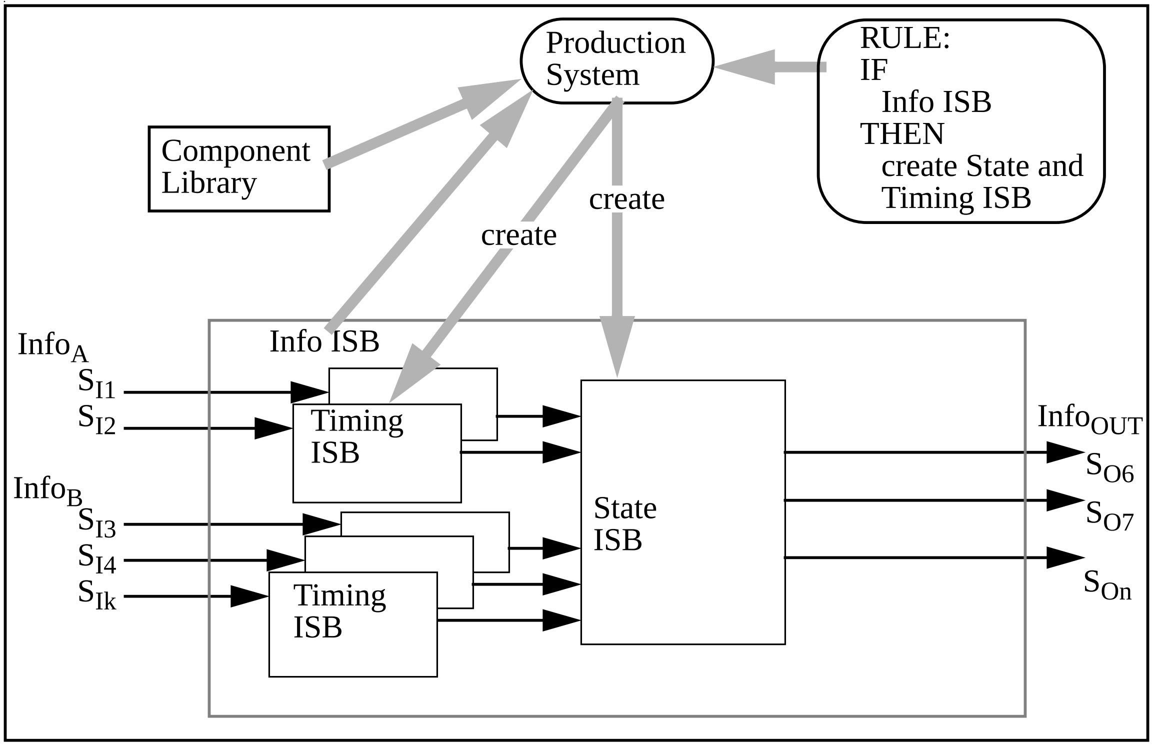


FIGURE 6-8. State and Timing ISB Creation

The Timing ISB will be used to change the timing template of the Info ISB input signal to that of the goal timing of the Info ISB, or to one of the input specification compatible timings, while the State ISB is used to change the state of the Info ISB input signals to the goal state of the Info ISB. The Timing ISB passes the Info ISB input states unchanged, while conceptually the State ISB passes the timing template of the Timing ISB output unchanged. In practice, however, the State ISB will affect the timing in two ways: First, the Combinatorial ISBP within the State ISB will have a non-zero propagation delay, and second, if the State ISB input timing templates are different, the combination of the different timing templates in the State ISB may produce a completely different timing template. Since these aspects of the State ISB deal with timing, a procedure was developed that allows the timing aspects of the State ISB to be dealt with when designing the Timing

ISBs. This allows the State ISB to be designed independently by considering only the state information into and out of the Info ISB. The following section discusses the generation of the ISBP for the State ISB. This is followed by a section discussing the generation of the ISBPs for the Timing ISBs, which also deals with the effect of the State ISB on the Info ISB output timing.

6.7 Generating the Combinatorial ISBP for the State ISB

A Combinatorial ISBP is inserted into the State ISB as shown in Figure 6-9. If states other than asserted or negated must be generated on the output of the Info ISB, a level conversion ISBP such as a Tri-state Buffer or an Open Collector Buffer will also be inserted into the State ISB. The combinatorial logic expression is determined by finding the Boolean logic equation that maps the Info ISB input state to the ISB goal state. For example in Figure 6-6, the internal address decode signal `SELECT`, has the goal state (asso `SELECT`). This state must be attained whenever the Info ISB input state is (AND (asso `A15`) (asso `A14`) (asso `A13`) (asso `A12`)). The Combinatorial ISBP will have the Boolean logic equation: `SELECT` = `A15*A14*A13*A12`, where `*` represents the Boolean AND operation.

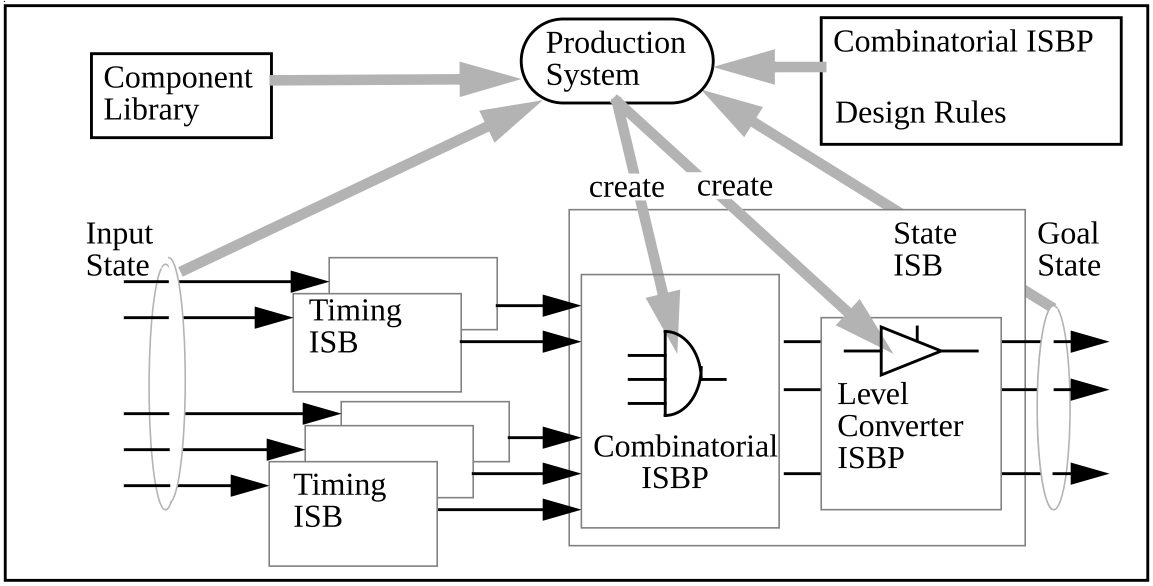


FIGURE 6-9. State ISB Primitive Circuit Creation

It is required that the Timing ISBs must pass the signal states from the Info ISB input unchanged to the State ISB inputs. Care was taken when developing the Timing ISBs to guarantee that this is the case.

6.8 Designing the Timing ISB using ISBP

The design of the Timing ISB involves two parts. First, an ISBP called a *Timing ISBP* is chosen as the basic building block of the Timing ISB. The Timing ISBP is chosen to assure that the desired output timing template (the goal timing) is produced on the Info ISB output. Second, the Info ISB output timing is finalized by considering the effect of the chosen Timing ISBP and of the State ISB on the Info ISB input timing. Figure 6-10 illustrates the action of the rules developed for Timing ISBP design.

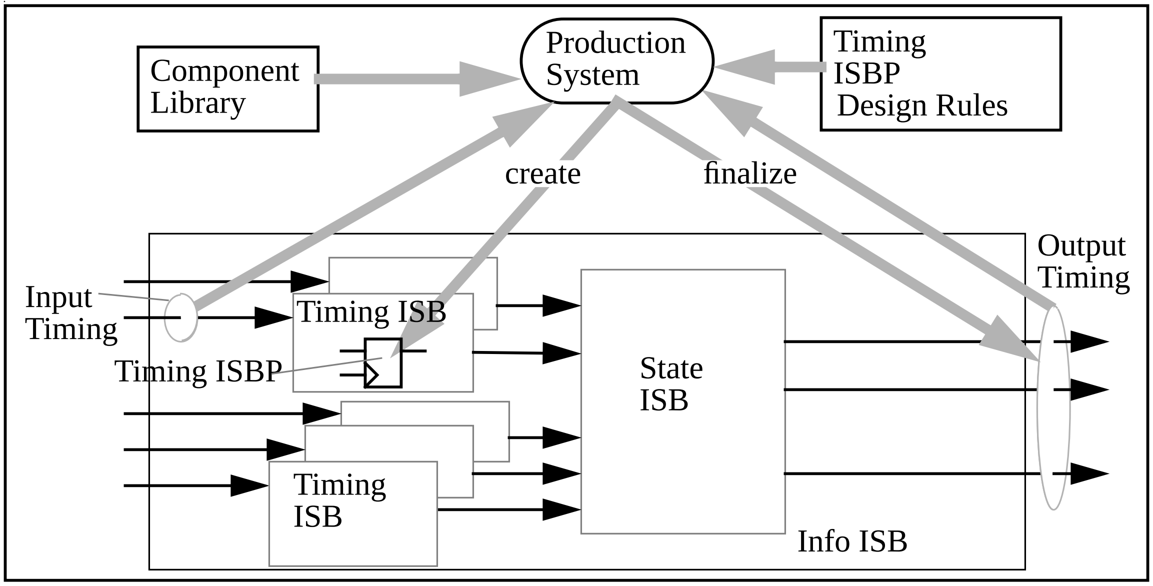


FIGURE 6-10. Timing ISBP Design

The development of the rules that insert the correct ISBP into the Timing ISB was central to this work since they represent an important heuristics a designer uses to complete interface design. The premise seems straight forward: the Timing ISB design rule must insert an ISBP into the Timing ISB that will generate the Info ISB output timing template (which was set to the goal timing), given the input timing template of the Info ISB. In practice, this task is complicated by the fact that after the ISBP has been inserted into the Timing ISB, the Info ISB output timing must be determined as a function of all its ISBP parameters. When the Combinatorial ISBP was chosen, only the input and output states of the Info ISB were considered. Now that the output timing of the Info ISB must be determined, the effect of the Combinatorial ISBP on the input timings must also be considered. The Info ISB output timing is therefore a function of the Timing ISB input timing, the Timing ISBP parameters and also the Combinatorial ISBP parameters and is determined through a process called *timing propagation*.

6.8.1 Overview of the Timing ISB Design Process

Figure 6-11 illustrates the signals and timings of an Info ISB. The output timing template of an Info ISB was determined when it was created: it was set to the goal timing. The signal generated by the Timing ISB is the intermediate signal. The timing of the intermediate signal is determined so that after the intermediate signals pass through the State ISB, the required output timing template or one of the input specification compatible timings is generated.

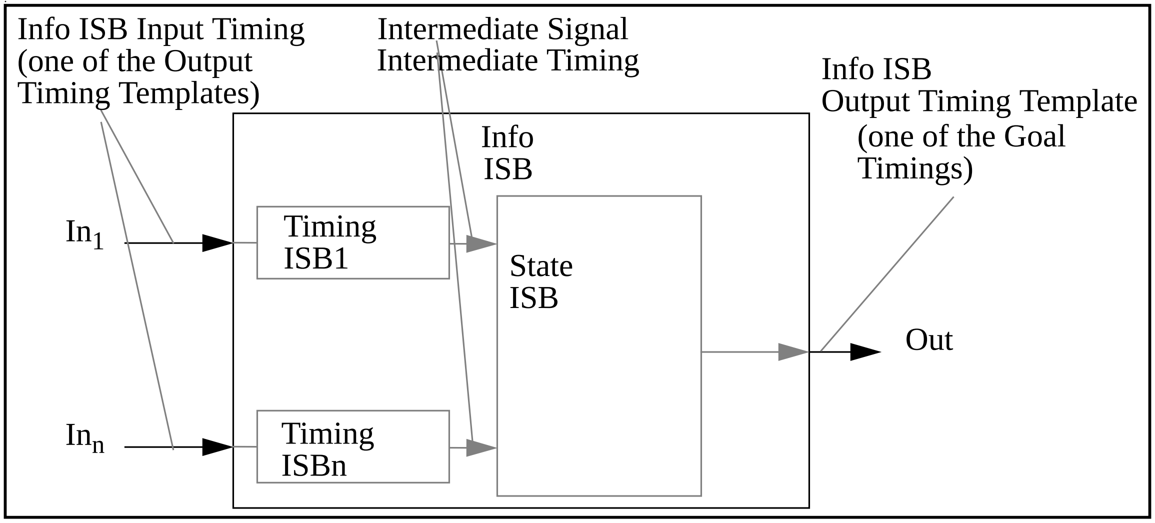


FIGURE 6-11. Info ISB with Timing ISBs

As a simple example of the Timing ISB design process, consider the Info ISB shown in Figure 6-12 which has 2 inputs: a multiplexed input (Latch timing) and a non-multiplexed input (Strobe timing). The output of the ISB must be a signal with the Strobe timing. An experienced designer knows that if each of the intermediate timings is a Strobe timing, then the State ISB output timing and hence the Info ISB output timing will be a Strobe timing. The problem of generating an ISB Strobe output timing is thus changed to the problem of generating intermediate signals with a Strobe timing. $\texttt{Signal}_2$ already has a Strobe timing, and thus can be used directly (Timing ISB2 is just a wire). $\texttt{Signal}_1$ has a Latch timing, which is changed to a Strobe timing in Timing ISB1 using a D-Latch. Both signals going into the Combinatorial ISBP now have a Strobe timing, and as will be shown in Section 6.8.4.1, if all signals going into a Combinatorial ISBP have a Strobe timing, the output will also have a Strobe timing.

The Interface Designer decides which Timing ISBP to use through a set of rules that generates the intermediate timing templates. These rules are based on heuristics a human

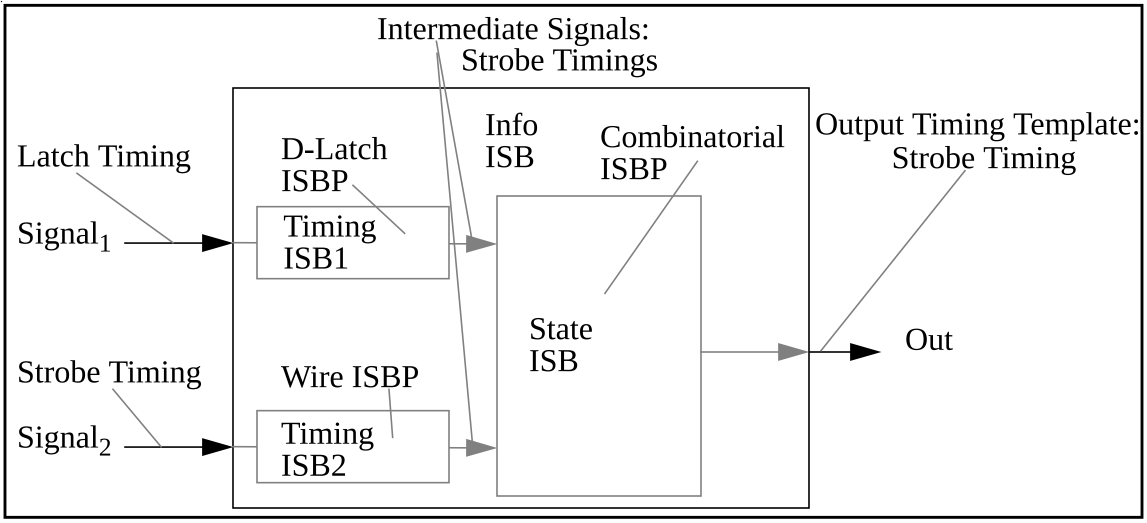


FIGURE 6-12. Interface Sub-Block example

designer would use for each possible Info ISB input timing / Info ISB output timing combination.

Once an ISBP has been chosen and inserted into a Timing ISB, the Info ISB output timing will be finalized. The building blocks of an Info ISB, are parameterized ISBPs such as buffers, combinatorial logic, flip-flops, latches and delays. The Info ISB output timing is represented in terms of the ISBP parameters and the timing parameters of the Info ISB input timings. For example, in Figure 6-13 an Info ISB used for generating the internal *decoded address* information is shown. It consists of two Timing ISBs and one State ISB. The Timing ISBs are made up of Wire ISBPs which pass the Info ISB input signals unchanged to the Combinatorial ISBP with zero propagation delay (T_{pd2}=0). The State ISB is made up of a Combinatorial ISBP with a propagation delay of T_{pd1}. Now assume that the *address* information on the microprocessor has a setup time of -25ns relative to

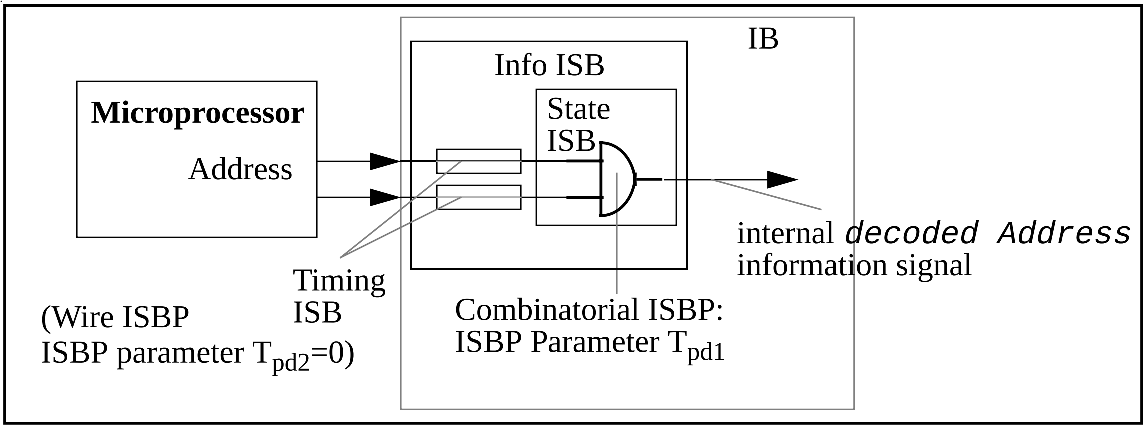


FIGURE 6-13. Example for Info ISB Timing Propagation

the initiate event. The output of the Combinatorial ISBP will have a setup time of $-25+T_{pd2}+T_{pd1} = -25 +T_{pd1}$ relative to this initiate event.

The process of determining the output timing parameter of the *`decoded address`* information signal in terms of the ISBP parameters T_{pd1} and T_{pd2}, and the microprocessor *`address`* information timing parameter is called *timing propagation*: the output timing of an Info ISB is determined by propagating the Info ISB input timing specification forward through the Timing ISBs, and then the State ISB, to the Info ISB output.

The next section provides detailed information on how a Timing ISBP is chosen. This is followed by a section that provides a more detailed insight into the process of timing propagation for the different ISBPs.

6.8.2 Choosing the ISBP to build up the Timing ISB

As shown in Figure 6-11, the input timing of an Info ISB will be based on one of the seven output timing templates, while the output timing of the Info ISB will be based on one of the six goal timings. This means there is a total of 42 possible input/output timing template possibilities for an Info ISB. The Interface Designer must therefore provide rules that can handle all of the Input/Output timing combinations that can occur during interface design.

Table 6-7 lists the input and output template combinations allowed for an Info ISB, marked with a ‘Yes’. Table 6-7 was determined by investigating each of the combinations in the context of microprocessor system design as accomplished by the Interface Designer and asking two questions:

	Info ISB Output Timing Template					
Info ISB Input Timing Template	**Strobe**	**Logic**	**Latch**	**Follows**	**Handshake**	**Wait**
Strobe	Yes	Yes	Yes	Yes		
Logic	Yes	Yes	Yes	Yes	Yes	Yes
Pulse						
Latch	Yes	Yes	Yes	Yes		
Follows				Yes		
Handshake					Yes	Yes
Wait					Yes	Yes

TABLE 6-7. Permitted Input / Output Timing Templates for Info ISB

1. Can the combination possibly occur during the design process used by the Interface Designer?

   and

2. If the combination can occur, is it possible to perform the timing template conversion?

Only if both questions can be answered with a yes, is the combination allowed in Table 6-7.

For an example applicable to the first question, the columns for the Handshake and Wait output timing template are considered. It is found that the Interface Designer will only encounter Logic, Handshake and Wait Info ISB input timings since there are only two possible sources of `delay` information: the `delay` information from the output of another component, which always will have a Wait or Handshake timing or the internal `access` information signal, which will always have a Logic timing.

For an example applicable to the second question the Strobe output timing is considered. The Interface Designer can conceivably generate Strobe, Logic, Latch and Follows input timings. The Follows input timing however must be eliminated, since the setup time event into the ISB for the Follows timing can have a later time of occurrence than that permitted by the Strobe timing as shown in Figure 6-14. This means that it will not be possible to generate a Strobe output timing from a Follows input timing. This combination will normally not occur, and if it does occur the Interface Designer will print an error message

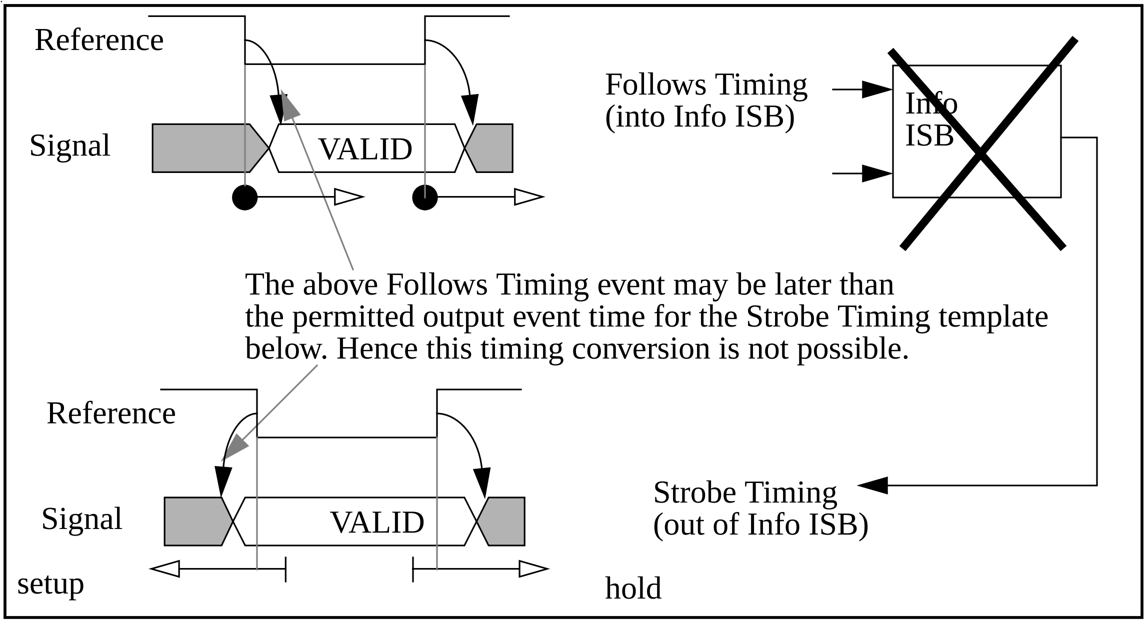

Reference
Signal
VALID
Follows Timing
(into Info ISB)
Info
ISB
The above Follows Timing event may be later than
the permitted output event time for the Strobe Timing template
below. Hence this timing conversion is not possible.
Reference
Signal
VALID
Strobe Timing
(out of Info ISB)
setup
hold

FIGURE 6-14. Follows Input to Strobe Output Timing Template

stating that the design can not be completed and that a different component should be selected.

Table 6-8 shows the intermediate timings for the Info ISB input/output combinations that are permitted. The intermediate timings were chosen to satisfy three criteria:

Info ISB Input Timing Template	**Info ISB Output Timing Template**					
	Strobe	**Logic**	**Latch**	**Follows**	**Handshake**	**Wait**
Strobe	*Strobe*	*Strobe*	Latch	*Strobe*		
Logic	*Logic*	*Logic*	Latch	*Logic*	**Handshake**	**Wait**
Pulse						
Latch	**Strobe**	**Strobe**	*Latch*	**Strobe**		
Follows				*Follows*		
Handshake					*Handshake*	Wait
Wait					Handshake	*Wait*

TABLE 6-8. Intermediate Timing Templates for Input / Output Timings of Info ISBs
Bold entries require a non-trivial Timing ISBPs
Italic entries require a Timing ISBP that consists of a simple Wire ISBP
Regular font entries can be avoided through proper choice of components

1. The intermediate timing must be able to produce the desired output timing after passing through the State ISB.
2. If different intermediate timings are allowed for a given output timing (i.e. the timings found in a column in Table 6-8 are different) then the combination of one or more of those timings must also be able to produce the desired output timing after passing through the State ISB.
3. It must be possible to use one of the Timing ISBPs presented in Section 5.4 to perform the timing conversion to the intermediate timing.

The methodology used to verify that each of the three criteria is satisfied will be discussed in Section 6.8.4, “Combinatorial ISBP Timing Propagation”.

If the Info ISB input timing and the intermediate timing are identical, the Timing ISBP is simply a wire which connects the Info ISB input signal directly to the intermediate signal. In Table 6-8, any intermediate timings that can be generated using a wire are shown in *italics*.

Those intermediate timings not shown in italics require a non-trivial (i.e. non-wire) Timing ISBP to convert from the Info ISB input timing. These Timing ISBPs must be able to convert from Strobe to Latch, Logic to Latch, Logic to Handshake, Logic to Wait, Latch to Strobe, Handshake to Wait and Wait to Handshake timing.

In real world design situations, it was found that a designer will normally avoid component combinations that require generation of multiplexed signal from a non-multiplexed signal (Strobe to Latch) since an overly complex design will result. The Interface Designer is a proof of concept system that represents a design expert’s knowledge of interface design. To limit complexity, no Timing ISBPs are provided for those Info ISB input and output timings that a human designer will normally avoid. These timing conversions include the Strobe to Latch, Logic to Latch, Handshake to Wait and Wait to Handshake. If these timing combinations occur, the Interface Designer will display an error message indicating that the Interface Designer can not complete the design due to a component incompatibility and that different components should be chosen. If timing conversions for these combinations are desired at a later date, they can be added to the Interface Designer by using the appropriate timing conversion rules.

The timing conversion provided in the Interface Designer includes Latch to Strobe (using a D-Latch), Logic to Handshake (using a Leading Edge Delay) and Logic to Wait (using a Trailing Edge Delay). The intermediate timings for which a non-trivial timing conversion is required are shown in **bold** in Table 6-8.

The next sections provide a description of how the signal timings are propagated through Timing and Combinatorial ISBPs. The techniques developed are presented using examples. To illustrate timing propagation through Timing ISBPs, timing propagation examples for a D-Latch ISBP and a Leading Edge Delay ISBP are provided. Two examples are given for the Combinatorial ISBP: one treats the case where all input timings are Strobe timings, the other treats the case where the input timings are Logic and Strobe timings.

6.8.3 Timing ISBP Timing Propagation

Timing propagation for each Timing ISBP must be developed in the context of ISBP input timings and the ISBP parameters. Other than the Wire ISBP, which represents the trivial case that passes any signal timing unchanged from one end of the wire to the other, a Timing ISBP has only one possible input timing template and several ISBP parameters. This is in contrast to the Combinatorial ISBP discussed in the next section which has one ISBP parameter (T_{pd}) and several input timing templates.

This section describes the process developed to propagate the timing on the input of a Timing ISBP to the output. Specifically it shows by two examples how expressions are derived that give the output timing parameters, such as setup and hold times, in terms of the input timing parameters and the Timing ISBP parameters.

6.8.3.1 D-Latch ISBP Timing Propagation

A D-Latch ISBP was developed to generate a Strobe timing signal from a Latch timing signal. In more practical terms, a D-Latch ISBP is used to convert a signal that has a multiplexed timing to a signal that has a non-multiplexed timing. As shown in Figure 6-15 a D-Latch has four ISBP parameters associated with it that can be used to completely describe it. Delay d_D is the delay for any event that occurs on the I_D input and passes through to the O_Q output, provided that the clock is in a state that passes the D input to the Q output (i.e. the latch is in the transparent state). d_{CLK} is the clock event to output change delay. The D-Latch also has data input setup and hold times d_s and d_h associated with it. The behavior of the D-Latch is described in detail in Section 5.4.1.3.

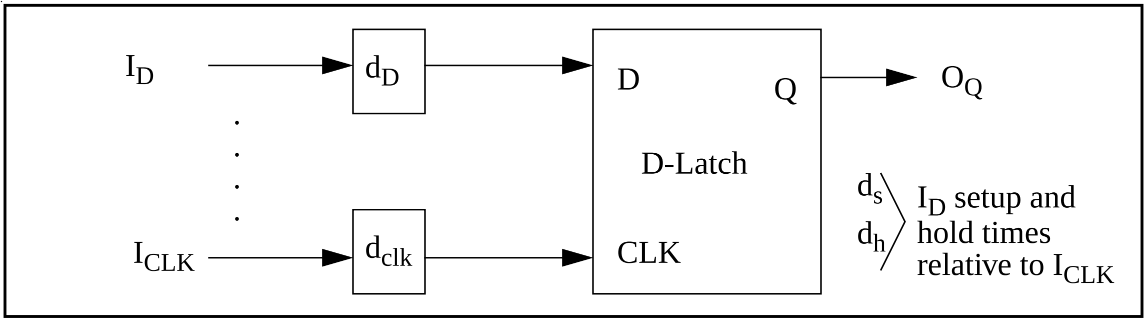


FIGURE 6-15. Model of D-Latch ISBP

Figure 6-16 shows the output Strobe timing of a D-Latch for a Latch timing input. Expressions for the output setup and hold times are desired. The output setup time Ts_O can be obtained by considering what causes the INVALID to VALID output signal transition: it is the INVALID to VALID input signal transition on I_D. The output setup time Ts_O relative to the reference initiate event can therefore be expressed as the sum of the setup time of the input signal I_D relative to the initiate event, $Ts=Ts_{CLK} + Ts_I$, and the signal delay d_D from the D input to the Q output of the D-Latch.

The output hold time Th_O can be obtained by considering what causes the VALID to INVALID output signal transition: It is the NEGATED to ASSERTED input signal transition on CLK. The output hold time Ts_O can therefore be expressed as the sum of the hold time of the clock Th_{CLK}, and the clock input to output delay d_{CLK} of the D-Latch.

$$Ts_O=Ts_{CLK}+Ts_I+d_D \quad \text{(EQ 6-1)}$$

$$Th_O=Th_{CLK}+d_{CLK} \quad \text{(EQ 6-2)}$$

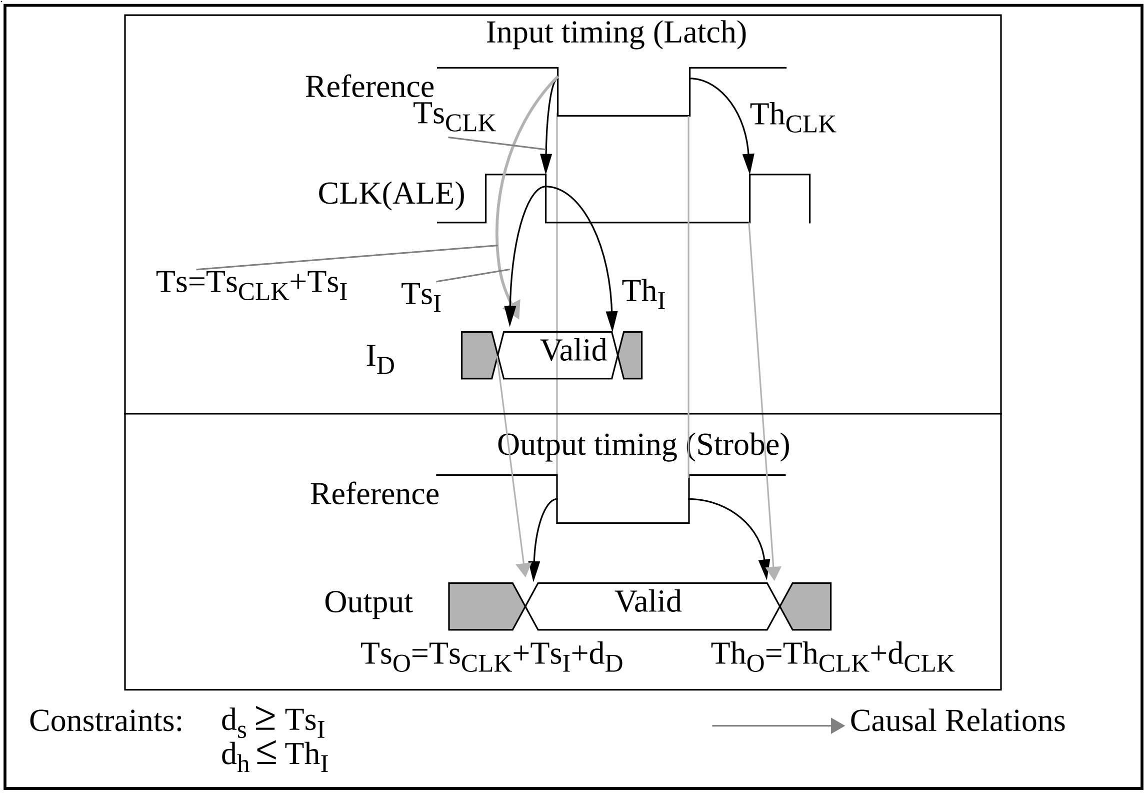


FIGURE 6-16. Timing for Latch Output if Input is Latch Timing

Given any Latch input timing with timing parameters Ts_{CLK}, Ts_I, Th_{CLK}, and a D-Latch ISBP with parameters d_D and d_{CLK}, the timing parameters of the resulting Strobe output timing of the D-Latch ISBP can be calculated using Equation 6-1 and Equation 6-2.

Equation 6-1 and Equation 6-2 are independent of the D-Latch input setup (d_s) and hold time (d_h) parameters. However for the D-Latch to operate correctly, the setup and hold time parameter range of the I_D input signal must satisfy d_s and d_h. To assure that this is the case, two timing constraints are generated. A *timing constraint* is a relationship between ISBP parameters (such as propagation delays) and/or component timing parameters (such as setup and hold times) that must always be satisfied. Throughout this work, whenever a timing constraint is generated, it is assumed that it will be satisfied after the design is complete. Later, during the timing verification phase, values will be assigned to the ISBP parameters and the constraints will be evaluated. However at this point in the design (timing propagation phase) the timing constraints are simply generated.

Two timing constraints are generated for the D-Latch ISBP:

$$d_s \geq Ts_I \qquad \text{(EQ 6-3)}$$

$$d_h \leq Th_I \qquad \text{(EQ 6-4)}$$

Whenever timing propagation for a D-Latch ISBP is performed to generate expressions for the output timing, the two timing constraint expressions shown in Equation 6-3 and Equation 6-4 are also generated.

6.8.3.2 Leading Edge Delay ISBP Timing Propagation

The Leading Edge Delay ISBP was developed to allow conversion of a Logic timing signal to a Handshake timing signal. In more practical terms, the Leading Edge Delay ISBP is used to generate the acknowledge signal for a microprocessor that uses the Handshake timing for the `delay` information. A Leading Edge Delay ISBP has one input and one output. It has the property of delaying the leading edge of the signal by a desired amount, while the trailing edge is delayed by a propagation delay.

Figure 6-17 shows the model for the Leading Edge Delay ISBP developed for this work, in terms of a leading edge ISBP parameter d_{var} and the trailing edge ISBP parameter d_{prop} which is the propagation delay of a combinatorial AND function. The d_{prop} delay is dependent on the implementation technology, i.e., it is the propagation delay of a 2-input AND gate for a given technology (typically 3-7nsec for standard TTL logic). The d_{var} delay however can be any value (usually $d_{var}{>>}d_{prop}$) and will be implemented using different techniques depending on the desired value of the delay. Small values of d_{var} from 10-1000ns usually will be implemented with delay lines, while delays greater than 100ns are usually implemented using shift registers or counters.

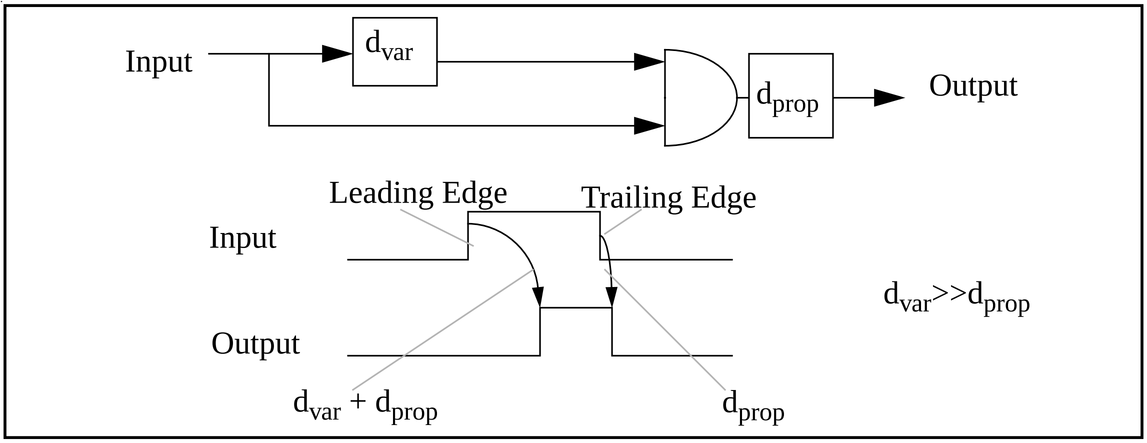


FIGURE 6-17. Model of Leading-Edge Delay ISBP

Figure 6-18 shows the Logic input timing parameters Ts_I and Th_I relative to the reference events. The model used for this work shown in Figure 6-17 shows that the leading

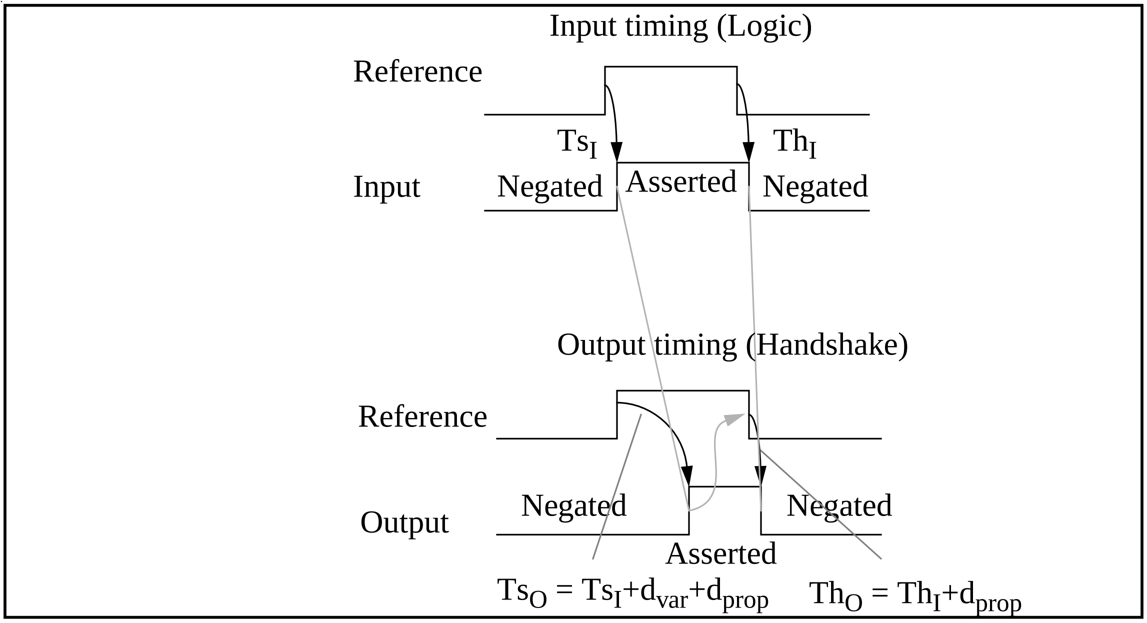


FIGURE 6-18. Logic input and Handshake Output Timing

edge will be delayed by d_{var} and d_{prop}, while the trailing edge will be delayed only by d_{prop}. Therefore the output setup and hold times of the Leading Edge Delay ISBP can be expressed as:

$$Ts_O = Ts_I+d_{var}+d_{prop} \tag{EQ 6-5}$$

$$Th_O = Th_I+d_{prop} \tag{EQ 6-6}$$

It should be noted that the primary purpose of the Leading Edge Delay ISBP was to delay the leading edge of a pulse by a certain amount represented by d_{var}. An ‘ideal’ Leading Edge Delay ISBP has an ISBP parameter d_{prop} of zero. However in practice a Leading Edge Delay ISBP with zero d_{prop} can not be built and therefore the propagation delay d_{prop} had to be introduced. For the Leading Edge Delay ISBP, the d_{var} timing parameter will be calculated after the interface has been implemented in a chosen technology, and when all other timing parameters of the interface have been assigned a value. This is explained in more detail in Section 6.9.3.

6.8.3.3 Summary of Timing ISBP Timing Propagation

Using the techniques introduced with the two examples given in this section, the process of timing propagation for a Timing ISBP is specified as a procedure for every input / output combination of a Timing ISBP. Table 6-9 summarizes the procedures as simple steps utilizing the equations developed in this section.

Timing ISBP	Input Timing	Output Timing (Intermediate Timing)	Steps to Propagate Timing Parameters
D-Latch	Latch	Strobe	Propagate input to output (Eq. 6-1 , 6-2) Generate 2 constraints for input setup and hold times (Eq. 6-3 , 6-4)
Leading-Edge Delay	Logic	Handshake	Propagate input to output (Eq. 6-5 , 6-6)
Trailing-Edge Delay	Logic	Wait	Propagate input to output (similar to Eq. 6-5 , 6-6)

TABLE 6-9. Steps for Timing ISBP Timing Propagation

6.8.4 Combinatorial ISBP Timing Propagation

This section describes the process developed to propagate the timing on the input of a Combinatorial ISBP to the output. Specifically it shows by example how expressions are derived that give the output timing parameters, such as setup and hold times, in terms of the Combinatorial ISBP input timing parameters (which in turn are given as ISB input and Timing ISBP parameters) and the Combinatorial ISBP parameter, T_{pd}.

Table 6-10 shows the possible input timings into the Combinatorial ISBP for each output timing. This table can be directly obtained from Table 6-8 since the Combinatorial ISBP output timing template is the same as the Info ISB output timing template. For several output timing templates more than one intermediate timing template can be found.

Possible Input Timings Template (one of intermediate timings)	Combinatorial ISBP Output Timing Template
Strobe, Logic	Strobe
Strobe, Logic	Logic
Latch	Latch
Strobe, Logic, Follows	Follows
Handshake	Handshake
Wait	Wait

TABLE 6-10. Possible Input Timing for each Output Timing Template for Combinatorial ISBP

The Combinatorial ISBP is modeled as a boolean function F($\mathtt{I_1, \ ... \ I_n}$) with a propagation delay T_{pd} on the output as shown in Figure 6-19. An event on an input $\mathtt{I_j}$ will have an effect on the output after a propagation delay T_{pd}. For timing propagation to proceed, expressions must be found that give the timing parameters of timing T_O of the output

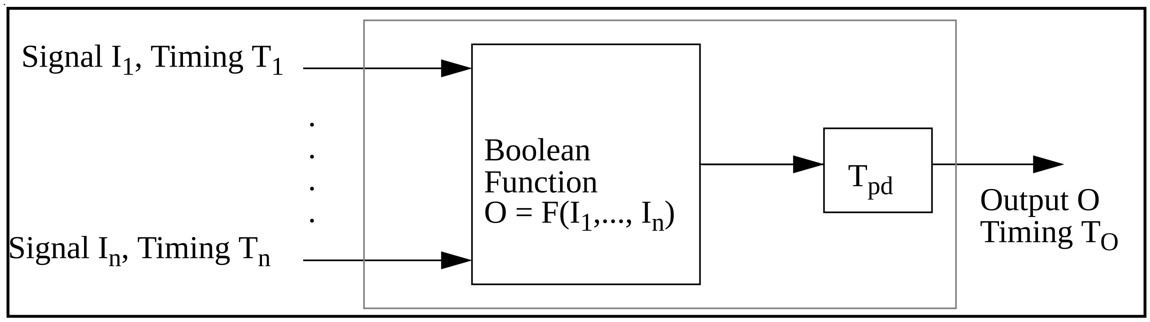

Signal I_1, Timing T_1
Signal I_n, Timing T_n
Boolean Function $O = F(I_1,..., I_n)$
T_{pd}
Output O Timing T_O

FIGURE 6-19. Model of Combinatorial ISBP

O as a function of the input timing parameters and T_{pd}. Note that only the timing templates as shown in Table 6-10 can be input timings to the combinatorial ISBP.

It will be shown that the output timing parameter expressions can be built up by considering each input to the Combinatorial ISBP in any order. This will allow the Interface Designer to derive the expressions incrementally using rules that will systematically process each of the Combinatorial ISBP input signals in an arbitrary order.

The technique used to develop the timing propagation expression is presented using two examples to give the reader an understanding of the issues involved. The first example shows how expressions for the output timing are obtained for Combinatorial ISBP that has all Strobe input timing. The techniques presented for the first example using all Strobe input timings can be applied to Combinatorial ISBPs that have the same timing template on each of the inputs. The second example provides timing propagation expressions for a Combinatorial ISBP that has both Strobe and Logic timing inputs.

6.8.4.1 Example of Strobe Input Timings for Combinatorial ISBP

Strobe timings are associated with VALID/INVALID transitions of signals. It is known that if all inputs of an arbitrary combinatorial function are VALID, then the output will be VALID, otherwise the output will be INVALID. This fact can be used to determine the output timing of a Combinatorial ISBP. When the last input becomes VALID, the output will become valid after T_{pd}, and when the first input becomes invalid the output will become INVALID after T_{pd}.

Figure 6-20 shows the Strobe input timings for inputs I_1 to I_n with their respective setup and hold times Ts and Th. An input that becomes VALID will not cause an INVALID!VALID transition on the output unless all other inputs are VALID. The time of occurrence of the INVALID!VALID transition on the output will therefore be determined by T_{pd} and by the input whose time of occurrence of the INVALID!VALID transition

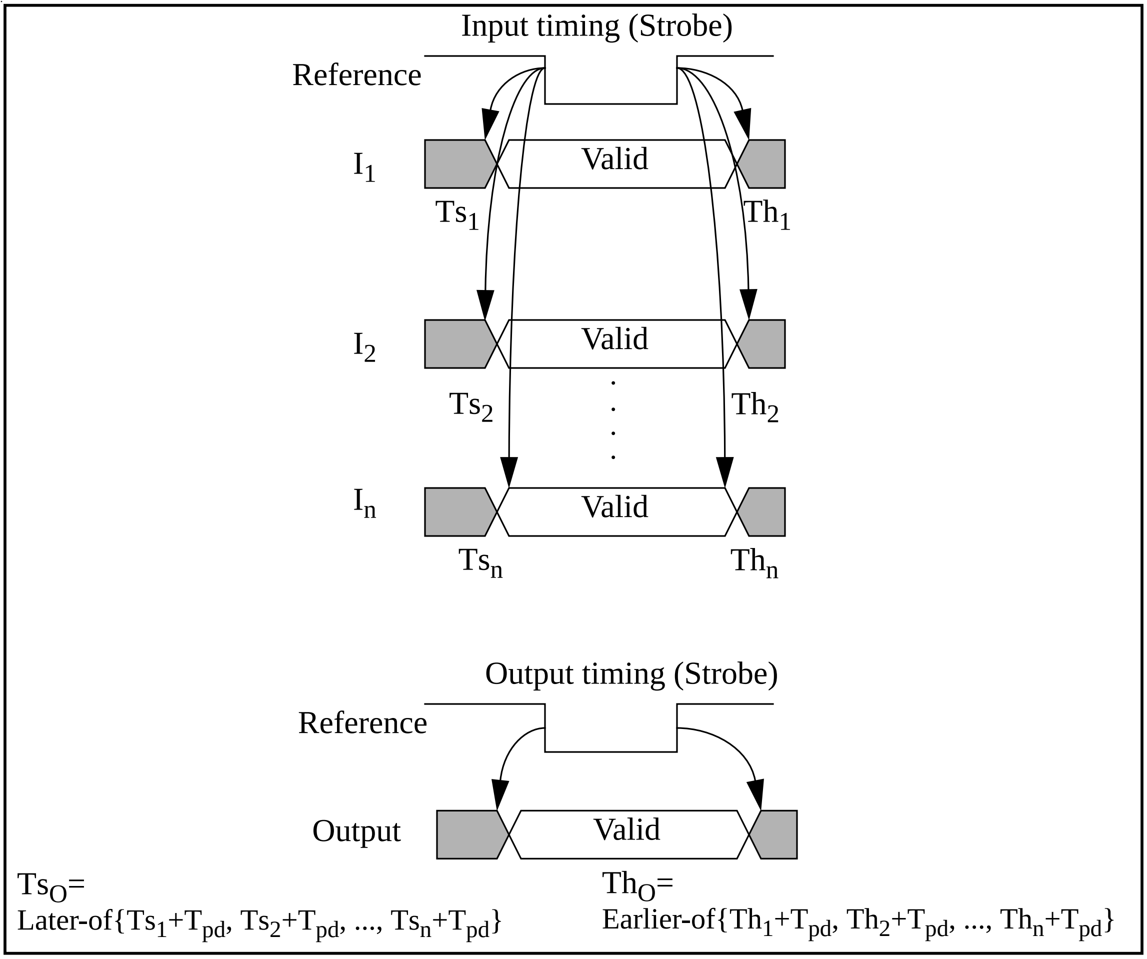


FIGURE 6-20. Timing for Combinatorial ISBP Output for all Strobe Input Timings

occurs latest in time. An event A is later in time than an event B if event A occurs after event B. Since the times as used in the signal timings are always relative to some reference event at time zero, a later event will have a greater signed time value than an earlier event. The setup times of the inputs under discussion will usually be bounded by ‘-~’ on the left side. For example a signal may have a setup time of (-~ -5) relative to some reference event. This setup time means that the address will become valid at time -5ns or earlier (-~ implies ‘or earlier’). A simple operator was developed that selects the time interval that extends the latest in time. This operator is called the *Later-of* operator and it allows the setup time for the output of the Combinatorial ISBP to be written as:

$$Ts_O = \text{Later-of}\{Ts_1+T_{pd}, Ts_2+T_{pd}, ..., Ts_n+T_{pd}\} \qquad \text{(EQ 6-7)}$$

Note that the ‘Later-of’ operator is commutative: Later-of{A, B} = Later-of{B, A}. For example:

$$\text{Later-of}\{(\text{-}\sim -10)\ (\text{-}\sim -5)\} = \text{Later-of}\{(\text{-}\sim -5)\ (\text{-}\sim -10)\} = (\text{-}\sim -5).$$

This example shows the result of the Later-of operator on two time intervals -10ns and earlier, and -5ns and earlier. The resulting time interval is -5ns and earlier (-~ -5), since -5ns is later than -10ns.

The determination of the output hold time of a Combinatorial ISBP with all Strobe input timings proceeds similar to the setup time. The first input to become INVALID will cause an INVALID!VALID transition on the output. The time of occurrence of the VALID!INVALID transition of the output will be determined by T_{pd} and by the input whose time of occurrence of the INVALID!VALID transition occurs earliest in time. An event A is earlier in time than an event B if event A occurs before event B. An earlier event will have a smaller signed time value than a later event. The hold times of the inputs under discussion will usually be bounded by '+~' on the right side. For example a signal may have a setup time of (5 +~) relative to some reference event. This setup time means that the address will become valid at time 5ns or later (+~ implies 'or later'). A simple operator was developed that selects the time interval that extends the earliest in time. This operator is called the *Earlier-of* operator and it allows the hold time for the output of the Combinatorial ISBP to be written as:

$$Th_O = \text{Earlier-of}\{Th_1+T_{pd}, Th_2+T_{pd}, ..., Th_n+T_{pd}\} \qquad \text{(EQ 6-8)}$$

Note that the 'Earlier-of' operator is also commutative: Earlier-of{A, B} = Earlier-of{B, A}.

In general, the setup and hold time ranges for a Strobe timing are (-~ +omp) and (-omp +~) (see Appendix A.1.1 for a description of the Strobe timing). Given that

1. All the inputs to the Combinatorial ISBP are Strobe timings whose setup and hold times fall within (-~ +omp) and (-omp +~).
2. The propagation delay invariance of timing templates (see Section 4.7.3 for a description of propagation delay invariance and the omp delay).
3. The propagation delay T_{PD} is a small delay where omp+T_{PD}=omp, by definition of omp.

It follows that each of the arguments of the Later-of and Earlier-of expressions given in Equation 6-7 and Equation 6-8, and therefore both Ts_O and Th_O, also fall within the setup and hold time ranges of a Strobe timing. A signal timing whose timing parameters satisfy all timing parameters of a timing template can be represented using that timing template. This means that the output timing of a Combinatorial ISBP that has all inputs with a Strobe timing can be represented using a Strobe timing template.

The analysis of the ISBP output timing was independent of the boolean function F of the Combinatorial ISBP, since only VALID/INVALID states have to be considered for the Strobe timing. Since the Earlier-of and Later-of operators are commutative the expressions shown in Equation 6-7 and Equation 6-8 can be generated by adding a propagation delay term to the argument list of the Earlier-of{} and Later-of{} operators as each of the I_1 to I_n input signals is investigated individually. The commutative property means that the final expressions will be independent of the order in which the inputs are considered.

6.8.4.2 Example of Logic Timing Inputs Mixed With Strobe Timing Inputs

The second example of timing propagation for Combinatorial ISBP is for mixed Logic and Strobe input timings. This typically occurs for an address decoder signal with Strobe timing that is gated (ANDed) with a signal that has a Logic timing. Normally when such a circuit is designed, its purpose is to produce an output without glitches or anomalies that has a Logic output timing. This is illustrated in Figure 6-21. Two address signals A14 and A15 (Strobe timing) and a data strobe signal DS (Logic timing) go into a Combinatorial ISBP to produce an output signal O. To assure that the output is glitch free, the output of the Combinatorial ISBP was chosen by the Interface Designer to be a Logic timing (i.e., the goal timing of the Info ISB output is a Logic timing). To assure that the output timing adheres to the timing parameter ranges of the Logic timing, we must place some restrictions on the relationships between the input Strobe and Logic timings:

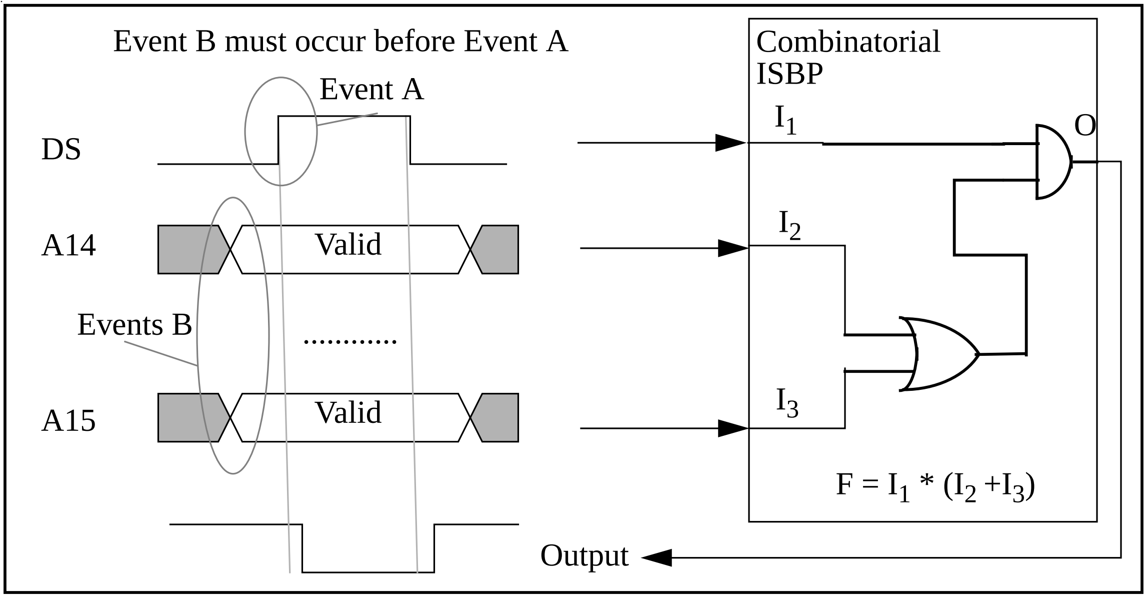


FIGURE 6-21. Overview of Input and Output Timings for Combinatorial ISBP

1. The leading event of the Strobe timing signals (events B in example) must occur before the leading event of the Logic timing signal (event A in example).
2. The trailing edge event of the Strobe timing signals must occur after the trailing event of the Logic timing signals.
3. The boolean logic function must logically AND the asserted level of the signals with a Logic timing to the other signals (For example, in Figure 6-21 the DS signal with Logic timing is ANDed with the address signals A14, A15).

If these restrictions are satisfied, then we can guarantee that the leading and trailing edges of the output will be glitch free as required.

Restriction 3 is normally automatically satisfied through the use of components that are known to work with each other without extraordinary interface circuitry and by employing a design strategy that will logically AND important intermediate information signals (as discussed in Section 6.5). The Interface Designer always checks that signals with Logic input timing are in fact ANDed with the other signals.

Figure 6-22 shows the two timing constraints (restrictions 1 and 2) that are generated for each Strobe timing input signal, one for the setup time and one for the hold time. If there are *k* Strobe timing input signals there will be 2*k* constraints. The constraints on the Strobe input setup and hold times are expressed as a function of the propagation delay T_{PD} and of the output setup and hold times:

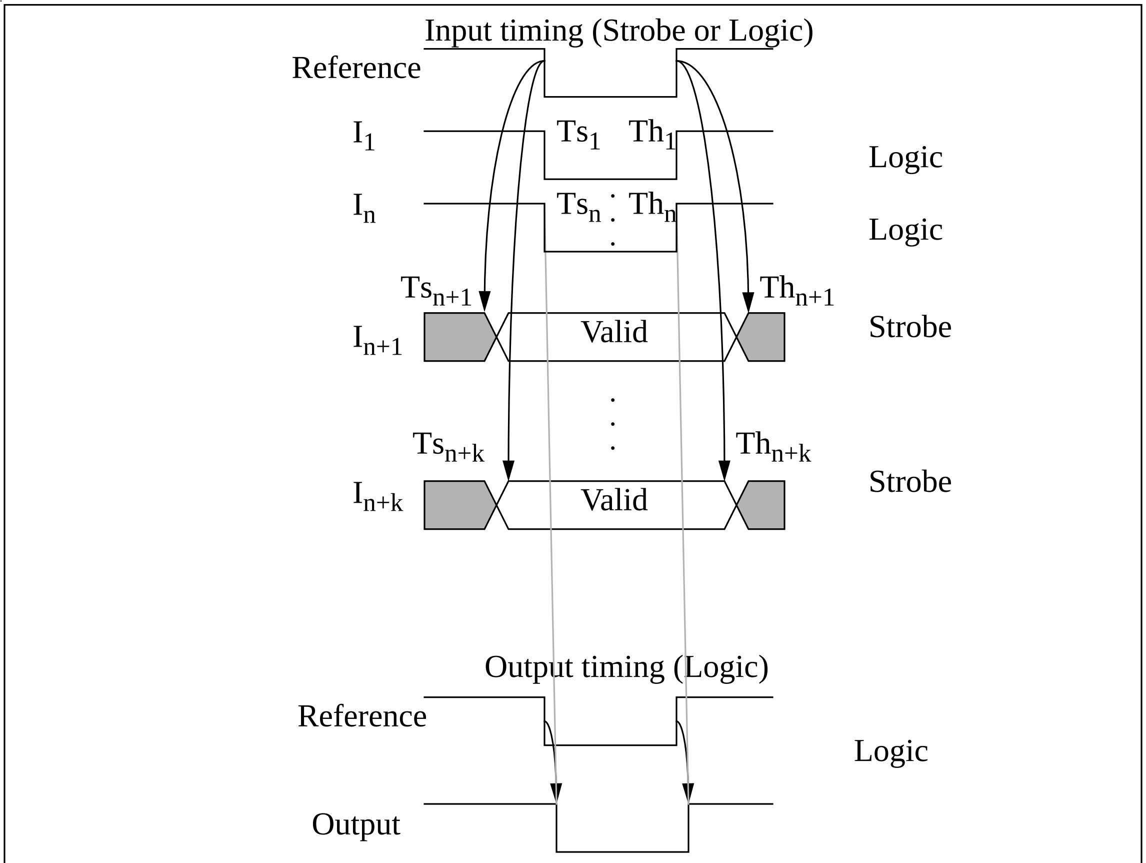
$$Ts_O \geq Ts_i + T_{pd} \qquad \text{(EQ 6-9)}$$

$$Th_O \leq Th_i + T_{pd} \qquad \text{(EQ 6-10)}$$

By expressing the constraints in terms of Ts_O and Th_O the process of obtaining the constraints is simplified. It makes it possible to look at each input signal independently and in any order, and if it has a Strobe timing, simply generate the two constraints shown in Equation 6-9 and Equation 6-10.

The setup and hold times of the output timing can be expressed using Equation 6-7 and Equation 6-8 with the *n* Logic input timing signals as shown in Figure 6-22. The expressions for the setup and hold times of the output timing are independent of the signals that have a Strobe timing, due to the three restrictions above placed on the inputs with Strobe timing.

As an example, assume that there are 3 input signals to an Info ISB as shown in Figure 6-21. I_1 has a Logic timing, while I_2 and I_3 have a Strobe timing. Then Ts_O=Later-of$\{Ts_1+T_{pd}\}=Ts_1+T_{pd}$ and Th_O=Earlier-of$\{Th_1+T_{pd}\}=Th_1+T_{pd}$. Four constraints will be generated: $Ts_O \geq Ts_2+T_{pd}$, $Ts_O \geq Ts_3+T_{pd}$, $Th_O \leq Th_2+T_{pd}$ and $Th_O \leq Th_3+T_{pd}$. Once



Ts_O=Later-of$\{Ts_1+T_{pd}, ..., Ts_n+T_{pd}\}$ Th_O=Earlier-of$\{Th_1+T_{pd}, ..., Th_n+T_{pd}\}$

Timing Constraints:

$$Ts_O \geq Ts_{n+1}+T_{pd} \qquad Th_O \leq Th_{n+1}+T_{pd}$$

$$\vdots \qquad\qquad \vdots$$

$$Ts_O \geq Ts_{n+k}+T_{pd} \qquad Th_O \leq Th_{n+k}+T_{pd}$$

FIGURE 6-22. Timing for Combinatorial Output for Logic and Strobe Input Timings

all inputs have been processed, Ts_O and Th_O can be substituted into the constraints giving $Ts_1 \geq Ts_2$, $Ts_1 \geq Ts_3$, $Th_1 \leq Th_2$ and $Th_1 \leq Th_3$.

6.8.4.3 Summary of Combinatorial ISBP Timing Propagation

Using the techniques introduced with the two examples given in this section, the process of implementing the timing propagation for a Combinatorial ISBP can be specified as a simple procedure consisting of one or more steps for every input / output combination. Care was taken when developing the procedures to assure that the inputs to the Combinatorial ISBP could be considered individually and in any order. Table 6-11 summarizes the steps developed for all combinations of input and output timings. Generally the steps consist of propagation of input timing parameters from the input to the output using the expressions from Equation 6-7 and Equation 6-8, and the extraction of con-

Comb. Input Timing	Comb. Output Timing	Steps to Propagate Timing Parameters
Strobe	Logic	Generate 2 Constraints (Eq. 6-9 , 6-10)
Logic*	Logic	Propagate input to output (Eq. 6-7 , 6-8) (Check for AND combinatorial equation)
Strobe	Strobe	Propagate input to output (Eq. 6-7 , 6-8)
Logic	Strobe	Propagate input to output (Eq. 6-7 , 6-8)
Strobe	Follows	Propagate input to output (Eq. 6-7 , 6-8)
Logic	Follows	Propagate input to output (Eq. 6-7 , 6-8)
Follows	Follows	Propagate input to output (Eq. 6-7 , 6-8)
Latch	Latch	Propagate input to output (Eq. 6-7 , 6-8)
Handshake*	Handshake	Propagate input to output (Eq. 6-7 , 6-8) (Check for AND combinatorial equation)
Wait*	Wait	Propagate input to output (Eq. 6-7 , 6-8) (Check for AND combinatorial equation)

TABLE 6-11. Steps for Combinatorial ISBP Timing Propagation

straints given by Equation 6-9 and Equation 6-10. Some of the timing propagation procedures (marked with *) also include a check to verify that the appropriate signals are ANDed together.

6.9 IB Timing Verification

Once all Timing and State ISBs have been filled in with ISBPs, the structure of the IB is completely defined as illustrated in Figure 6-23. The figure also shows that the signals out of the IB with TimingOUT will be connected to the input signals of Component2

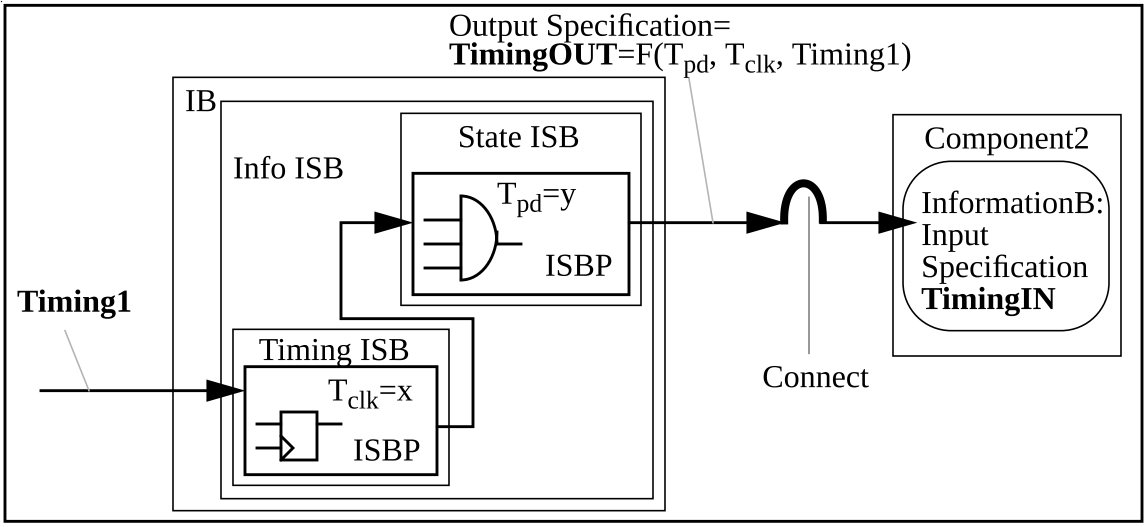


FIGURE 6-23. The Interface Output to Component Connection

with TimingIN. To allow the interface to operate correctly, the timing parameters of TimingOUT must satisfy the timing parameters of TimingIN. For example, if the input signal on component2 requires a certain setup time parameter range, the IB output signal setup time must fall within that range. The output timing TimingOUT is a function of the ISBP parameters which may or may not satisfy the input specification of component2, depending on the implementation. The process of checking whether the IB output timing satisfies the component input timing for a given implementation is called *IB timing verification*.

The verification proceeds in two steps. First a timing constraint, called a *connection timing constraint*, is extracted for every timing parameter of every component input timing. As stated before, a *timing constraint* is a relationship between ISBP parameters (such as propagation delays) and component timing parameters (such as setup and hold times) that must hold for correct operation of the interface. Second, all timing constraints extracted during the interface design must be verified. This includes both the connection timing constraints and the timing constraints extracted during timing propagation as explained in Section 6.8.3 and Section 6.8.4. The timing constraints are verified by choosing an implementation technology, assigning numeric values to the ISBP parameters and then evaluating the constraints.

6.9.1 The Connection Timing Constraint Extraction Process

The timing of the signals in the interface block is given relative to the component that generates the initiate and terminate events for the data transfer (this is usually the bus master for the current data transfer). Figure 6-24 shows a simple example interface for a non-multiplexed address signal `A1` of a microprocessor and the non-multiplexed address signal `A0` of a memory component. Since both address signals have a non-multiplexed signal timing, the Timing ISB for the `A1` signal will simply be a Wire ISBP. Since the Wire ISBP passes the input signal unchanged, this discussion only has to consider the buffer C1.

The timing relationship between signals in the IB is shown in simplified form in Figure 6-25. The microprocessor generates signal `A1` which has timing specification T1, relative to the reference signal `DS`. The output `A1`’ of the buffer has timing specification T2 relative to the reference signal `DS`. The ISBP for the buffer has a propagation delay d1. Thus we can state that the timing T2 of `A1`’ is the same as the timing T1 relative to `DS`, except that the `A1` signal is delayed by d1. The propagation delay d1 depends on the technology that will be used to implement the interface. For example a LSTTL buffer will have a minimum delay of 2ns and a maximum delay of 8ns, written as the interval (2ns 8ns).

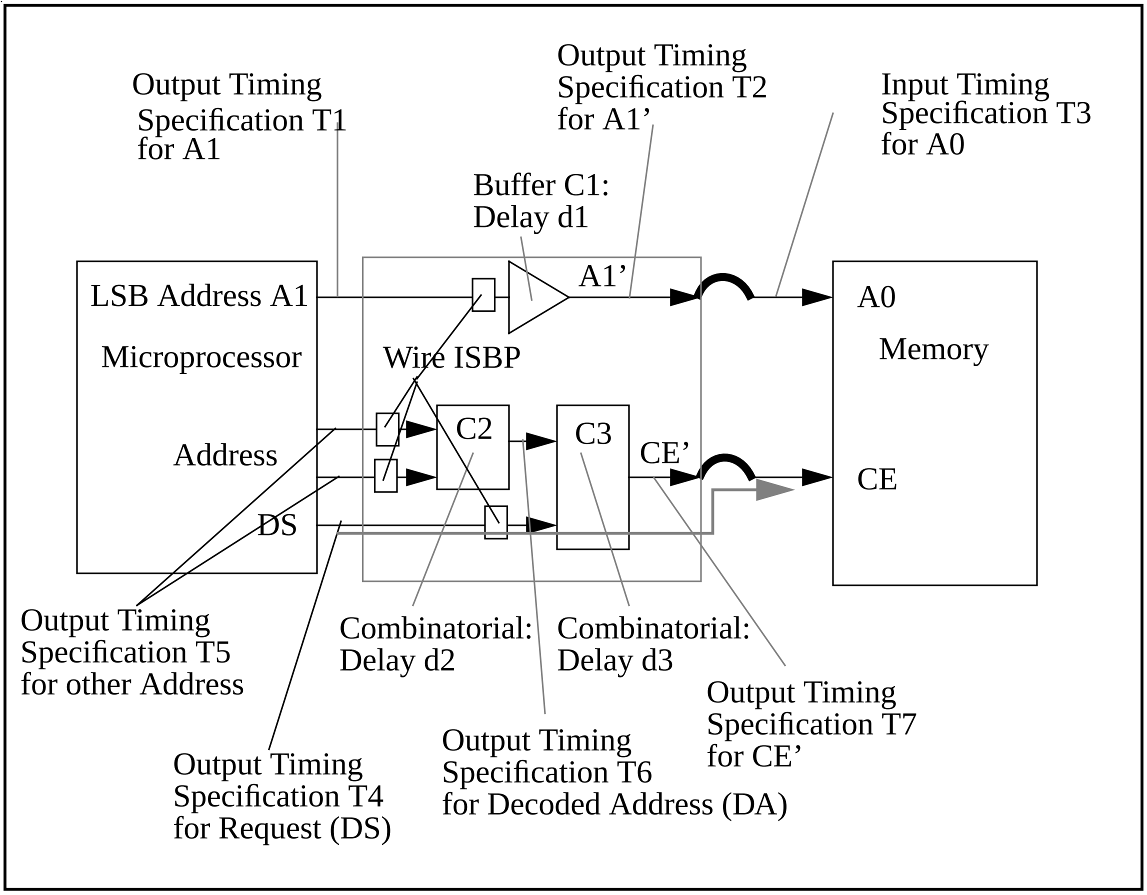

Output Timing Specification T1 for A1
Output Timing Specification T2 for A1'
Input Timing Specification T3 for A0
Buffer C1: Delay d1
A1'
LSB Address A1
Microprocessor
Address
DS
Wire ISBP
C2
C3
CE'
A0
Memory
CE
Output Timing Specification T5 for other Address
Combinatorial: Delay d2
Combinatorial: Delay d3
Output Timing Specification T7 for CE'
Output Timing Specification T4 for Request (DS)
Output Timing Specification T6 for Decoded Address (DA)

FIGURE 6-24. Example Interface for an Address Signal

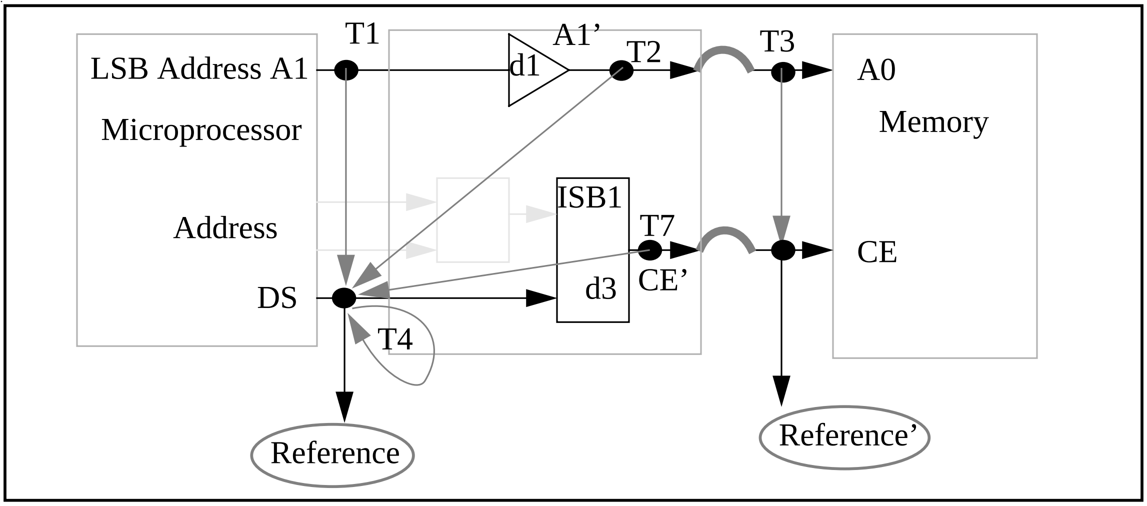


FIGURE 6-25. Relative Timing Relationships for Example Interface

The Interface Designer also determines the signal timing T7 of the CE' signal relative to the microprocessor reference DS signal. The component manufacturer specifies the timing relationship T3 of the memory A0 signal relative to the CE signal.

The microprocessor signal A1' is connected to memory signal A0. To assure that the memory component will operate as specified by the component manufacturer, the Interface Designer must assure that the timing parameters of the A1' signal satisfy the timing parameters of the A0 signal. In order to compare the timing parameters of two signals, both signals must be given relative to the same reference events. Signal CE' is to be connected directly to the signal CE which is the reference for the A0 signal in timing T3. A timing TX is to be found, as shown in Figure 6-26(c), that gives the timing of signal A1' relative to CE'. Timing parameters between TX and T3 can therefore be compared since both timings are relative to the same reference (CE' connects to CE). Figure 6-26 shows how the timing parameters for timing TX are obtained.

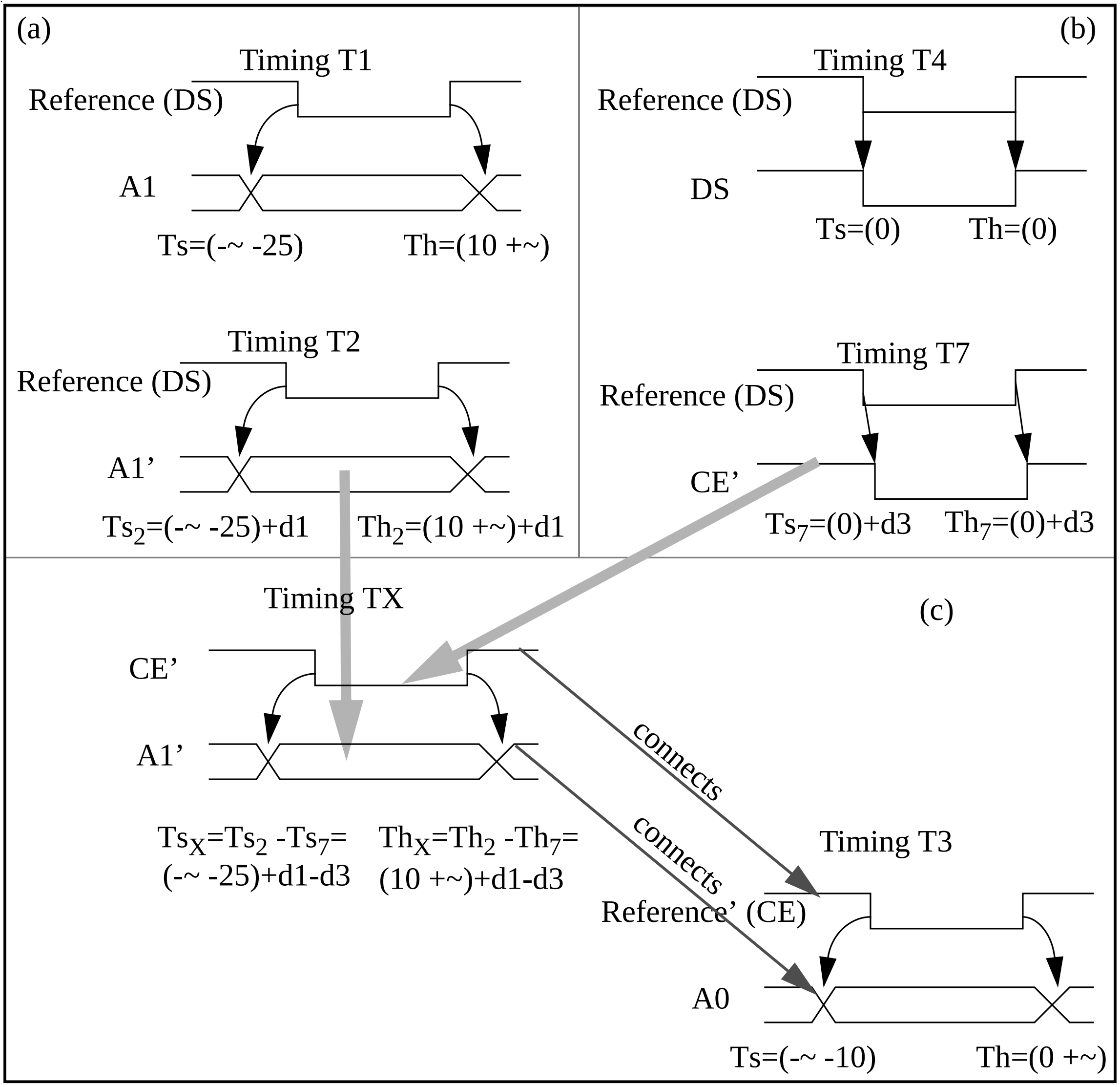


FIGURE 6-26. Finding Timing TX of A1' relative to CE'

In Figure 6-26(a) timing T1 is a Strobe timing with a setup time of 25ns and a hold time of 10ns relative to the reference signal (DS). Timing T2 of A1’, determined by the Interface Designer, also is a Strobe timing with a setup time of (-~ 25)+d1 and a hold time of (10 +~)+d1 relative to the same reference signal (DS), where d1 is an ISBP parameter propagation delay.

In Figure 6-26(b), timing T4 is a Logic timing with a setup and hold time of 0ns relative to the reference signal (DS). Timing T7 of the CE’ signal, determined by the Interface Designer, is also a Logic timing with a setup and hold time of 0ns+d3 = d3, relative to the same reference signal (DS), where d3 is an ISBP parameter propagation delay.

The events on the CE’ signal in T7 occur d3 ns later than the events on the reference DS. Since DS is the reference signal in both T2 and T7, it can be concluded that the events on the A1’ signal occur d3 ns earlier relative to the events on the CE’ signal. These relationships are shown as timing TX in Figure 6-26(c). The A1’ signal has a setup time parameter of (-~ 25)+d1-d3 and a hold time parameter of (10 +~)+d1-d3 relative to the CE’ signal. It should be noted that timing TX is a Strobe timing since signal timings are small delay invariant, and both d1 and d3 are small delays.

This example showed that the timing relationship of the reference events between components is required to verify the timing parameters of two signals being connected. In the example shown in Figure 6-26, the required timing relationship is between the DS reference signal on the microprocessor and the CE reference signal on the memory. The timing relationship between these two signal is given in timing T7. As it turns out, the timing of the reference events between components is always available in the form of *request* information output of the IB. In fact, the *request* information was designed primarily to provide a simple, consistent method to obtain the timing relationship between reference events between components. The Interface Designer will connect all information ports found on a component, and in doing so will always generate a connection for the *request* information ports. Due to the design methodology employed, the timing of *request* information will always be in the form of a Logic timing. For timing verification and constraint extraction, the Interface Designer directly looks at the IB *request* information output port that is connected to the slave component *request* information input port to determine the timing relationship between the reference events of components.

6.9.1.1 Extracting the Timing Constraints

To assure setup and hold time parameters of T3 are satisfied, timing constraints are extracted using the derived timing TX. Figure 6-26(c) shows the input setup time of A0 as (-~ -10) relative to the CE signal while the setup time of the A1′ as (-~ -25)+d1-d3 relative to the CE’ signal (which is connected directly to the CE signal). To assure that the A1′ output signal does generate a signal that meets the A0 input specification, the Interface Designer must assure that the timing parameter interval (-~ -25)+d1-d3 of the A1′ signal falls within the timing parameter (-~ -10) of the A0 signal.

The *contains-interval* operator was developed to provide the Interface Designer with a method to express a constraint that must be satisfied to assure that one interval falls within another interval. The expression

$$\text{((A B) contains-interval (C D))} \qquad \text{(EQ 6-11)}$$

represents a constraint that states that interval (C D) falls within the interval (A B). The constraint is satisfied (true) iff A ≤ C and B ≥ D as illustrated in Figure 6-27. The upper and lower *margins* indicated by how much interval (C D) falls within interval (A B). Positive margins indicate that the constraint is satisfied, while any negative margin indicates that a constraint fails.

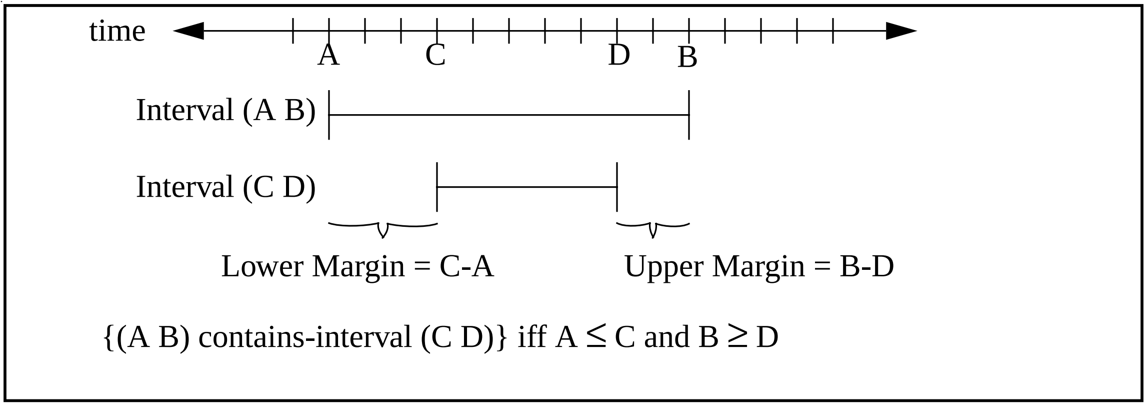


FIGURE 6-27. Contains Interval Operator

Using the contains-interval operator, a constraint for the setup time of the A0 and A1′ signals can be written as:

$$\text{\{(-~ -10) contains-interval ((-~ -25)+d1-d3)\}} \qquad \text{(EQ 6-12)}$$

Figure 6-28 illustrates this constraint graphically. It shows how the timing parameter Ts_{out} of the output specification of the `A1`’ signal falls within the input timing parameter Ts_{in} of the `A0` signal input specification.

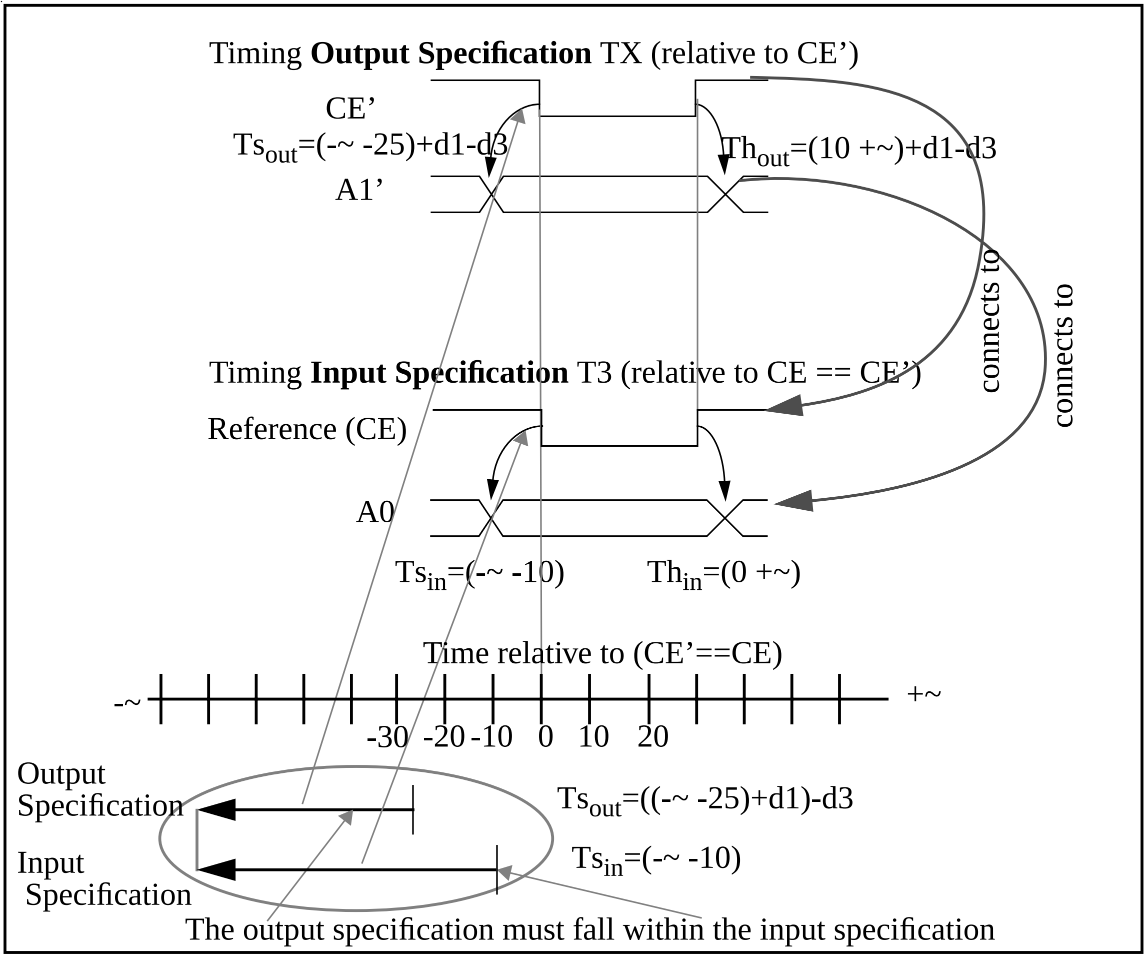

Timing **Output Specification** TX (relative to CE’)
CE’
Ts_{out}=(-~ -25)+d1-d3
Th_{out}=(10 +~)+d1-d3
A1’
connects to
connects to
Timing **Input Specification** T3 (relative to CE == CE’)
Reference (CE)
A0
Ts_{in}=(-~ -10)
Th_{in}=(0 +~)
Time relative to (CE’==CE)
-~
+~
-30
-20
-10
0
10
20
Output Specification
Input Specification
Ts_{out}=((-~ -25)+d1)-d3
Ts_{in}=(-~ -10)
The output specification must fall within the input specification

FIGURE 6-28. Constraint Output and Input Specification

For the hold time of the `A0` signal the constraint is:

$$\{(0 +\sim) \text{ contains-interval } ((10 +\sim) + d1\text{-}d3)\} \quad \text{(EQ 6-13)}$$

When the interface is implemented in a given technology, the delay parameters d1 and d3 will be found. The timing constraint shown in Equation 6-13 can then be evaluated to verify that the input timing specification of `A0` is satisfied.

6.9.1.2 Constraint Extraction Rules

For every IB output to component input connection, a set of connection timing constraints must be found. The timing constraints extracted depend on the signals’ timing templates. A set of constraint extraction rules was developed to extract the timing constraints for each possible combination of IB output timing templates and component input templates as outlined in Figure 6-29. The steps for timing constraint extraction are summarized in Table 6-12: extracting a constraint for the setup time and hold time of the input timing or in some cases extracting a constraint for the initiate to terminate time interval as discussed in Section 6.9.3.

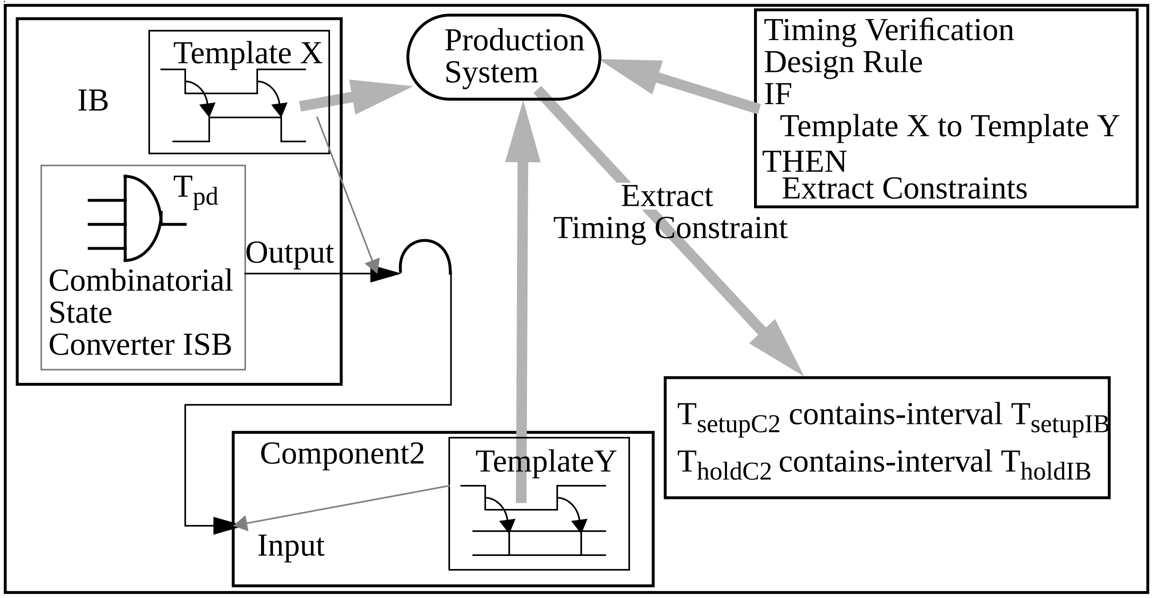

IB
Template X
T_{pd}
Combinatorial State Converter ISB
Output
Production System
Timing Verification
Design Rule
IF
Template X to Template Y
THEN
Extract Constraints
Extract Timing Constraint
$T_{setupC2}$ contains-interval $T_{setupIB}$
T_{holdC2} contains-interval T_{holdIB}
Component2
TemplateY
Input

FIGURE 6-29. IB Constraint Extraction Rules

Interface Block Output Timing	Component Input Timing	Steps for Timing Extraction
Strobe, Logic	Pulse	Extract initiate-terminate constraint
Strobe, Logic	Strobe	Extract setup & hold constraints (Eq. 6-11)
Latch	Latch	Extract setup & hold constraints (Eq. 6-11), include ALE delay
Follows, Strobe, Logic	Follows-Latch	Extract hold (Eq. 6-11) & initiate-terminate constraints
Follows, Strobe, Logic	Follows-Pulse	Extract hold (Eq. 6-11) & initiate-terminate constraints
Handshake	Handshake	Extract hold constraint (Eq. 6-11)
Wai**t**	Wait	Extract setup constraint (Eq. 6-11)

TABLE 6-12. Steps for Timing Constraint Extraction

6.9.2 Choosing an Implementation Technology

The ISBP parameters are represented as unknown variables within the constraints. An implementation technology is now chosen which assigns a fixed range of values to every ISBP parameter found within the IB. The choice of implementation technology is up to the user of the Interface Designer, and often is important for a successful design. Various factors will affect the choice of implementation technology, such as cost, speed power consumption and availability. Careful consideration must be given to the proper speed of the implementation technology. If the implementation technology is too slow for the components being connected, it may be impossible for the Interface Designer to generate an interface where all timing constraints are satisfied.

6.9.3 Calculating the Initiate-Terminate Delay

Up to this point, the Interface Designer has completed the design of the interface and has extracted timing constraints, that if satisfied will assure correct operation. Once the implementation technology had been chosen, the Interface Designer could assign values directly to all ISBP parameters with one exception: the adjustable propagation delay (for example d_{var} in Figure 6-17) in Leading and Trailing Edge ISBPs. This delay is used to adjust the initiate to terminate delay of the reference events on the master, and can now be determined. If the timing template of the `delay` information is a Pulse timing, the initiate to terminate delay provided by the master is fixed and can not be changed.

The timing constraints extracted include several that place a lower limit on the initiate to terminate delay. For example, the a memory device usually places a restriction on the initiate to terminate delay: it must be at least as long as the access time. The Interface Designer systematically searches through all timing constraints involving the initiate and terminate delay to check for restrictions, and from the restrictions calculates a lower limit. This lower limit is then used to calculate and set the d_{var} ISBP parameter of the Leading and Trailing edge ISBPs. For example, the d_{var} ISBP parameter of a Leading Edge Delay ISBP shown in Figure 6-30 is calculated as follows. An IB Handshake delay output timing

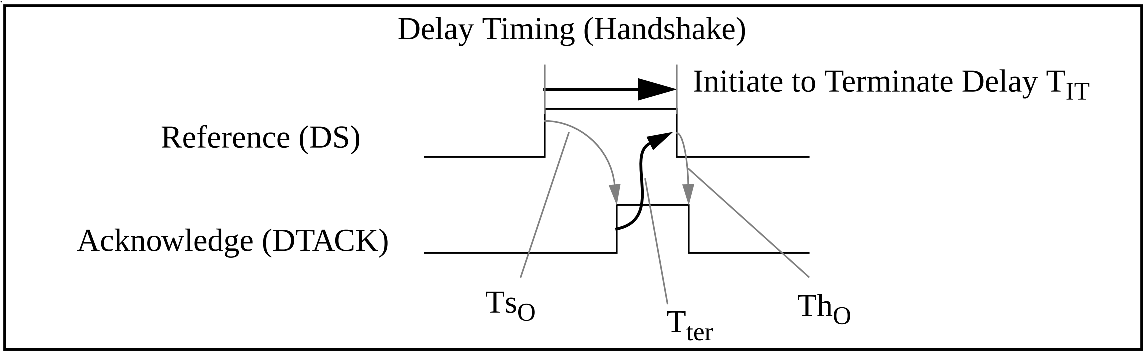


FIGURE 6-30. Example Handshake Delay Timing of a Microprocessor

is connected to the Handshake delay input timing of the microprocessor. For the microprocessor we know that the initiate to terminate delay T_{IT} is the sum of Ts_O and T_{ter}:

$$T_{IT} = Ts_O + T_{ter}$$

which allows Ts_O to be written as

$$Ts_O = T_{IT} - T_{ter} \qquad \text{(EQ 6-14)}$$

For the IB Handshake output timing from Figure 6-18 we know that

$$Ts_O = Ts_I + d_{var} + d_{prop}$$

where Ts_I is the setup time of the input signal of the Leading Edge Delay ISBP. This allows us to solve for d_{var} by substituting Ts_O

$$d_{var} = T_{IT} - T_{ter} - (Ts_I + d_{prop}) \qquad \text{(EQ 6-15)}$$

All parameters on the RHS of Equation 6-15 are known: T_{IT} is the initiate to terminate delay determined from the constraints, T_{ter} is the terminate delay from the handshake signal timing of the component, d_{prop} is the propagation delay of the AND gate used to build the Leading Edge Delay ISBP and Ts_I is the setup time of the signal entering the Leading Edge Delay ISBP.

6.9.4 Timing Constraint Evaluation and Verification

A timing constraint is a relationship between ISBP parameters and timing parameters. Both the ISBP parameters and the timing parameters are expressed as *time intervals* representing a range of time values. Two types of timing constraints were relevant in this work: inequality timing constraints such as ($A \leq B$) and ($A \geq B$), involving single value setup and hold times A and B, and the contains-interval connection timing constraints involving time intervals X and Y (X contains-interval Y). The inequality constraints can be written using a contains-interval constraint:

($A \leq B$) can be written as ((-~ B) contains-interval (-~ A))

($A \geq B$) can be written as ((B +~) contains-interval (A +~))

The Interface Designer therefore only has to be able to evaluate constraints involving the contains-interval operator.

To evaluate the contains-interval constraint (X contains-interval Y), the end points of the time intervals X and Y are calculated, and a check is performed to see if the end points of interval X enclose the endpoints of interval Y (see Figure 6-27). Calculating the end points of an interval requires evaluation of arithmetic expressions using addition and subtraction of time intervals. Evaluation of arithmetic expressions using intervals is called *interval arithmetic*. For example, if an event of a signal that has a given timing parameter to a reference passes through an ISBP such as a buffer, a timing parameter for the delayed signal relative to the same reference can be obtained by adding the ISBP parameter to the propagation delay as shown in Figure 6-31. The addition of two intervals is represented using the symbol $\oplus$. Conversely if the reference is passed through an ISBP such as a buffer, a timing parameter for the signal relative to the delayed reference can be obtained by subtracting the interface delay parameter from the timing parameter as shown in Figure 6-32. The subtraction of two intervals is represented using the symbol $\ominus$.

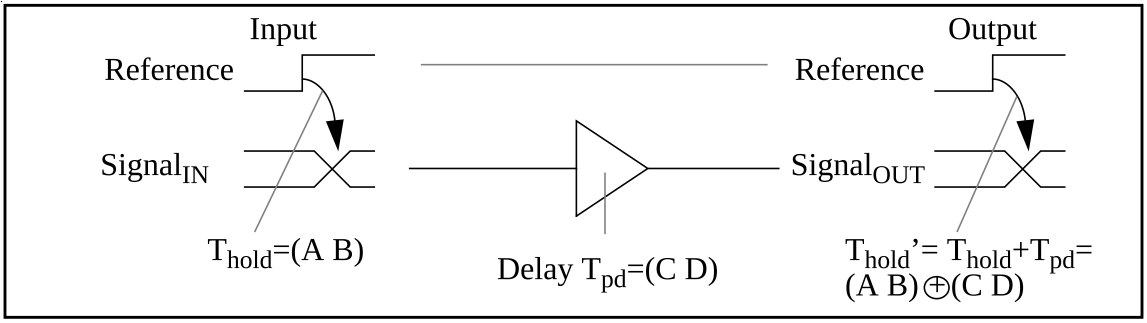


FIGURE 6-31. Delay of a Signal Relative to a Reference

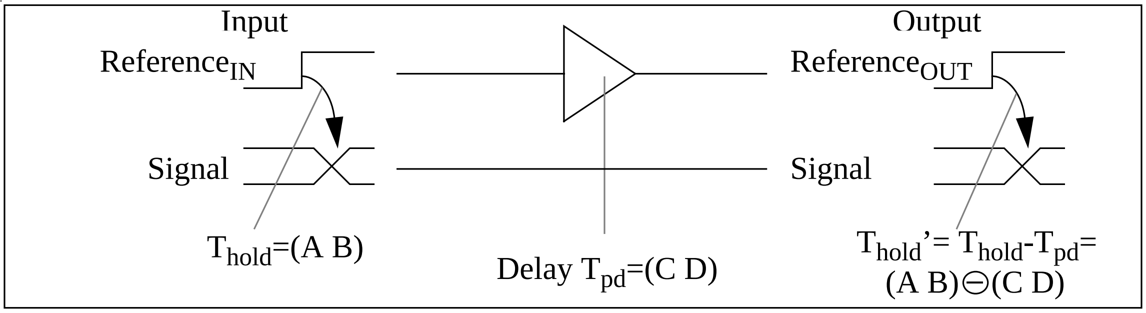


FIGURE 6-32. Delay of a Reference Relative to a Signal

When two time intervals (A B) and (C D) are added or subtracted, the range of the resulting sum or difference is determined. The range will specify the earliest possible time to the latest possible time.

For addition,

$$\text{Earliest possible time of ((A B)} \oplus \text{ (C D))} =$$

$$\text{[Earliest possible time of (A B)] + [Earliest possible time of (C D)] = A+C} \qquad \text{(EQ 6-16)}$$

and

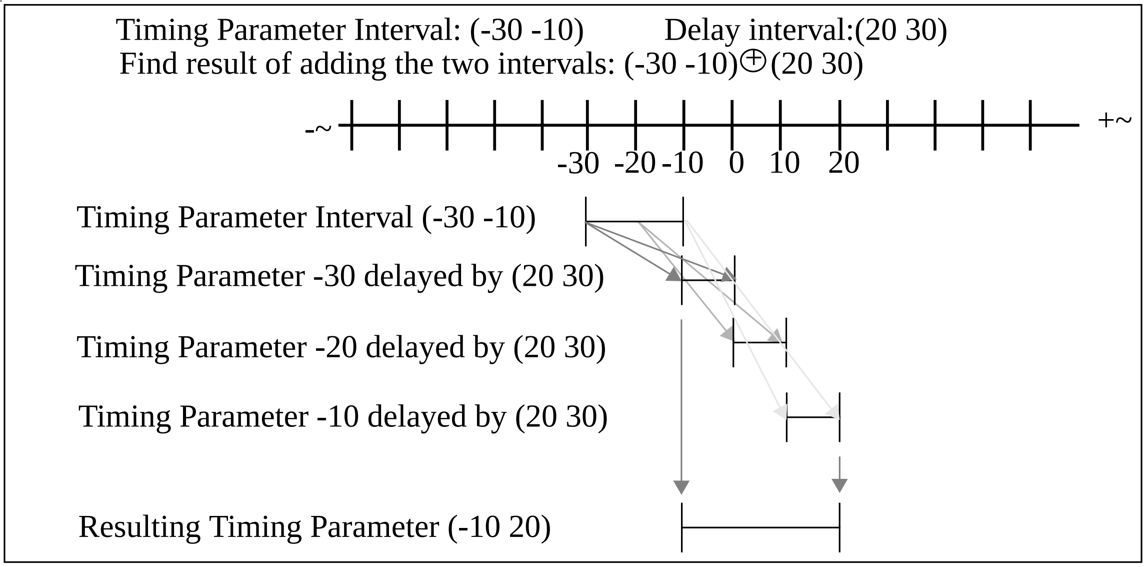
Latest possible time of((A B) $\oplus$ (C D)) =

[Latest possible time of (A B)] + [Latest possible time of (C D)] = B+D (EQ 6-17)

The Interval for the time of occurrence of the sum (A B) $\oplus$ (C D) is the interval:

(A B) $\oplus$ (C D) = (A+C B+D) (EQ 6-18)

This is illustrated in Figure 6-33 by showing what happens when a timing parameter interval of (-30 -10) is delayed by an interval of (20 30).



Timing Parameter Interval: (-30 -10)
Delay interval:(20 30)
Find result of adding the two intervals: (-30 -10)⊕(20 30)
-~
+~
-30
-20
-10
0
10
20
Timing Parameter Interval (-30 -10)
Timing Parameter -30 delayed by (20 30)
Timing Parameter -20 delayed by (20 30)
Timing Parameter -10 delayed by (20 30)
Resulting Timing Parameter (-10 20)

FIGURE 6-33. Example of Addition of a Timing Parameter and a Propagation Delay

For subtraction,

Earliest possible time of ((A B) $\ominus$ (C D)) =

[Earliest possible time of (A B)] - [Latest possible time of (C D)] = A-D

and

Latest possible time of((A B) $\ominus$ (C D)) =

[Latest possible time of (A B)] - [Earliest possible time of (C D)] = B-C

The Interval for the time of occurrence of the difference (A B) $\ominus$ (C D) is the interval:

(A B) - (C D) = (A-D B-C) (EQ 6-19)

This is illustrated in Figure 6-34 by showing what happens when a timing parameter interval of (-30 -10) occurs earlier by an interval of (20 30).

Both the interval addition and subtraction operators are associative. This means:

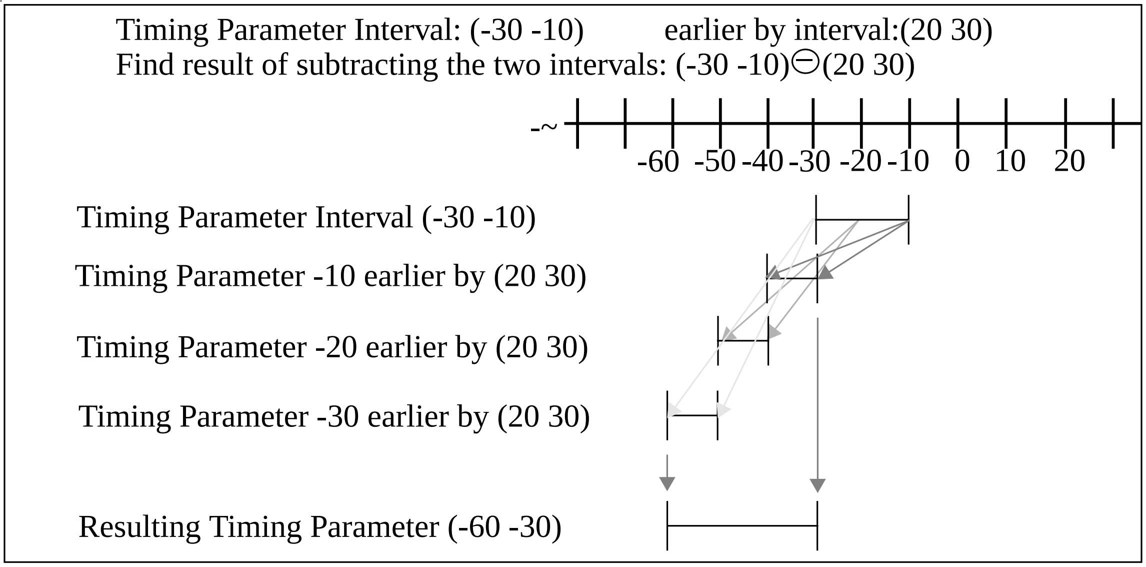

Timing Parameter Interval: (-30 -10)
earlier by interval:(20 30)
Find result of subtracting the two intervals: (-30 -10)⊖(20 30)
-~
-60
-50
-40
-30
-20
-10
0
10
20
Timing Parameter Interval (-30 -10)
Timing Parameter -10 earlier by (20 30)
Timing Parameter -20 earlier by (20 30)
Timing Parameter -30 earlier by (20 30)
Resulting Timing Parameter (-60 -30)

FIGURE 6-34. Example of Subtraction of a Timing Parameter and a Propagation Delay

$$(A\ B) \oplus (C\ D) \ominus (E\ F) = ((A\ B) \oplus (C\ D)) \ominus (E\ F) =$$

$$(A\ B) \oplus ((C\ D) \ominus (E\ F)) = (A+C-F\ B+D-E)$$

The interval arithmetic expressions given in Equation 6-18 and Equation 6-19 can be used to evaluate the timing constraints extracted. The interface example in Figure 6-26 has the two timing constraints extracted as given in Equation 6-12 and Equation 6-13. If the circuit is implemented using a technology such as LS TTL, then delay d1 will be (4 12) ns, while delay d3 will be (6 15) ns (worst case range of values for one implementation) [62] and the Equation 6-12 constraint becomes:

$$\{(-\sim -10)\ \text{contains-interval}\ ((-\sim -25) \oplus (4\ 12) \ominus ((0) \oplus (6\ 15)))\} \tag{EQ 6-20}$$

$$\{(-\sim -10)\ \text{contains-interval}\ ((-\sim -25) \oplus (4\ 12) \ominus (6\ 15))\} \tag{EQ 6-21}$$

$$\{(-\sim -10)\ \text{contains-interval}\ ((-\sim -25) \oplus (-11\ 6))\} \tag{EQ 6-22}$$

$$\{(-\sim -10)\ \text{contains-interval}\ (-\sim -19)\} \tag{EQ 6-23}$$

The interval (-~ -10) does contain the interval (-~ -19), which means that Equation 6-23 is always true and the constraint is satisfied for this implementation. The upper margin for this constraint is 9ns. The margin can be used as an indication of how tolerant the design will be to fluctuations in the timing parameters. The timing parameters are usually influenced by the type of implementation technology, operating temperature range, power supply fluctuations or semiconductor processing consistency. A large positive margin indicates that the design is robust and will still operate correctly even if the timing parameters fluctuate from the values assigned during the interface design process.

Continuing with the interface example in Figure 6-26, the Equation 6-13 constraint becomes:

$$\{(0\ +\sim)\ \text{contains-interval}\ ((10\ +\sim) \oplus (4\ 12) \ominus ((0)+(6\ 15)))\} \tag{EQ 6-24}$$

$$\{(0\ +\sim)\ \text{contains-interval}\ ((10\ +\sim) \oplus (4\ 12) \ominus (6\ 15))\} \tag{EQ 6-25}$$

$$\{(0\ +\sim)\ \text{contains-interval}\ ((10\ +\sim) \oplus (-11\ 6))\} \tag{EQ 6-26}$$

$$\{(0\ +\sim)\ \text{contains-interval}\ (-1\ +\sim)\} \tag{EQ 6-27}$$

Equation 6-27 fails since the interval (-1 +~) is not contained within the interval (0 +~), by a lower margin of -1ns. The failed constraint means that for this implementation the hold time of the `A0` signal relative to the `CE` signal is not satisfied under all operating conditions, thus resulting in an invalid design. This failed constraint shown is hypothetical for illustration purposes and would normally indicate that the selected components are incompatible with each other. Chapter 7 discusses how a failed constraint is handled by the Interface Designer.

6.10 Generating the VHDL Code

Once the design is completed and the timing constraints are verified, VHDL code is generated for the interface. This work uses the premise that at the most detailed level, all interface blocks are built up from known ISBPs (defined in Chapter 5). A VHDL primitive circuit library was created that contains architectures of these ISBPs. This means that the ISBPs that are used to build up the IB can be ‘instantiated’ as ‘components’ within the ‘architecture’ of each interface block ‘entity’[1]. Since the primitive circuit library contains an architecture for each of the VHDL primitive circuits, it will be possible to simulate or synthesize the IB by simply compiling the completed design and including the VHDL primitive circuit library. There are two important advantages to using a VHDL primitive circuit library. First, the Interface Designer is able to generate a completely structural representation of the interface using the VHDL primitive circuits and second, the architectures of the VHDL primitive circuits can be verified and optimized separately.

The Interface Designer will generate an IB for each data transfer connection request. For a microprocessor system there usually will be more than one connection request. For example a microprocessor system may consist of one or more banks of RAM memory, one or more banks of ROM memory and different IO devices. Such a system would be

[1] Quoted words are terminology used in the VHDL language and have different meaning than the same terms used previously within this work.

designed using several connection requests, one for each bank of RAM, one for each bank of ROM and one for each IO device. To aid in the simulation and synthesis of the complete microprocessor system, the Interface Designer will generate a system VHDL entity which contains all the data transfer IB VHDL entities.

Finally, a VHDL system test bench is produced for the system VHDL entity. A *test bench* is a VHDL entity that instantiates the system VHDL entity and applies simulation excitation signals (called *test vectors*) to the inputs of the IBs within the system VHDL entity. By simulating the test bench using a VHDL simulator the operation of the interface can be analyzed and verified. More importantly, the capability to simulate the resulting interface makes it possible to validate the Interface Designer. By studying and analyzing the output from the simulator, a design engineer can validate that the interface generated by the Interface Designer allows all components to operate as specified by the manufacturer. The simulation test vectors are generated automatically by the Interface Designer. The state and timing of the test vectors are extracted from the state specification and signal timings in the component library. The Interface Designer will provide test vectors that simulate a basic read cycle followed by a write cycle.

6.11 Controlling the Design Process

The design knowledge of the Interface Designer consists of a set of rules to accomplish the different tasks of the design process. The orderly execution of the tasks is controlled using context limiting and specificity ordering strategies.

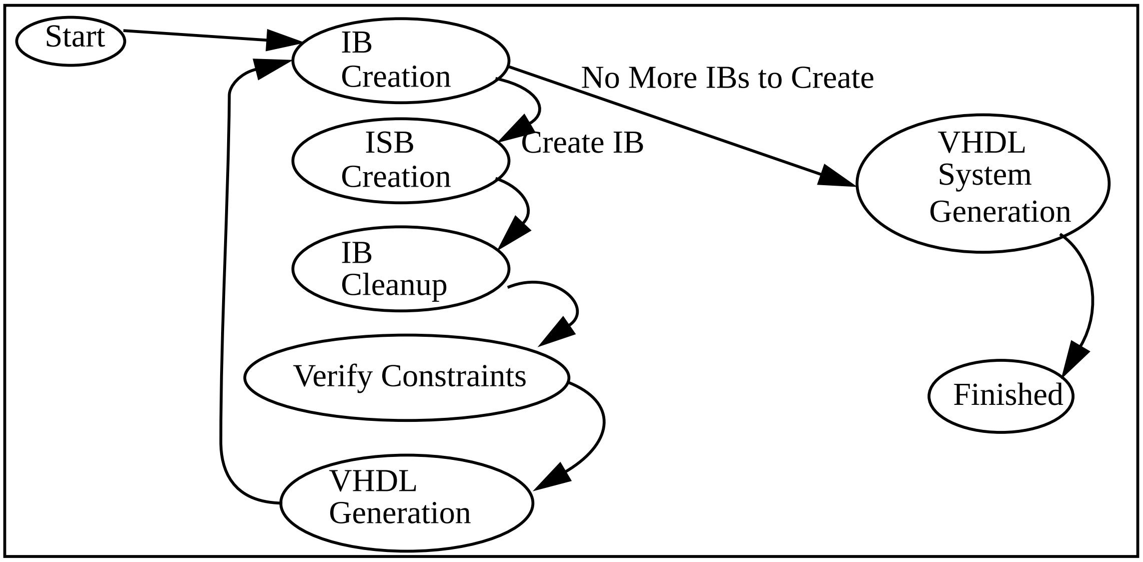


FIGURE 6-35. Design Phases used for Contexts Limiting

The Interface Designer system steps through the phases shown in Figure 6-35. Interface design starts with the IB Creation phase. Once the initial IB is created, the ISB Creation phase commences. After completion of the ISBs, unused and redundant ISBs are deleted during the IB Cleanup phase. An example of a redundant ISB is a Timing ISB that consists of a Wire ISBP. The deletion of a Wire ISBP in no way changes the behavior of the IB, it only reduces the complexity by reducing the number of ISBs. This step was added since it was found that Wire ISBPs were cluttering up the user display of the IB data structure and the large number of Wire ISBPs slowed the generation and compilation time of the VHDL code. An example of an unused ISB is an Info ISB that is used to generate an internal `decoded read` information signal, the output of which is not used anywhere in the IB. Such an ISB can be removed without affecting the operation of the interface.

Next the timing constraints will be verified during the Verify Constraints phase. Timing constraint verification is followed by generation of the VHDL code for the IB that has just been designed, during the VHDL Generation phase. If more IBs must be designed, another IB is created and the ISB Creation phase once more commences. If all required IBs have been designed, VHDL code for the system test bench incorporating all IBs is generated during the VHDL System Generation phase.

6.12 Summary of the Interface Design Process and Representation

This chapter developed the interface design process and its representation in the Interface Designer. The interface design process is divided into seven steps which closely follow the abstraction hierarchy developed for the component and interface models.

The first step creates an IB for each capability of a component that must be connected. The IB is sub-divided into Info ISBs which in turn are sub-divided into State and Timing ISBs. The State and Timing ISBs are next designed using ISBPs and the timings on the input of the ISBs are propagated to the output of the ISB. Once the IB is designed, the timing constraints are verified to see if all input timing specifications are satisfied. Finally a VHDL representation of the IBs is produced. A simulation test bench for the IBs is also generated and allows the operation of the interface to be validated using VHDL simulation tools.

Chapter 7

Data Transfer Interface Design Implementation and Results

Chapters 4 to 6 developed the knowledge representation framework and the inference process used in the Interface Designer. This chapter presents the results obtained from the Interface Designer implemented in Knowledge Craft expert system shell [17]. The Knowledge Craft shell version 4.1 runs on a Sun Microsystem SPARCstation 2 with 48MB of RAM under UNIX SunOS release 4.1.3.

Data structures in Knowledge Craft are implemented using the CRL (Carnegie Representation Language [16]), a frame based knowledge representation language which provides object-oriented programming that describes inheritance as relations between objects that are called frames. A directed graph that uses frames as nodes and the relations between frames for the links between nodes is called a *frame network*. In Knowledge Craft, the frame network can be displayed and modified using the Palm Network Editor tool. Rules are represented in CRL-OPS, a forward chaining rule based system language [16].

This chapter first presents the components entered into the Interface Designer component library. This is followed by the detailed analysis of a design example using the 68000 microprocessor and the 6116 RAM. Other design examples are provided in Appendix F to illustrate different features of the Interface Designer. The chapter concludes with a summary of the different microprocessor systems designed with the Interface Designer.

7.1 Component Library

Various components were entered into the component library using the knowledge representation techniques developed in Chapter 4. The component library is used to store all component information relevant to interface design. The development of the component library was accomplished in two phases. First prototype frames were created which represent the classes of building blocks from which components can be constructed. Second, the appropriate prototype frames were instantiated to produce the device frames that represent an actual, specific component. (For an overview of the component frames see B.1).

7.1.1 Prototype Frames

Prototype frames represent the building blocks from which component and interface block frames can be created through instantiation. Prototype frames are organized into networks of related classes using the *^is-a* relation. For example, Figure 7-1 shows the prototype frames that were created for all the signal timings developed in Chapter 4, displayed using the Knowledge Craft Palm Network Editor. The root class `TIMING` has sub-classes `PULSE_TIMING`, `PULSE_LATCH_TIMING`, etc., which in turn may have further sub-classes.

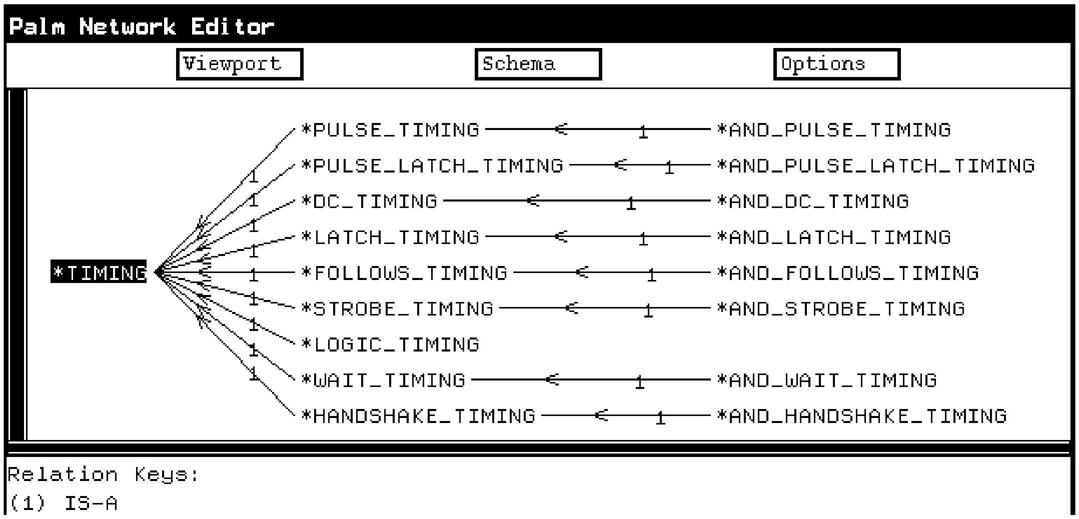


FIGURE 7-1. Class Network of Prototype Frames for Signal Timings

7.1.2 Device Frames

Once the prototype frames were designed and completed, device frames were created for different components based on the prototype frames: every device frame in a component is the instantiation of a prototype frame. For example, the Motorola 68000 microprocessor device frame was created by instantiating the microprocessor prototype frame. Figure 7-2 shows a partial network of device and prototype frames that make up the 68000 microprocessor. Both device frames and prototype frames are shown to give the reader an understanding of the *^is-a* class relations between the prototype frames and device frames, and an understanding of the relations between device frames. A frame enclosed in a box indicates it occurs more than once in the displayed frame network.

In Figure 7-2, the arrow between nodes in the frame network represents the relations between nodes. A key for the relations between nodes is given at the bottom of the figure to allow the reader to interpret the frame network. For example, Figure 7-2 indicates that

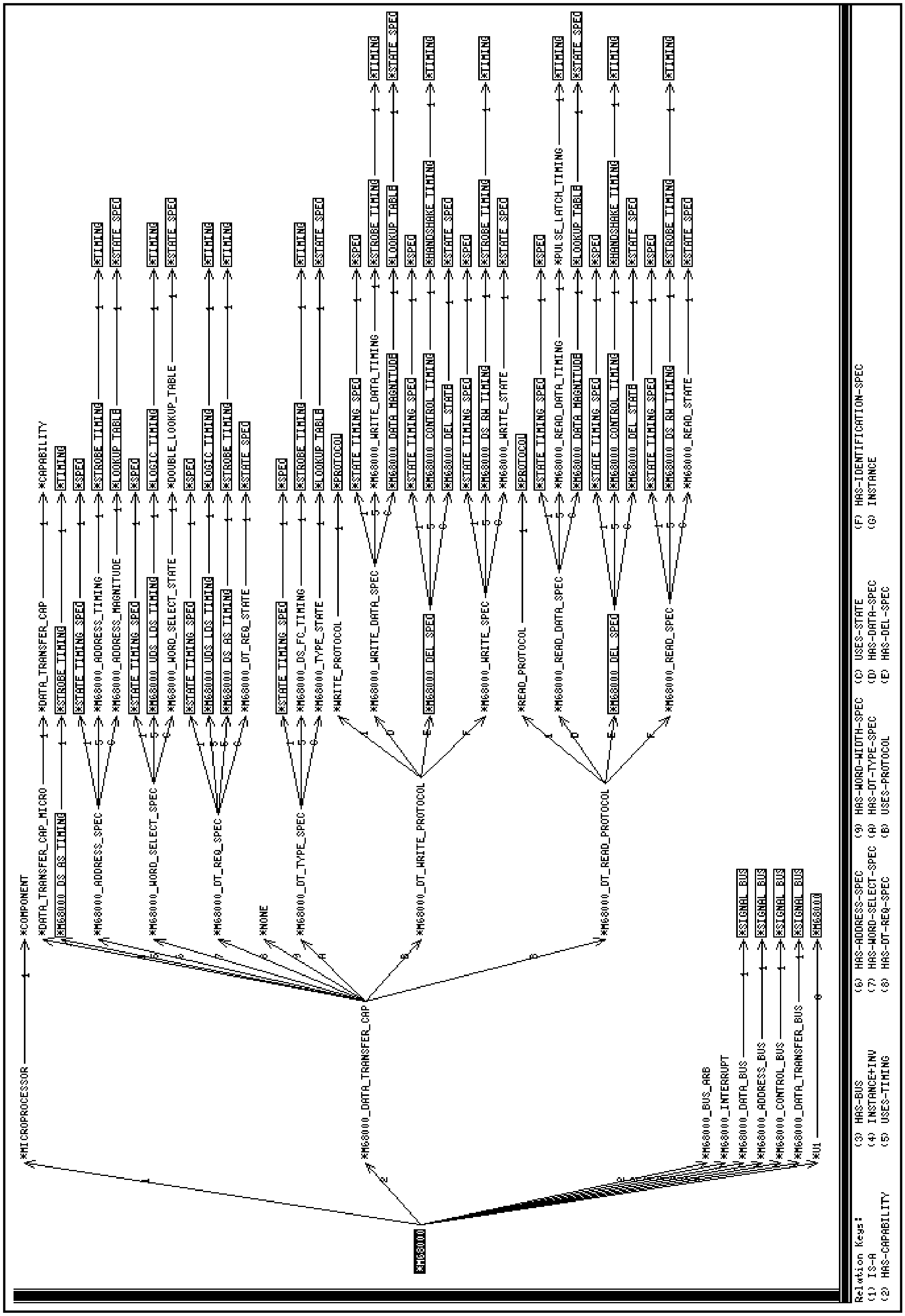


FIGURE 7-2. Motorola 68000 Microprocessor Frame Network

the M68000 (device frame) *^is-a* MICROPROCESSOR (prototype frame) which in turn *^is-a* COMPONENT (prototype frame). A complete listing of the device frames for the Motorola 68000 microprocessor (8Mhz) is given in D.1.

7.1.3 Components Represented

To illustrate the design capabilities of the Interface Designer, a cross section of components from different families and manufacturers and for different operating speeds was selected and entered into the component library. These components are listed in Table 7-1. The type column specifies if a component is a microprocessor (CPU), RAM memory, ROM memory or IO device. The name column lists the generic name for the component while the part number column gives the name from the manufacturer for a specific speed. The speed is listed as either a frequency (in Mhz), for those components that require an external clock signal to operate, or as a period for those component whose speed can be given as an access time. The address and data columns give the size of the address bus and the data bus respectively.

7.1.4 Component Entry Guidelines

Data entry into the component library requires a detailed analysis of the component specification. From the experience gained by entering the components shown in Table 7-1 into the component library, the following guidelines were established:

1. Review a component's data transfer capability.
2. Review all signals and decide which signals are involved in data transfer. Extract the signals used for each of the information transfers: *request, data, address, type, direction, word, size* and *delay*.
3. For signals determined in step 2, determine the signal states that occur during data transfer. This is the state information for information transfer.
4. Since the current design system can only handle standard TTL logic levels, check the DC logic level compatibility.
5. From the AC timing diagrams for the read and write cycle, find the initiate and terminate events. Extract the signals involved and develop their event expressions.
6. From the AC timing diagrams, determine the timing for each of the signals extracted in Step 2 for each information transfer. At this point only determine the timing template, not the specific timing parameters.
7. From the AC timing parameter tables, determine the specific timing parameter for each event relation found for the timings from Step 6.

Type	Name	Part No.	Ref.	Family	Manufacturer	Address (bits)	Data (bits)	Speed
CPU	6809	mc6809	[58]	6800	Motorola	16	8	1Mhz
	6809	mc68A09	[58]	6800	Motorola	16	8	1.5Mhz
	6809e	mc6809e	[58]	6800	Motorola	16	8	1Mhz
	68000	mc68000-8	[55]	68000	Motorola	23	16	8Mhz
	68000	mc68000-12.5	[55]	68000	Motorola	23	16	12.5Mhz
	68020	mc68020-12.5	[56]	68000	Motorola	32	32	12.5Mhz
	68020	mc68020-16.7	[56]	68000	Motorola	32	32	16.7Mhz
	z80	z80	[88]	Z80	Zilog	16	8	2.5Mhz
	z80	z80h	[88]	Z80	Zilog	16	8	8Mhz
	8085	i8085a	[43]	8080	Intel	16	8	3Mhz
	8085	i8085a-2	[43]	8080	Intel	16	8	5Mhz
	8086	i8086a-2	[41]	8086	Intel	20	16	8Mhz
	32020	tms32020	[80]	32020	TI	16	16	20Mhz
Memory	6116	cmd6116-3	[67]	SRAM	RCA	11	8	150ns
(RAM)	6116	cmd6116-9	[67]	SRAM	RCA	11	8	250ns
	6164	mcm6164-45	[59]	SRAM	Motorola	13	8	45ns
	6810	mcm6810	[58]	SRAM	Motorola	7	8	450ns
	6810	mcm68b10	[58]	SRAM	Motorola	7	8	250ns
Memory	2716	etc2716-1	[82]	EPROM	Mostek	11	8	350ns
(ROM)	2732	2732a	[44]	EPROM	Intel	12	8	250ns
	2732	2732a-2	[44]	EPROM	Intel	12	8	200ns
	2732	2732a-4	[44]	EPROM	Intel	12	8	450ns
	2764	2764a-1	[44]	EPROM	Intel	13	8	180ns
	27128	27128a-2	[44]	EPROM	Intel	14	8	200ns
	27256	27256a-1	[44]	EPROM	Intel	15	8	170ns
	27512	27512a-1	[44]	EPROM	Intel	16	8	170ns
IO (PIO)	6821	mc68b21	[58]	6800	Motorola	2	8	210ns
	8255	i8255a	[42]	8080	Intel	2	8	400ns
	8255	i82c55a-2	[42]	8080	Intel	2	8	150ns
IO (CRT)	6845	mc68a45	[58]	6800	Motorola	1	8	280ns
	6845	mc68b45	[58]	6800	Motorola	1	8	210ns
IO(UART)	6850	mc6850	[58]	6800	Motorola	1	8	450ns
	6850	mc68a50	[58]	6800	Motorola	1	8	280ns
	6850	mc68b50	[58]	6800	Motorola	1	8	210ns

TABLE 7-1. List of Components in Component Library

8. Create the device frame network of a component using the CRL language, by instantiating the appropriate prototype frames.

7.2 Design Rules

The Interface Designer is comprised of 93 design rules. Table 7-2 shows the rules grouped according to the interface design function they perform. An example CRL-OPS rule is shown in Table 7-3. This rule has been simplified for presentation purpose. The `modify-latch-to-strobe-block` rule will fire when a Timing ISB with a Latch input timing and a Strobe output timing must designed. This rule is only active during the `data_xfer_interface_creation` phase of the design. The consequents of the rule have been organized into several modular LISP routines that will fill in the appropri-

Rule Function	**Number of Rules**
IB Creation	1
Information Connection ISB creation and State ISB creation / design	36
Timing ISB creation / design	24
Timing constraint extraction	11
Implementation & timing constraint verification	1
Generation of VHDL code, Test Bench and Netlist	2
Housekeeping: context limiting, deletion of unused ISBs	18

TABLE 7-2. Rule Design Function Summary

```
;rule to fill in a Timing ISB with the appropriate logic
;if the input is latch timing and output timing is a strobe timing
;i.e. demultiplex the signal
(p modify-latch-to-strobe-block
  (interface_sub_block ^schema-name <int-block> ^instance ‘interface_sub_block
                       ^modified-by+inv <superblock>
                       ^function (member single_signal_converter <>)
                       ^status ‘new
                       ^input-timing <itim> ^output-timing <otim>
                       ^input-signal <isig> ^output-signal <osig>)
  (latch_timing ^schema-name <itim>)
  (strobe_timing ^schema-name <otim>)

  (step ^phase data_xfer_interface_creation)
-->
   (create-nonmux-signal <int-block>)
   (modify-strobe-from-latch-output-timing <int-block> <superblock>)
   (adjust-timing (get-value <int-block> ‘output-timing) <otim>
                  (get-value <int-block> ‘output-signal))
   (mark-timing-completed <itim> <otim> <isig> <int-block> ‘latch_to_stb))))
```

TABLE 7-3. Example Rule for Timing Constr10int Extraction

ate slots in the appropriate frames. The `create-nonmux-signal` routine will insert the D-Latch primitive circuit into the Timing ISB. The `modify-strobe-from-latch-output-timing` routine will determine the output signal timing parameters of the Timing ISB from its input. The `adjust-timing` routine will determine signal timing parameters of the output of the State ISB. Finally the `mark-timing-completed` routine will mark the Timing ISB as being completed.

7.3 Interface Designer Output

The Interface Designer produces a variety of outputs as shown in Figure 7-3. An *execution logfile* tracks every step of the Interface Designer. Any errors encountered during the design are recorded in the execution logfile. This helps the Interface Designer user to track down any problems encountered during the design process. After completion of the design, the Interface Designer saves all the IBs and ISBs generated in the *IB and ISB Frame File*. This file allows the user to inspect and verify any frames generated, if desired. The *Connection Netlist* file provides a listing of all the component signals being connected and their pin numbers.

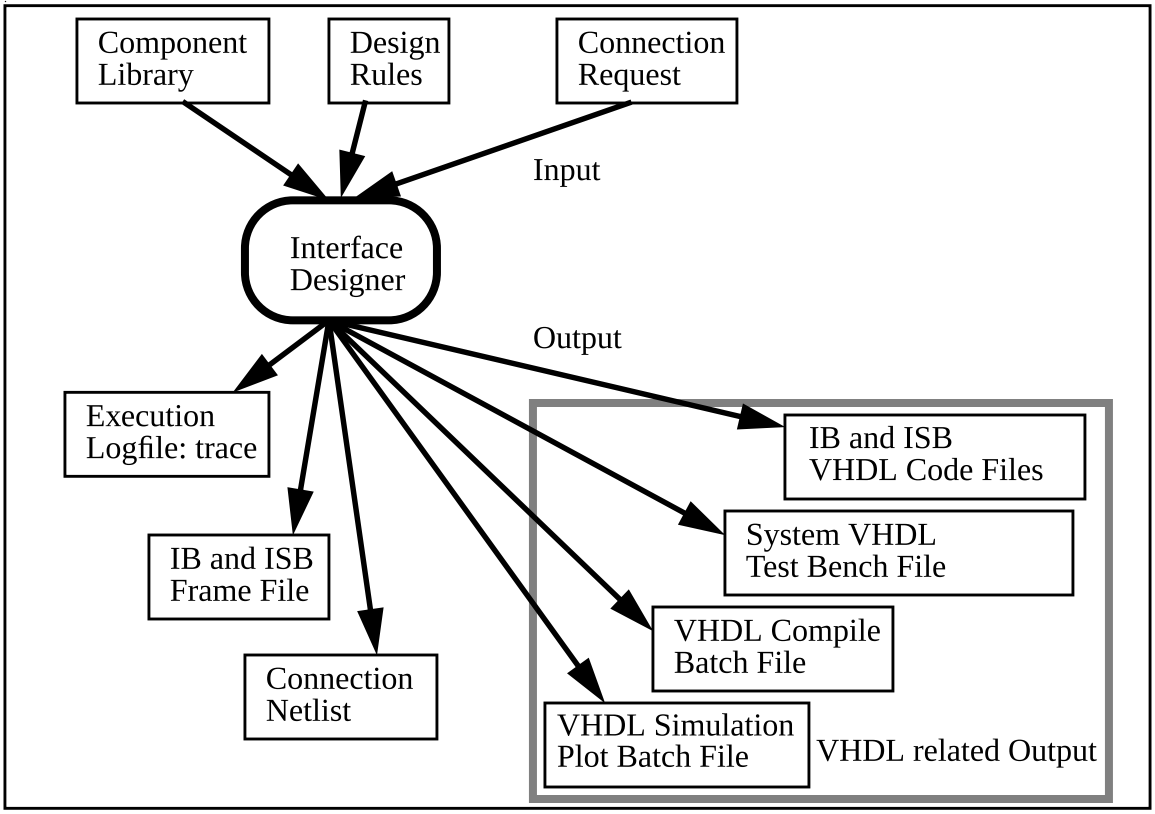


FIGURE 7-3. Interface Designer Output

The primary output of the Interface Designer are several VHDL files of the interface:

- An *IB and ISB VHDL Code* file is generated for each IB designed.
- A *System VHDL Test Bench* file is generated which instantiates VHDL entities of all IBs generated and can be used to simulate the operation of the completed data transfer interface.
- A *VHDL Compile Batch File* is produced to assist the user in the compilation of the VHDL code. Compilation of the VHDL code is required before simulation and synthesis of the VHDL code.
- A *VHDL Simulation Plot Batch File* is produced to assist the user in the display of timing waveforms from the VHDL simulator.

Both the VHDL compile batch file and the VHDL simulation plot batch file were included for the convenience of the user and they do not contain any design information. The complete design is contained within the IB, ISB and System test bench VHDL files.

7.4 Interface Design Example: 68000 to 6116

This section illustrates a complete design using the Interface Designer. The aspects of the design process discussed include the problem specification, IB and ISB design, frame representation, timing verification, VHDL code generation and the VHDL simulator output.

7.4.1 Problem Specification: 68000 to 6116

The Interface Designer was given the following microprocessor data transfer interface design problem:

- Microprocessor: Motorola 68000 (8 Mhz clock frequency)
- Memory: Four RCA 6116 (150ns access time) 2Kx8 CMOS static RAM devices
- 16-bit datapath interface between microprocessor and memory
- The memory is organized into two banks of 16-bit datapath width, mapped at address 0x000000 and 0x008000 in 24-bit address space (hex address)
- The memory is accessible from the User and Supervisor, Program and Data spaces.

An overview of the 68000 to 6116 design example specification is given in Figure 7-4. Some of the features illustrated by this design example are: address decoding, 16-bit data bus connection allowing both 16-bit and 8-bit data transfer, data bus buffering, data transfer internal `decoded type` information utilization, data transfer acknowledge generation and connection of non-multiplexed address signals.

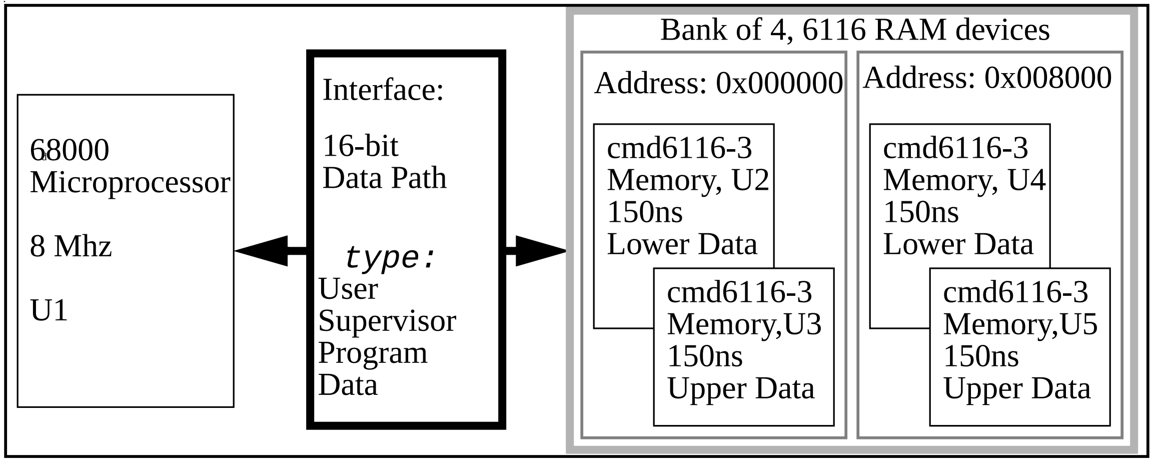


FIGURE 7-4. 68000 to 6116 Design Example Specification

The Interface Designer is provided with instances of the components being connected and a connection request. Table 7-4 shows the frames (using pseudo code for simplicity) that are passed to the Interface Designer. The complete CRL frames for the component instances and the connection request are given in D.2.

```
COMPONENT INSTANCES:
   u1: instance-of: m68000
   u2: instance-of: m6116
   u3: instance-of: m6116
   u4: instance-of: m6116
   u5: instance-of: m6116

CONNECTION REQUEST:
   connection-request-1
      purpose: data-transfer
      connect: U1 to U2, U3, U4, U5
      memory-map:   U2, U3 at address 0x000000
                    U4, U5 at address 0x008000
      type-map:     U2, U3, U4, U5 at
                            user & supervisor, program & data
      data-bus-map: U2 to U1 lower-data
                    U3 to U1 upper-data
                    U4 to U1 lower-data
                    U5 to U1 upper-data
      address-bus-map: U2, U3, U4, U5 (A0-A10) to U1 (A1-A11)
```

TABLE 7-4. Component Instances and Connection Request for Design Example

7.4.2 Execution: 68000 to 6116

To illustrate the operation of the interface design rules, the design process was interrupted after the creation and design of the `Request` information ISB. The rules that fired are shown in Table 7-5. The resulting IB frame network is shown in Figure 7-5. The `IB_1_RW_CONNECT` frame represents the interface block. `ISB_4_REQUEST_INT` is the ISB that generates the internal request signal, `ISB_4_REQUEST_INT_SIGNAL`, with a timing `ISB_4_REUQEST_INT_TIMING`, from the 68000 *`request`* information. `ISB_9_REQUEST_INT_OUT` represents the Info ISB used to generate the

Rule Fired	Function of rule
`create-rw-control-connect`	Create IB for 68000 to 6116
`create-request-int`	Create the internal request generate ISB (ISB_4_REQUEST_INT)
`create-request-in`	Create an ISB with instructions to finish the internal request generate ISB (ISB_5_REQUEST_IN) and design State ISB
`extract-conversion-blocks`	Create a Timing ISB for each input into the internal request generate ISB
`modify-strobe-to-logic-block`	Design Timing ISB (ISB_6_CONV_SS)
`modify-logic-to-logic-block`	Design Timing ISB (ISB_7_CONV_SS)
`modify-logic-to-logic-block`	Design Timing ISB (ISB_8_CONV_SS)
`modify-block-to-finished`	Indicate that the internal request generate ISB is finished

TABLE 7-5. Rules fired for Request Information ISB design

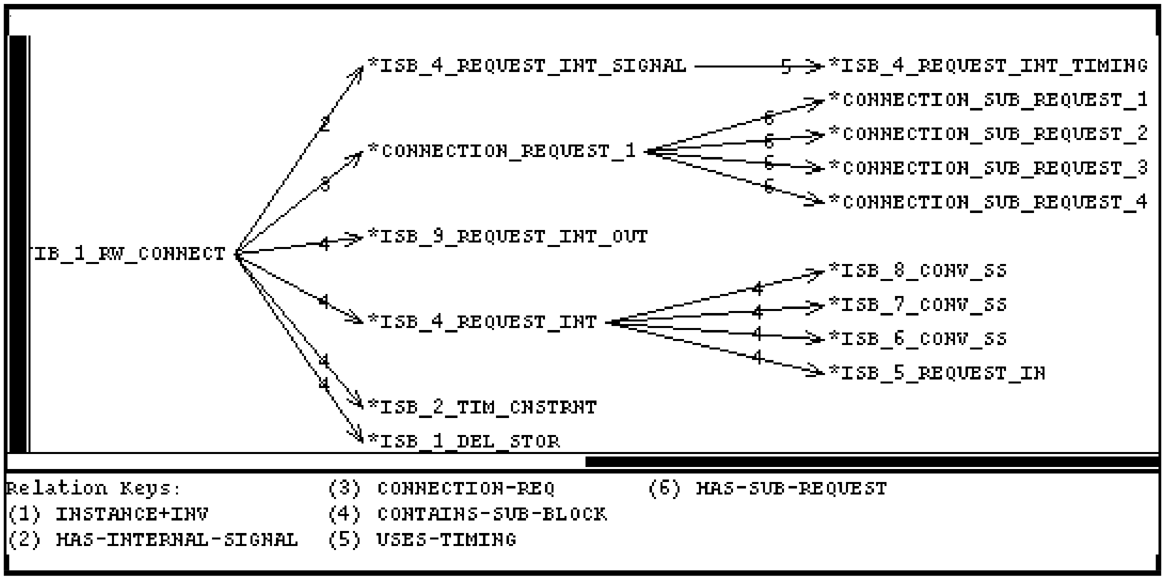
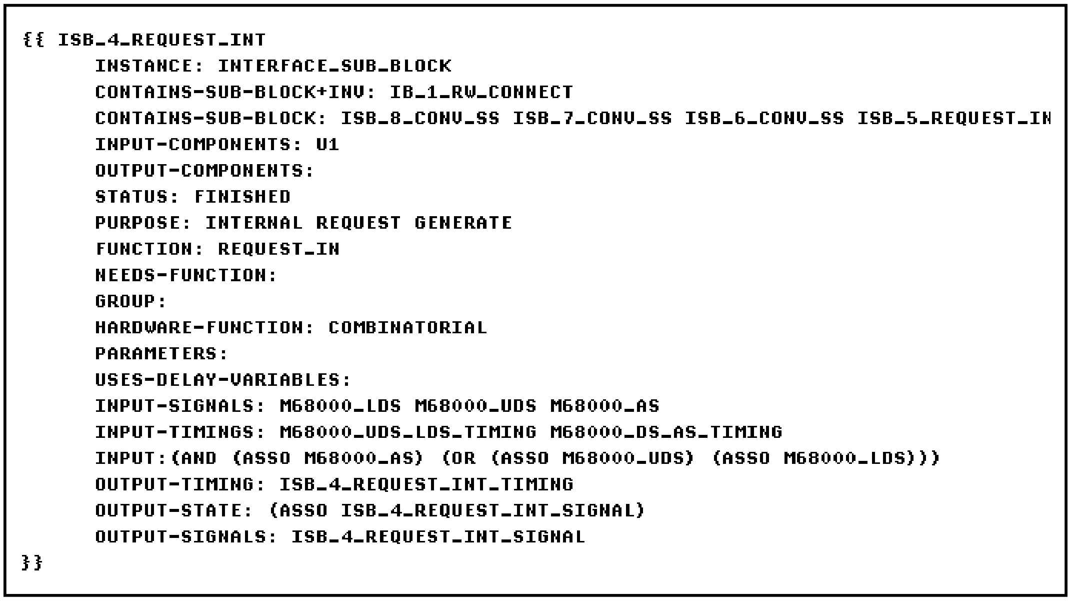


FIGURE 7-5. The Example Interface After 8 Rules Have Fired

`request` information connected to the 6116 memory. `ISB_5_REQUEST_IN` is an ISB that is used to keep track of the completion of `ISB_4_REQUEST_INT`. The rule that fires to create the `ISB_4_REQUEST_IN` ISB is also used to finalize the State ISB state equation. For convenience, the state equation for the State ISB is stored in the internal request information frame as seen in Table 7-6. A Timing ISB is provided for every signal going



```
{{ ISB_4_REQUEST_INT
      INSTANCE: INTERFACE_SUB_BLOCK
      CONTAINS-SUB-BLOCK+INV: IB_1_RW_CONNECT
      CONTAINS-SUB-BLOCK: ISB_8_CONV_SS ISB_7_CONV_SS ISB_6_CONV_SS ISB_5_REQUEST_IN
      INPUT-COMPONENTS: U1
      OUTPUT-COMPONENTS:
      STATUS: FINISHED
      PURPOSE: INTERNAL REQUEST GENERATE
      FUNCTION: REQUEST_IN
      NEEDS-FUNCTION:
      GROUP:
      HARDWARE-FUNCTION: COMBINATORIAL
      PARAMETERS:
      USES-DELAY-VARIABLES:
      INPUT-SIGNALS: M68000_LDS M68000_UDS M68000_AS
      INPUT-TIMINGS: M68000_UDS_LDS_TIMING M68000_DS_AS_TIMING
      INPUT:(AND (ASSO M68000_AS) (OR (ASSO M68000_UDS) (ASSO M68000_LDS)))
      OUTPUT-TIMING: ISB_4_REQUEST_INT_TIMING
      OUTPUT-STATE: (ASSO ISB_4_REQUEST_INT_SIGNAL)
      OUTPUT-SIGNALS: ISB_4_REQUEST_INT_SIGNAL
}}
```

TABLE 7-6. Internal Request Generation Frame for Design Example

into the `ISB_4_REQUEST_INT` (the `ISB_nn_CONV_SS` ISBs). A schematic representation of the *`request`* information circuit is shown in Figure 7-6 to give the reader a clearer picture of the signals involved. Figure 7-6 shows the circuit inside the `ISB_4_REQUEST_INT` ISB and how the ISB signals are related to signals on the 68000 and 6116. The three Timing ISBs in Figure 7-6 are wires since no timing conversion is required. It should be noted that at this point in the design, the `ISB_4_REQUEST_INT_SIGNAL` is not connected to the `ISB_9_REQUEST_INT_OUT` ISB.

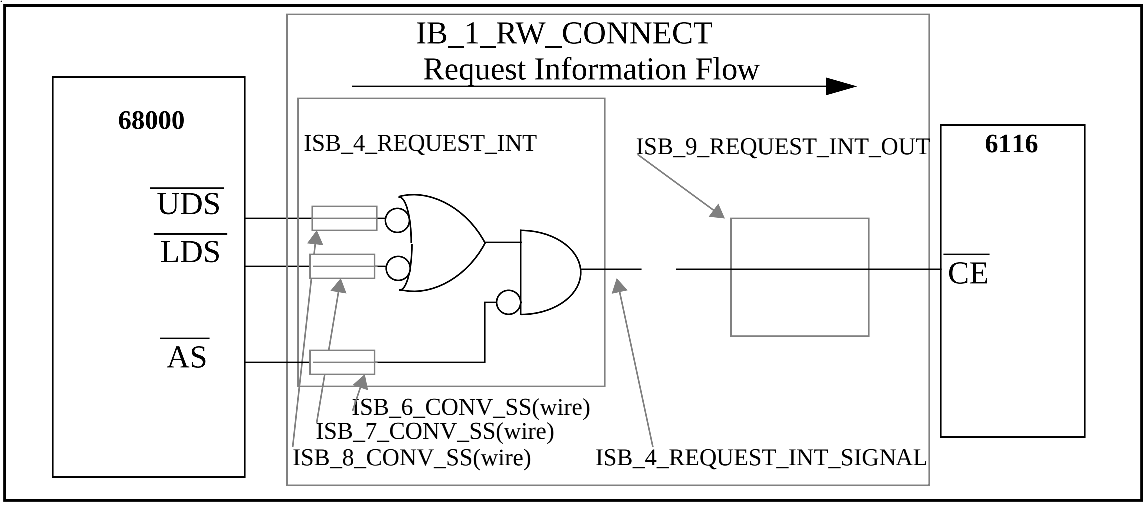


FIGURE 7-6. Request Interface Information Schematic

If the inference engine is allowed to continue the design process until completion (627 rule firings) the IB frame network in Figure 7-7 is obtained. It should be noted that some of the IB frames were removed for display clarity (for example, only some of the data signal and address signal ISB frames are shown).

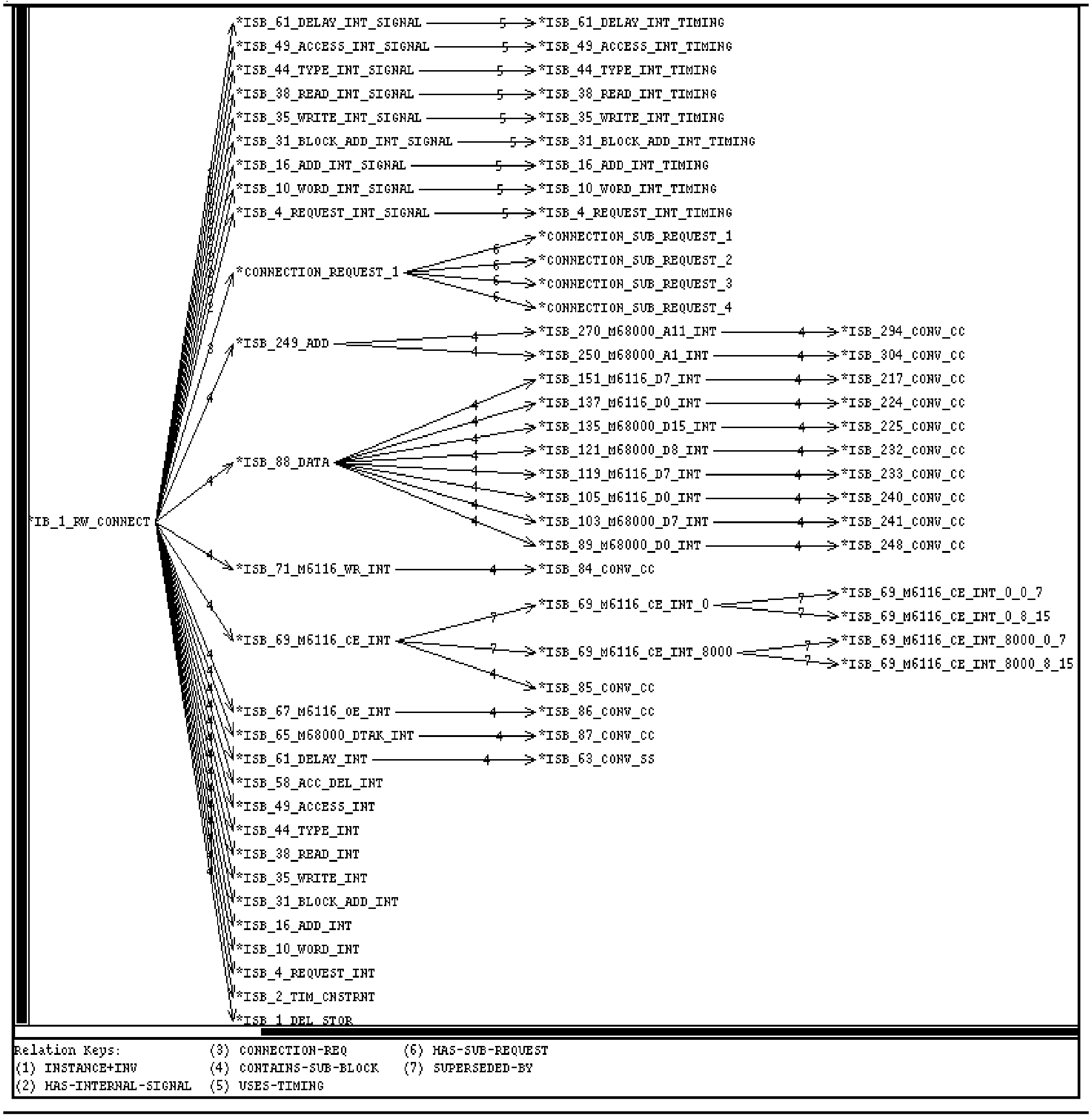


FIGURE 7-7. Completed Interface Design Example Frame Network

Internal signals and their timings are shown at the top of Figure 7-7 (`ISB_nn_xxxx_INT_SIGNAL`). `ISB_249_ADD` represents the interface between the 68000 and 6116 address signals. `ISB_88_DATA` represents the interface between the 68000 and 6116 data signals. The ISBs called ISB_nn_CONV_CC represent the interface blocks that connect the output of the IB to the input of a component. It should be noted, that for this example design, only one Timing ISB is required: ISB_69_CONV_SS, which is a Leading Edge Delay primitive circuit.

7.4.3 System Schematic: 68000 to 6116

To provide a visual representation of the interface design example, the output of the Interface Designer was drawn manually as a schematic shown in Figure 7-8 from the IB frame network. The IB schematic is given in terms of its Info ISBs to emphasize the general interface methodology and how the methodology relates to the final design output. Different information port signals from the microprocessor are decoded within ISBs to generate internal signals. There are two internal *decoded address* information signals for the two different address banks. S0 selects address 0 and S1 selects address 0x8000. There are two internal *decoded word* information signals for the upper and lower data bytes. For this design, the internal *decoded word* information signals correspond directly to the UDS and LDS signals from the 68000. The internal *decoded type* information signal gets activated whenever the User, Supervisor, Program or Data space is selected. The internal *decoded request* information signal is activated when any data transfer is in progress.

The information from the internal *decoded address*, *type*, *word* and *request* information ISBs are combined using a four input AND combinatorial circuit and applied to the CE signal on the 6116. The *direction* information from the master is connected to the *OE* and *WR* signal on the 6116 in the form of internal *decoded read* and *write* information signals. The internal *decoded read* information signal is also connected to the direction signal input (DIR) on the bidirectional buffers (shown as

) connecting the data signals. The internal *access* information signal is generated so that it is active whenever the memory devices are activated and is connected to the enable input (EN) on the bidirectional buffers connecting the data signals. The internal *access* information signal is also connected to a Leading Edge Delay input whose output connects to the *DTACK* acknowledge signal. The Leading Edge Delay takes the first edge of the internal *access* information signal (the edge that is related to the initiate transition from the reference), and delays it by an amount calculated from the timing constraints extracted during interface design, which is 76 ns for this example. The buffered address signals (A1-A11) from the 68000 are connected to the (A0-A10) address signal inputs on the 6116 memory. As shown in the schematic the lower data bus (D0-D7) from

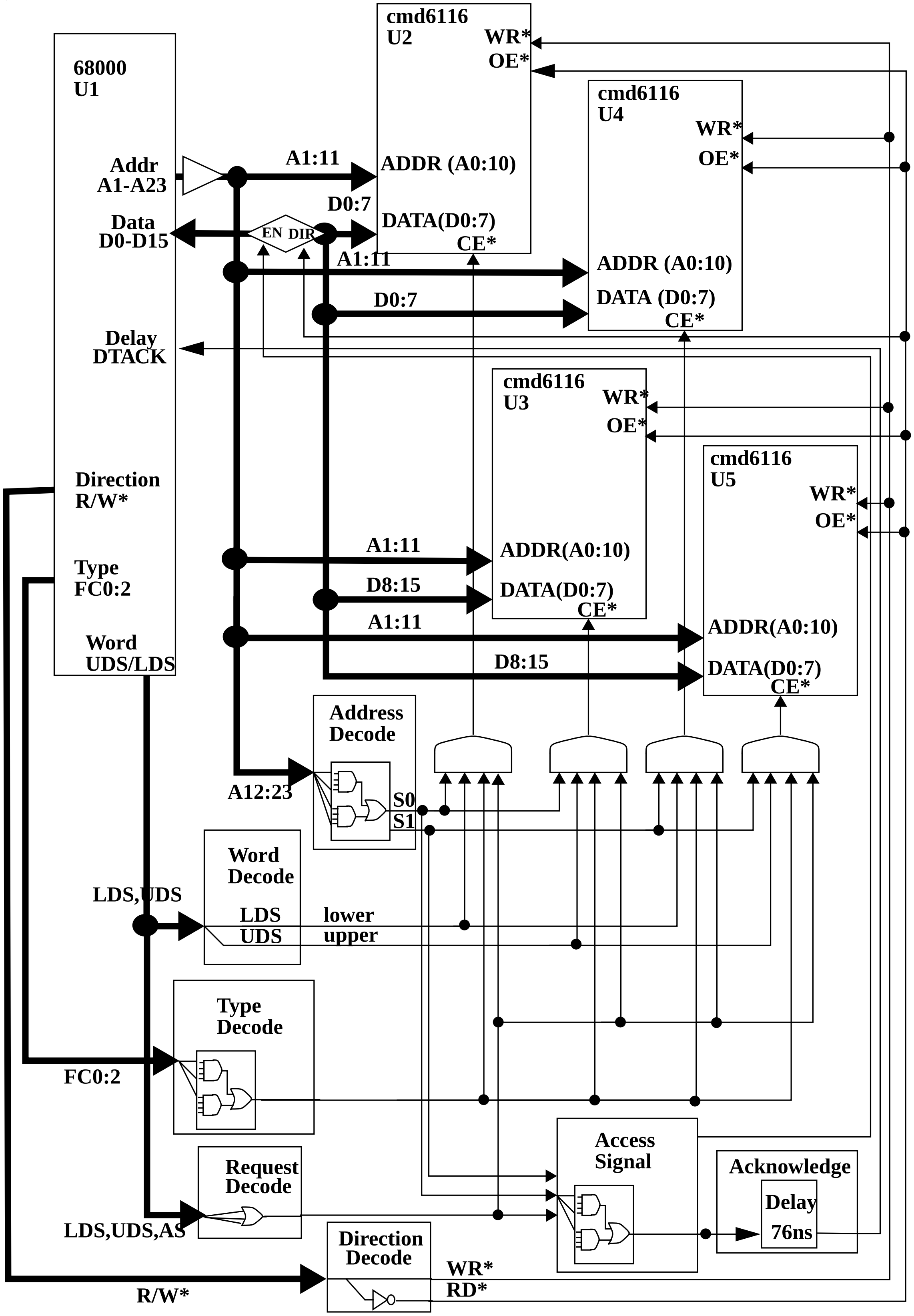


FIGURE 7-8. Schematic for Interface Design Example

the 68000 is connected to the data signals on U2 and U4, while the upper data bus (`D8-D15`) from the 68000 is connected to the data signals on U3 and U5.

7.4.4 Timing Constraint Verification: 68000 to 6116

For this work, the default target implementation technology was the XILINX XC4000 programmable logic device family [89]. This logic family was chosen due to its flexible architecture, and large range of gate counts from 3,000 (standard 2-input gates) for the XC4003 device to 25,000 for the XC4025 device. The Interface Designer allows the ISBP parameters to be specified as a triplet of values: (minimum maximum typical). For the XC4000 series the ISBP combinatorial delay parameter was specified as (3ns 7ns 5ns), the clock to output delay was specified as (2ns 4ns 3ns) and the tristate enable delay was specified as (7ns 15ns 10ns). These ranges of values were chosen since they encompass the -4, -5 and -6 speed grades of the XC4000 series devices. A triplet of values was chosen for the ISBP parameters since it allows the Interface Designer to perform worst case timing analysis during the timing constraint verification phase by using the minimum and maximum values, while also providing a typical value that can be used to investigate how the interface will perform under typical conditions in a VHDL simulation.

All timing constraints for the 68000 to 6116 interface design example were found to be satisfactory. This indicates that the Interface Designer produced a valid design from a signal timing perspective.

7.4.5 VHDL Code Output: 68000 to 6116

The Interface Designer automatically translates the IB frame network into VHDL-87 code. The VHDL-87 code was adopted by the Institute of Electrical and Electronic Engineers (IEEE) in 1987 in the form of IEEE standard 1076. The VHDL-87 standard was updated and extended in 1993 to version VHDL-93 by adding several new features. This work uses VHDL-87 since none of the VHDL-93 extensions of the language were required and because VHDL-87 has easily available compile and simulation tools. The VHDL language proved to be ideal, since it allows representation of a hierarchial data structures similar to the frame network developed for the interface. For example, the internal *`request`* generation frame from Figure 7-6 is translated into the VHDL code shown in Table 7-7.

A complete design example of VHDL code for the IB `IB_1_RW_CONNECT`, is given in E.2. The VHDL code for the ISBs used to build up `IB_1_RW_CONNECT` is given

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library DAMELIB;

entity ISB_4_REQUEST_INT is
        generic (
             TPD : time );
        port (
             M68000_AS_U1 : IN std_logic;
             M68000_UDS_U1 : IN std_logic;
             M68000_LDS_U1 : IN std_logic;
             ISB_4_REQUEST_INT_SIGNAL : OUT std_logic );
end ISB_4_REQUEST_INT;

architecture ONLY of ISB_4_REQUEST_INT is
begin
     ISB_4_REQUEST_INT_SIGNAL <=
                (( not M68000_AS_U1 ) and
                (( not M68000_UDS_U1 ) or ( not M68000_LDS_U1 ))
                ) after TPD;
end ONLY;
```

TABLE 7-7. VHDL Request Generation Entity for Design Example

in E.1, while the VHDL test bench that can be used to simulate the `IB_1_RW_CONNECT` interface is given in E.3.

7.4.6 VHDL Simulation: 68000 to 6116

Once the VHDL code is generated, the Interface Designer has completed all its assigned tasks and execution is terminated. The user can now take the VHDL code and pass it onto a VHDL simulation tool to verify the validity of the design by comparing the simulation output to the component specification from the manufacturer's data books. It should be noted that simulation is not a required step in the design process and the VHDL code can be used directly to synthesize the interface using a VHDL synthesis tool. However, for this work, verification of the correct operation of the Interface Designer necessitated the VHDL simulation of all designs generated.

It should be noted that in the VHDL simulation ISBP parameters are single values, representing typical values. If a method is developed that allows more accurate values to be found for the ISBP parameters (such as back annotation through the use of a XILINX VHDL synthesis tool), then those values can be used for the VHDL simulation instead.

The 68000 to 6116 interface design was compiled using the Mentor Graphics Qvhcom VHDL compiler. The VHDL code was simulated using the Qhsim VHDL simulator in Mentor Graphics. The resulting simulation waveforms can be seen in Figure 7-9. The excitation waveforms simulate a write cycle at address 0x008012 followed by a read cycle

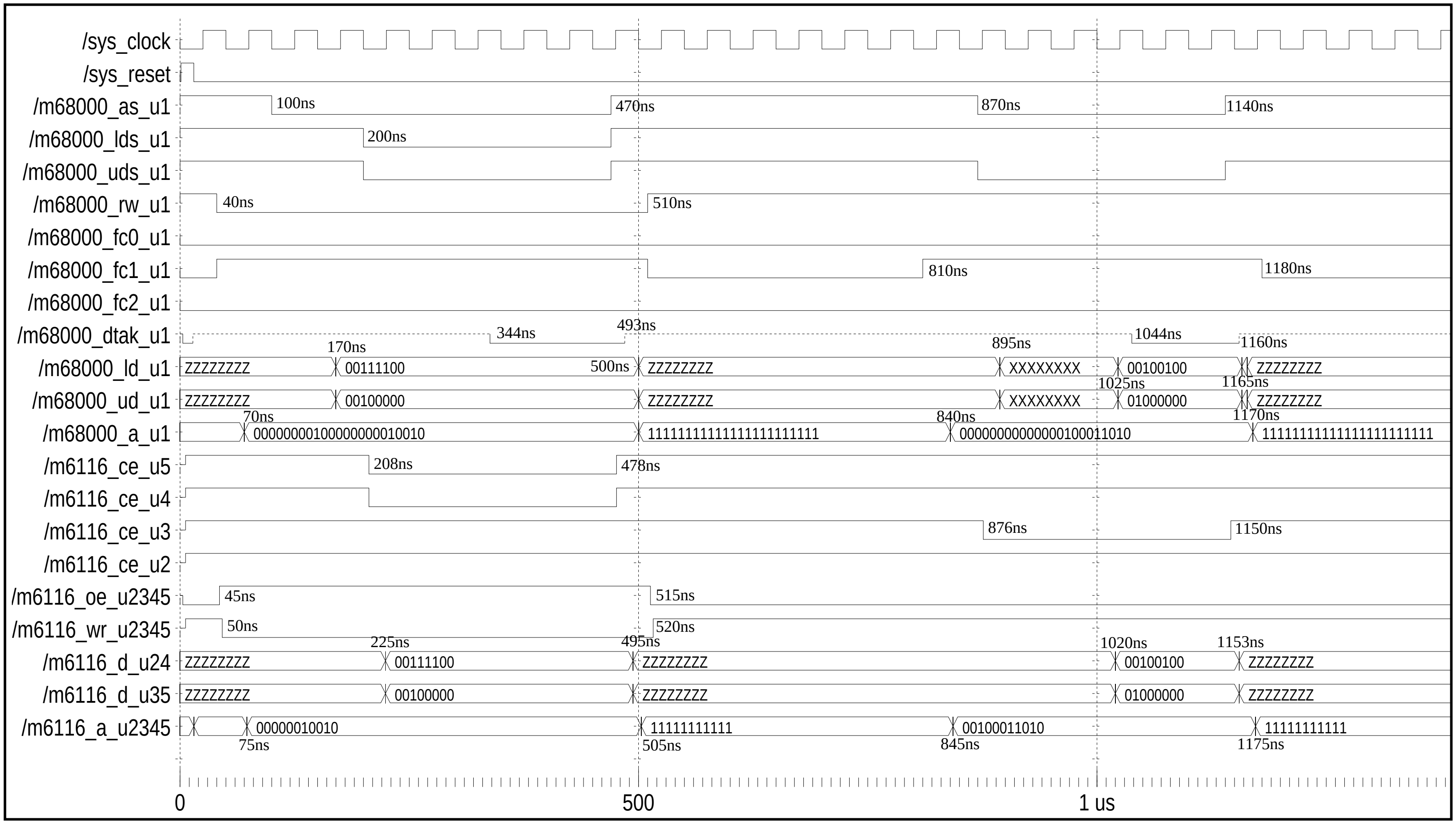


FIGURE 7-9. 68000 Design Example VHDL Simulation

at address 0x00011a. These addresses were chosen by the Interface Designer, using unique bit patterns, to fall within the memory banks starting at 0x000000 and 0x008000. Unique bit patterns are used to allow a design engineer to quickly verify the correct connection of the address signals in the timing diagram. The write cycle is a 16-bit data transfer, while the read cycle is an 8-bit data transfer. The simulation plot displays all data transfer signals on both the microprocessor and the memory. The events in the timing diagram were labelled by hand with their relative time of occurrence for analysis and discussion purposes. The interpretation of the signal states in the timing diagram is given in Figure 7-10.

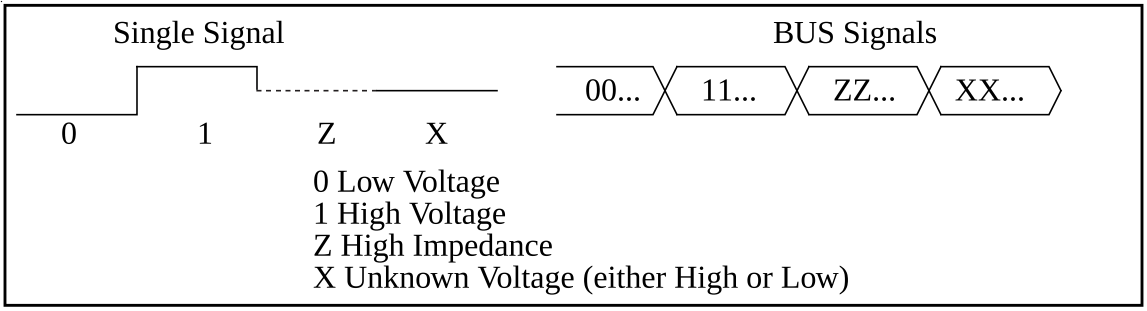


FIGURE 7-10. Simulation Timing Diagram States

The signal naming convention in the simulation plots is as follows. If a signal is named '/aaa_bbb_ccc', then aaa is the component name, bbb is the signal name and ccc is the device instance name. For the signal name 'a' is used for the address bus, while 'd' is used for the data bus, if the data bus is 8-bit wide. If the data bus is 16-bit wide, the upper

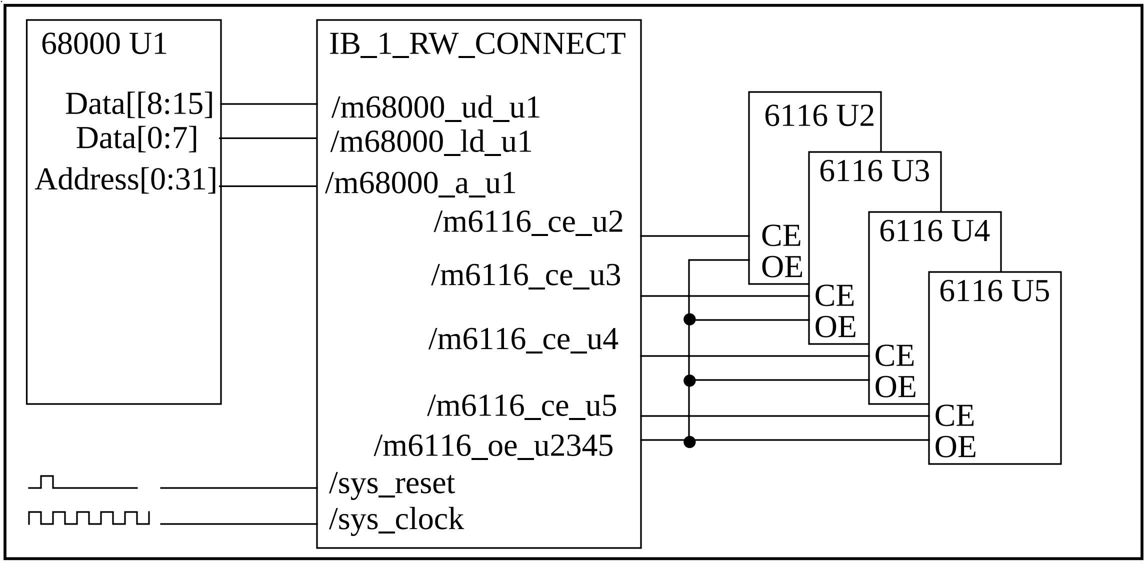


FIGURE 7-11. IB Signal Naming for Simulation

8 bits are named ‘ud’ and the lower 8 bits are called ‘ld’. For example in Figure 7-11, the upper 8 data bus signals of device U1, a 68000 microprocessor, is called /m68000_ud_u1.

The simulation plot in Figure 7-9 shows the activation of the CE signal of U4 and U5 during the write cycle starting at t=200ns at address 0x8012. For the write cycle, the 6116 OE signal is negated, while WR is asserted. The data signals from the 68000 microprocessor pass through the IB to the appropriate data signals of each 6116 RAM. For example, the /m68000_ud_u1 data signals (=0x20) pass to the /m6116_d_u35 signals. /m68000_DTAK_U1 becomes asserted at t=344ns, while the write cycle terminates at t=470ns with the negation of the /m68000_UDS_U1 and /m68000_LDS_U1 signals. Once the UDS and LDS signals are negated, DTAK becomes negated as expected.

The next cycle in Figure 7-9, a read cycle, starts at time t=870ns at address 0x011a. Only the /m68000_UDS_U1 is activated during this read cycle indicating an 8-bit data transfer. The IB correctly activates the /m6116_CE_U3 signal. The data supplied by U3 gets driven onto the 68000 upper data bus (/m68000_ud_u1) as expected. The lower data bus of the 68000 is not used during this data transfer.

7.4.7 Validation of the Interface: 68000 to 6116

To validate the data transfer interface generated by the Interface Designer, the VHDL simulation output timing diagram given in Figure 7-9 was manually compared to the Motorola 68000 data sheet[55] and the RCA 6116 data sheet[67]. The comparison was accomplished in two stages: First the state of each signal in the VHDL timing diagram was verified by checking the required state from the data sheets. For example the CE signal on the 6116 is at a low voltage level during a data transfer and high otherwise. Next, individual data sheet timing parameters were compared to the timing parameters from the VHDL simulation. Some important timing parameters that were compared are shown in Table 7-8. The time provided by the IB from the timing simulation is calculated from the relative time of occurrence of event in the timing diagram. For example, the 6116 address valid event occurs at t=75ns, while the ↓CE event occurs at t=208ns. Thus the time from address valid to ↓CE is 208ns-75ns = 133ns. The *margin* represents the difference between the timing parameter provided by the IB VHDL simulation and the timing parameter required by the input of a component. A positive margin indicates that the timing parameter provided by the output of the IB meets the timing parameter requirement of the input of the component. All timing margins were found to be positive, showing that the timing specifications given by the component manufacturers were met and therefore indicating a

Timing Parameter	**Required** **(from data sheet)**	**Provided by IB** **(from simulation)**	**Margin**
Write Cycle:			
6116 Address Valid to ↓CE	>0ns	208-75 = 133ns	+133ns
6116 ↑CE to Address Invalid	>0ns	505-478 = 27ns	+27ns
6116 Data-in Valid to ↑CE	>50ns	478-225 = 253ns	+203ns
6116 ↑CE to Data-in Invalid	>5ns	495-478= 17ns	+12ns
Read Cycle:			
6116 Address Valid to ↓CE	>0ns	880-845 = 35ns	+35ns
6116 ↑CE to Address Invalid	>0ns	1175-1150= 25ns	+25ns
68000 Data-In Valid to ↓DTACK	>-90ns	1044-1020= 24ns	+114ns
68000 ↑UDS to Data-In Invalid	>0ns	1160-1140=20ns	+20ns
68000 ↑UDS to ↑DTACK	<245ns	1163-1140= 23ns	+222ns

TABLE 7-8. 68000 Interface Timing Margins

valid design from a signal timing perspective. As well, the logic levels and the sequence of the signal events are as expected.

7.5 Timing Verification Failures

Once the Interface Designer completes the interface, it uses a chosen implementation technology with given propagation delays, setup and hold times for the ISBP parameters. The Interface Designer then evaluates the timing constraints. If no timing constraints are violated, the Interface Designer then generates the VHDL code for the interface. Since no constraints failed the designed interface is assumed to be correct and the user can proceed directly to the synthesis of the VHDL interface code without performing a VHDL simulation.

If a timing constraint fails, the Interface Designer will pause and present the user with a list of the failed constraints in the `EXECUTION_LOGFILE` file. The user is responsible for investigating the reason for the failed timing constraint and has several options to proceed. The choice of which option to use depends primarily on the severity of the failure.

1. After analyzing the failed timing constraint the user decides the components being connected are incompatible and selects a different component.
2. If the failed constraint is within approximately two implementation technology propagation delays (e.g. 20ns for LS TTL logic), the user may choose a faster implementation technology and have the Interface Designer re-evaluate the timing constraints.

3. If the failed constraint is within approximately half an implementation technology propagation delay (e.g. 5ns for LS TTL logic), the user may proceed to the VHDL simulation stage to manually check if the failure also exists in the simulated interface. The constraint evaluation process used by the Interface Designer is conservative since it uses a worst case range of values for the ISBP parameters. The VHDL simulator on the other hand uses a single ISBP parameter value to simulate the hardware implementation of the interface circuit and thus produces a more accurate and realistic estimate of the interface signal timings. If the VHDL simulation indicates that all timing parameters are satisfied, then the design can be used without modification.

Several components in Table 7-1 were found to be incompatible with each other and will always generate timing constraints that fail. The 6809 microprocessor was found to be incompatible with the Intel 2732, 2764, 27128, 27256 and 27512 EPROMs due to a data hold-time violation: the 6809 requires a 10ns hold time for the data from the EPROM, while the EPROMs only provide a hold time of zero relative to the address and chip enable. The Intel i8255 is another device that has compatibility problems with various microprocessors due to its data hold time requirement for a write cycle. The i8255 and i82c55a-2 both require a 30ns hold time, while most microprocessors in the list provide less than 30ns. The incompatibilities between components found are not unique to the Interface Designer, but also would have been discovered by a human design engineer. Once discovered, the design engineer has two choices: Re-design using different components or generate an exceptional, complex interface. The Interface Designer will always re-design using different components.

If two components are found to be incompatible, one of the components must be replaced with a component of similar functionality and the Interface Designer must be invoked again with the new set of components. A case data base for the incompatible components could be constructed to avoid failures due to incompatibility in any future design.

7.6 Summary of Designs

Table 7-9 presents a summary of the designs used to test the DAME data transfer Interface Designer. The 68000 design was presented in this chapter while others are presented in Appendix F. The designs were selected to use a cross section of devices with different complexity and speed.

The last table column lists the wall clock execution times to design completion, including the generation of the VHDL code representation. The execution times shown are only approximate and will vary according to the CPU use by other users. Due to the large memory footprint and high CPU utilization of Knowledge Craft, it is better to have a dedicated system to run the Interface Designer. Approximately 30% of the execution time was required to generate the IB and ISB frames, 30% of the time was required for the timing constraint verification, while 40% of the time was required to convert the frame representation for the IBs and ISBs to VHDL representation. As can be seen from Table 7-9, the design time is approximately proportional to the total number of devices in the system (about 4.5 minutes/device).

Master	Slave	Address	Data Path Width	Total Time min:sec
mc68000-8	4 * cmd6116-3	0x000000, 0x008000	16 bits	18:20
i8086a-2	4 * cmd6116-3 2 * etc2716-1 1 * i8255a	0x00000, 0x08000 0x0c000 0x0e400	16 bits 16 bits 8 bits	31:09
mc68020-12.5	4 * mcm6164-45 2 * m27128 1 * m6810	0x00008000 0x00000000 0x0001f000	32 bits 16 bits 8 bits	29:07
mc6809	1 * m68b50 1 * m68b45 1 * m68b21	0xe000 0xe800 0xec00	8 bits 8 bits 8 bits	14:11
tms32020	2 * mcm6164-45 2 * m2764a-1	0x00008000 0x00000000	16 bits 16 bits	17:19
i8085a	1 *cmd6116-9 1 * m27256a-1 1 * m68b21	0x8000 0x0000 0x4000	8 bits 8 bits 8 bits	13:39

TABLE 7-9. Summary of Designs

The simulation results for the microprocessor systems shown in Table 7-9 were all verified with the component manufacturer’s data sheets and found to be correct. The simulation results were also analyzed from a system architecture perspective and found to be correct: Each device was activated only when required, the data from each device was sent over the correct data bus signals and the address signals of a device were connected in the correct sequence. The goal of this proof of concept Interface Designer was the development of an automated interface design expert system that could produce data transfer interface that assures that the components operate correctly according to the specifications provided in the component manufacturers’ data sheets. The interface design examples show that this goal has been achieved.

The designs produced by the Interface Designer are similar to that produced by a human designer. This is primarily due to the fact that the design process developed for this work attempts to mimic a human designer. For example, the human interface design example for a 68000 to 6116 interface given in Figure 3-1 in Chapter 3, is similar in many respects to the design generated by the Interface Designer shown in Figure 7-7: Both systems use a separate address decoder, bank select decoder, DTACK delay generator and they ‘combine’ the decoded address signals using an AND gate with the UDS and LDS data strobes to generate the CE signals on the 6116. Some minor differences exist in the use of the AS signal and the lack of utilization of the *type* information signals in Figure 3-1. These differences, however, will not change the basic operation of the interface and it would be difficult to decide which design is more optimal.

The completion times for the Interface Designer shown in Table 7-9 indicate that a complete interface for a simple system can be designed in 15 to 30 minutes. From experience this should be faster than an expert human designer solving the same problem. In addition, it includes the generation of machine readable VHDL code and a complete verification of component timing parameters. An expert designer may be able to draw up an interface in 15 minutes, but he may not be able to perform a thorough check of the timing parameters. Furthermore, a manual process is more prone to human errors. With new, faster computer workstations becoming available every day, the Interface Designer will be able to complete a design within minutes, giving the user the ability to experiment and try out many different configurations in a short time.

Chapter 8

Conclusions and Future Work

This chapter presents the conclusions of this work and provides an overview of the contributions of this research in the fields of microprocessor system design, expert systems and knowledge representation techniques. Further research areas of interest are also discussed.

8.1 Conclusions

This work develops an expert system that is capable of designing the data transfer interface of a customized microprocessor system. One of the most difficult aspects of automating the interface design is the existence of the many subtle variations of the interface protocols. Based on the central premise that interface design could be automated by developing a limited number of representative timing patterns to represent the signal protocols and making design decisions based on the recognition these patterns, an automated interface designer is built to design microprocessor system interfaces using commonly available components.

The overall approach of this work is to perform design based on the recognition of a standard set of timing patterns. Any signal on the component or the interface must follow one of the standard timing patterns. To perform interface design, the Interface Designer must be able to make certain assumptions about the behavior of signals through circuit elements such as wires: a human designer simply assumes that a wire delay is so small, that the timing pattern will not change from one end of a wire to the other. To give the Interface Designer the capability to use this assumption requires the development of a property of the timing patterns called small delay invariance: the type of timing pattern that a signal follows will not be changed by a small delay. All timing patterns developed for this work are small delay invariant.

There are several advantages of using this pattern matching approach for interface design and using a limited number of timing patterns for making design decisions:

- Rules can be used to capture human designer’s expertise for interconnecting signals with different timing patterns using primitive circuits. In addition the Interface Designer does not require a sub-system capable of generating the primitive circuits themselves, since the required primitive circuits can be pre-designed.

- There is a reduction in the level of detail, and hence the complexity, of the design process and the information that must be modeled and represented by the Interface Designer: The level of detail needs only be sufficient to allow the pattern matching rules to select one of the pre-designed primitive circuits.
- The timing patterns provide a powerful tool for simplifying the representation of the timing behavior of a component. Essentially, the timing patterns model only those aspects of a signal's timing that are required for interface design.
- Any component whose data transfer interface protocol can be represented by the timing patterns developed can be added to the component library database. Once in the component library database, the component can be used immediately in designs, without changes to the design rule base.
- The number of different rules required to perform the interface design is limited by the small number of different timing patterns. Using a human designer's expertise, the number of rules can be further reduced by eliminating the impossible or improbable cases.
- The system can be extended to use new timing pattern with relatively little effort. The only addition to the design rule base will normally be rules to manipulate the new timing pattern. Once the new rules are added to the Interface Designer, it will be able to generate designs with component using the new timing pattern.

The approach to interface design used in this work is shown to be valid by generating a number of microprocessor systems using a variety of different components and verifying the design using simulation tools. The designs produced are of similar complexity and speed as that of a human designer. However, there are some potential problems associated with the timing pattern based approach:

- Interface design may fail with a given set of components: the interface will be functionally correct, but some timing parameters may be violated. The timing violation is caused by the Interface Designer choosing interface primitive circuits based on the timing patterns without knowing the actual timing parameters of the finished interface. The Interface Designer will detect such problems, but currently can not redesign the interface when a timing parameter violation is found. Similar to what a human designer will do, another design is to be generated using more compatible components, instead of producing an overly complex design.
- A component may have a complex interface that uses signal timings which can not be modelled using the developed timing patterns. In such a case it may not be feasible to develop a new timing pattern due to the complexity of the protocol. This may occur with new microprocessors such as the Pentium II microprocessor, where the designer is expected to use third party interface components between any peripherals and the microprocessor. The signals between these interface chips and the microprocessor can be considered tightly coupled and will usually be connected directly. In these cases, an automated interface designer that connects the required signals directly will be sufficient to accomplish the interface design.

The Interface Designer is given the design expertise to manage the representative timing patterns in the form of rules. In essence, the system is provided with the knowledge of how to connect signals that follow certain timing patterns using a set of pre-defined primitive circuits. However, the design of a data transfer interface can not be accomplished with just these rules since they effectively only address the issue of when signals change state (i.e. their timing behavior), but not what the different state of the signals represent. Furthermore, to accomplish interface design, the Interface Designer must know the purpose and the source or destination of the information represented in the states of the signals. To address this issue, a novel representation of the signal behavior is developed that represents the data transfer protocol of a component as a series of information transfers. Each information transfer has a specific purpose in the protocol of a capability and consists of two parts: the state information indicating what the information encoded in the state of signals represents, and timing information representing when the state information is valid. Once this approach is taken, it is relatively simple to analyze the protocol of components to isolate the different types of information transferred: data, address, direction, type, size, width, request and delay information.

The advantages provided by the technique of representing the data transfer protocol as a series of information transfers proves to be quite far reaching:

- It provides a simple method to represent the purpose and function of an information transfer by assigning it a unique type. A design expert’s knowledge on how to connect signals with a specific type could then be represented as rules that recognize and connect specific information transfer types.
- Furthermore, once the Interface Designer has indicated that signals involved in an information transfer must be connected, the connection process can be accomplished without consideration of the type of information using the timing pattern matching rules and state information management rules.
- It provides a method for abstraction and information hiding. It allows the component to be represented as a hierarchial network of frames where each lower level of the hierarchy reveals more detail about a component. At a given abstraction level, corresponding rules at that specific level perform design without having to know specific details from the lower levels.

The primary disadvantage of this method is encountered when developing the design rules that must handle the state and timing information separately. When a signal enters an interface block, it is relatively simple to determine how the state must be changed and how the timing behavior must be changed, but it is difficult to develop a general method to design an interface block that can accomplish both changes. To overcome this problem, different microprocessor system designs were analyzed to see if a common method could be determined. It was found that the timing information of a signal can be changed using primitive circuits such as a Flip-Flop, followed by a change of the state information using a combinatorial circuit. This work uses the same approach, since it fits in very well with the state and timing information based information transfers.

Once the design is completed, an interface implementation technology is chosen and each of the interface primitive circuit parameters is assigned a range of values specified as an upper limit, lower limit and typical value. Each input timing parameter that must be satisfied by an interface output timing parameter is represented as a timing constraint. Each timing constraint is verified at the maximum and minimum values of the interface parameters. The approach taken for the analysis of the timing constraints was conservative. An approach based on probabilistic evaluation of timing constraints as proposed by Escalante [26] may provide a more accurate estimation of the actual behavior and performance of the interface.

The primary goal of this work is to produce real-world designs: an automated system that actually generates a data transfer interface between components. In producing such a design it was found that a method had to be developed that allows testing and implementation of the data transfer interface. The method developed generates a VHDL representation of the interface which can be simulated and synthesized using standard VHDL tools.

The organization of the component and interface models as hierarchial networks of frames and design rules that utilize information from the frame networks allows design to proceed in a top down, divide and conquer fashion, starting with the high levels, working towards the more detailed lower levels. The hierarchial interface generated using this method greatly facilitates the generation of VHDL code for the interface, since there is a direct mapping from the hierarchial interface frames to a VHDL representation.

To summarize, the major contributions of this work are:

- The development of a set of standard timing patterns to represent the timing behavior of signals involved in data transfer.
- The development of a representation of the data transfer protocols in terms of information transfers, where each information transfer is based on one of the timing patterns.
- The development of a simple and complete hierarchial frame based representation of the components.
- The development of a hierarchial frame based representation of the interface.

- The development of a set of forward chaining rules that build up the data transfer interface.
- The development of a set of primitive circuits used by the interface design rules as the basic building blocks of the components.
- The development of parallel abstraction levels for the component model, interface model and the interface design process so that independent interface design rules can carry out the design at each abstraction level.
- The development of a method to verify that the timing behavior of the interface satisfies the timing behavior of the components being connected.
- The development of a method to translate the interface frames into VHDL code to allow implementation and testing of the interface in real-world applications.
- The development of a method to automatically generate a VHDL test bench that allows simple verification of the operation of the interface.
- The implementation and testing of the Interface Designer using real-world interface design examples.

8.2 Future Work

Based on the success of the simple automated Interface Designer developed for this work we believe it is worthwhile to further develop and extend the DAME microprocessor design system. This section elaborates on how the Interface Designer's capabilities could be extended and discusses some areas of interest for future research.

One area of focus for the DAME system should be the development of an intelligent component editor. The intelligent component editor would assist the design engineer in the entry of new components. An extension of the work by Li [49] using model frames as outlined in Appendix G would allow an intelligent component editor that guides the knowledge engineer during component entry. This would remove many of the problems and errors introduced during manual component entry and it would allow the component data structures to be verified before they are entered into the component library.

The DAME system should be extended to include design optimization information in the knowledge base. This would allow microprocessor systems to be produced that are optimized according to a requirement such as lowest cost, highest speed or lowest power consumption. The knowledge representation techniques developed for this work lend themselves readily to the inclusion of such information. For example, for memory devices, information could be included that indicates lowest power consumption is achieved when chip select is negated. This information could then be utilized when trying to minimize power consumption.

The primary focus of the DAME system to this point has been the generation of the interface between components. The higher levels of the DAME system should be developed to allow a complete microprocessor system to be generated using only original specifications. When developing the higher design levels, a case history knowledge base could be integrated into the DAME system that avoids the use of incompatible components. Designs using components that have been previously found to be incompatible and requiring time consuming redesign of the interface could thus be avoided.

The integration of VHDL simulation tools directly into the DAME system to allow direct simulation of the designed interface would enhance the utility of the system. Ideally, a VHDL representation of each of the components in the component library should be developed which would allow full functional simulation of the designed interface. Furthermore, the development of a method for cross annotation of primitive circuit parameters between the Interface Designer and the VHDL (or other) synthesis tools would allow timing constraints to be re-checked after implementation using actual timing parameters.

Further research should be directed to the development of techniques for redesign if a design fails for some reason. The current system simply requires complete redesign using different components. An investigation into the feasibility of backtracking would be required. This would allow some of the already generated design to be reused, avoiding complete redesign.

Another useful area of research would be the development of a more theoretical and formalized representation of the timing templates. This may allow automatic generation of the primitive circuits used for interconnecting signals with given timing patterns. For this work, the connection of signals following given timing pattern was accomplished by specific rules. Rules were provided for all timing patterns. By adding a representation of the timing patterns as signal transition graphs as used by Escalante [27][28], it may be possible to replace these rules using a design system that will generate primitive circuits automatically. This would avoid the tedious and error prone task of manually designing and writing the rules that generate the primitive circuits.

Research should address the development of a method for representing timing links between timing templates, and/or timing links between the timing templates and a common clock, to allow representation of more complex timing relationships found on some of the newest microprocessors. This research could be further extended to the development of timing templates for other component capabilities such as interrupt and bus arbitration. The new capabilities will require completely new timing templates to be developed that allow the protocol of these capabilities to be represented using information transfers.

Further research efforts should also be directed at methods for finding an optimal set of primitive circuit propagation delays. These propagation delays could then be passed to the synthesis/layout tools for the interface as guidelines. The current system uses fixed intervals for the primitive circuit propagation delays that are dependant on the implementation technology chosen. By telling the layout/synthesis tool what the propagation delay should be, it will be more likely that the resulting interface will not violate any timing constraints, resulting in less requirement for redesign.

An extension of the DAME design system could be an useful interactive tool for educational institutions as a teaching aid for microprocessor system design. Such an expert system could systematically guide the student towards designing a complete microprocessor system. It could present the student with the components being connected, highlight the different signals that must be connected and present the student with explanations of why certain design decisions are made as the design proceeds. The system could either produce the design automatically, illustrating the different steps taken, or it could let the student make the design decisions, pointing out errors or suggesting alternative designs.

A commercial product based on the DAME interface designer should be feasible and will require further development of the current system. A commercial automated design system will most likely be only used for microprocessor systems based on simpler microprocessor and memory components. Designs involving more complex design issues such as caches or synchronous DRAM using burst data transfer will still require manual design. However, even with such restrictions, there would be a large market for an automated design system since more and more commercial products include custom microprocessor systems.

Bibliography

[1] Aylor, J. H., R. Waxman and C. Scarratt, “VHDL - Feature Description and Analysis”, *IEEE Design & Test,* Vol. 3 No. 2 pp. 17-27, April 1986.

[2] Ashenden, P. J., *The Designer’s Guide to VHDL,* Morgan Kaufmann Publishers Inc., San Francisco, California, *1996.*

[3] Baer, J., *Computer Systems Architecture,* Computer Science Press, Rockville, Maryland, 1980.

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

[5] Balph, T., *VMEbus - A Microprocessor Bus for the Future*, Motorola Semiconductor Products Inc., Phoenix Arizona, 1982.

[6] Bansal, V. K., *Design of Microprocessor Based Systems*, John Wiley & Sons, Toronto, 1985.

[7] Begg, Vivienne, *Developing Expert CAD Systems,* Anchor Press Limited, London, 1984.

[8] Bennets, R. G., *Design of Testable Logic Circuits,* Addison-Wesley Menlo Park, California, 1984.

[9] Bibbero, R. J. and David M. Stern, *Microprocessor Systems, Interfacing and Applications*, John Wiley and Sons, Toronto, 1982.

[10] Birmingham, P. W. and A. P. Gupta, “The Micon System for Computer Design”, *IEEE Micro*, Vol. 9 No. 5 pp. 61-69, October 1989.

[11] Birmingham, P. W., MICON: “A Knowledge Based Single Board Computer Designer”, *Technical Report*, Research Report No. CMUCAD-83-21, Dec 1983.

[12] Bowen, B. A. and R. J. A. Buhr, *The Logical Design of Multiple-Microprocessor Systems*, Prentice-Hall Inc., New Jersey, 1980.

[13] Breuer, M. A. and A. D. Friedman, *Diagnosis & Reliable Design of Digital Systems*, Computer Science Press Inc., Rockville, Maryland, 1976.

[14] Brozozowski, J. A. and M. Yoeli, *Digital Networks,* Prentice-Hall Englewood Cliffs, N. J., 1976.

[15] Bushnell, M. L., *Design Automation*, Academic Press Inc., New York, 1988.

[16] Carnegie Group Inc., *Knowledge Craft Manual (Version 3.2), Volume 1*, Carnegie Group Inc., Pittsburgh, PA, 1988.

[17] Carnegie Group Inc., *Knowledge Craft Manual (Version 3.2), Volume 2*, Carnegie Group Inc., Pittsburgh, PA, 1988.

[18] Clements A., Mi*croprocessor System Design,* PWS-Kent Publishing Company, Boston, MA, 1992.

[19] Conley, W., *Computer Optimization Techniques*, Petrocelli Books, New York, 1980.

[20] Comer, D. J., *Microprocessor Based System Design*, Holt, Reinhart and Winston, Toronto Ontario, 1986.

[21] Davis, R. H. Austin, I. Carlbom, B. Frawley, et al., "The Dipmeter Advisor: Interpretation of Geological Signals," *Seventh International Joint Conference on Artificial Intelligence*, Vancouver, British Columbia, Canada, 1981.

[22] Dimopoulos, N.J., K.F. Li, and E.G. Manning, "DAME: A Rule Based Designer of Microprocessor Based Systems," *Proceedings of the 2nd International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems*, vol. 1 pp. 486-492, Tullahoma, Tennessee, June 6-9, 1989.

[23] Dimopoulos, N. J., B. T. Huber, K. F. Li, D. Caughey, M. Escalante, D. Li, R. Burnett and E. G.Manning. "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.

[24] Dimopoulos, N. J., K. F. Li, E. G. Manning, B. T. Huber, M. Escalante, D. Li, D. Caughey. "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, British Columbia, pp. 388-391, May 9-10, 1991.

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

[26] Escalante, M. A., *Probabilistic Timing Verification and Timing Analysis for Synthesis of Digital Interface Controllers,* Ph.D Dissertation, Dept. of Electrical and Computer Engineering, University of Victoria, 1998.

[27] Escalante, M. A., *Bus Arbitration Modelling and Design in DAME: an Expert Microprocessor-Based-Systems Designer,* M.A.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Victoria, 1991.

[28] Escalante, M., N. J. Dimopoulos, B. T. Huber, K. F. Li., D. Li and E. G. Manning "Generic Design Rules for the Design of Microprocessor Based Systems in DAME:

Bus Arbitration Subsystems," In *Proceedings of the 1991 IEEE International Symposium on Circuit and Systems*, Singapore, pp. 3166-3169, June 11-14, 1991.

[29] Ferguson, J., *Microprocessor Systems Engineering*, Addison-Wesley Publishing Company, Don Mills, Ontario, 1985.

[30] Fletcher, W. I., *Engineering Approach to Digital Desig*n, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1980.

[31] Freedman, M.D. and L. B. Evans, *Designing Systems With Microcomputers*, Prentice-Hall Inc., New Jersey, 1983.

[32] Gilman, A. S., "VHDL - The Designer Environment", *IEEE Design & Test,* Vol. 3 No. 2 pp. 42-47, April 1986.

[33] Greenbaum, J. R. and R. Osann, "Digital Design and Analysis" in *Analysis and Design of Electronic Circuits Using PCs*, Van Nostrand Reinhold Company, New York, 1988, pp. 138-157.

[34] Hall, D. V., *Microprocessors and Digital Systems*, McGraw-Hill Book Company, Toronto, 1983.

[35] Hamacher, V. C., Zvonko G. Vranesic and Safwat G. Zaky, *Computer Organization*, Third edition, McGraw-Hill Publishing Company, Montreal, Quebec, 1990.

[36] Hansen, G. R. and E. V. Hathaway, *CAD Applications*, Delmar Publishers Inc., New York, 1986.

[37] Hayes, J. P., *Introduction to Digital Design*, Addison-Wesley, Don Mills, 1993.

[38] Huber, B., K.F. Li, N.J. Dimopoulos, D. Li, R. Burnett, E.Manning, "Modelling Signal Behavior in DAME," *Proceedings of the 1990 International Symposium on Circuits and Systems,* New Orleans, La., Vol. 2 pp. 1497-1500, Apr. 29 - May 3, 1990.

[39] Huber, B. T., K. F. Li, N. J. Dimopoulos, M. Escalante, E. G. Manning,. "Modeling Data Transfer Signals in DAME," In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, British Columbia, pp. 505-509, May 19-21, 1993.

[40] Huber, B. T., K. F. Li, N. J. Dimopoulos, E. G. Manning,. "Data Transfer Interface Design in DAME," In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, British Columbia, pp. 510-513, May 19-21, 1993.

[41] Intel, *Microprocessor and Peripheral Handbook Volume I: Microprocessors,* Intel Corporation,1988.

[42] Intel, *Microprocessor and Peripheral Handbook Volume II: Peripherals*, Intel Corporation, 1988.

[43] Intel, *Intel Component Data Catalog*, Intel Corporation, Santa Clara, CA, 1982.

[44] Intel, *Intel Memory Components*, Intel Corporation, Santa Clara, CA, 1986.

[45] Kuo, Y.,L. Kung, C. Tzeng, H. Jeng, W. Chia, “KDMS: An Expert System for Integrated Hardware/Software Design of Microprocessor-Based Systems,” *IEEE Micro*, Vol. 3 No. 2 pp. 32-35, pp. 86-92, August 1991.

[46] Lam, H. and J.O'Malley, *Fundamentals of Computer Engineering*, John Wiley & Sons, New York, 1988.

[47] Lawrence, G. E., *Designing with Microprocessors*, Science Research Associates, Toronto, Ontario, 1985.

[48] Lesa, A. and Rodnay Zaks, *Microprocessor Interfacing Techniques*, Sybex, Berkeley, California, 1978.

[49] Li, Dongni, *The DAME Editor: A User Interface for Data Acquisition in an Expert Microprocessor-based-Systems Designer*, M.A.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Victoria, 1993.

[50] Mano, M. M., *Computer Logic Design*, Prentice Hall, Englewood California, 1972.

[51] McDermott, J., “R1: A Rule-Based Configurer of Computer Systems”, *Artificial Intelligence*, No. 19 pp. 39-88, 1982.

[52] McFarland, M. C., A.C. Parker and P. Camposano, “The High-Level Synthesis of Digital Systems”, *Proceedings of the IEEE*, Vol. 78, No. 2, February 1990.

[53] McGllynn D. R., *Modern Microprocessor System Design: Sixteen-Bit and Bit-Slice Architecture*, John Wiley & Sons, Toronto, Ontario, 1980.

[54] Mostek, *Byte Wide Memory Data Book*, Mostek Inc., 1981.

[55] Motorola, *MC68000 16-/32-Bit Microprocessor*, Motorola Inc., Austin, Texas, 1985.

[56] Motorola *MC68020 32-Bit Microprocessor User’s Manual*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1984.

[57] Motorola, *Motorola Microprocessors Data Manual*, Motorola Inc., Austin, Texas, 1983.

[58] Motorola, *Motorola 8-Bit Microprocessors & Peripheral Data*, Motorola Inc., Austin, Texas, 1985.

[59] Motorola, *Motorola Memory Data*, Motorola Inc., Austin, Texas, 1988.

[60] Motorola, *VMEbus Specification Manua*l, Motorola Inc., Rev B, 1982.

[61] Napper, Simon, “Embedded System Design Plays Catch-Up”, *IEEE Computer*, Vol. 29 No. 8 pp. 118-120, August 1998.

[62] Motorola, *FAST and LS TTL Data*, Motorola Inc., Fifth Edition, Motorola Inc., 1992.

[63] Odgin, C., *Tutorial: Microcomputer System Design and Techniques*, IEEE Computer Society, 1980.

[64] Patterson, A. D. and John L. Hennessy, *Computer Architecture. A Quantitative Approach*, Morgan Kaufman Publishers Inc., San Mateo, California, 1990.

[65] Protopapas, D. A., *Microcomputer Hardware Design*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1988.

[66] Parsaye, K. and Mark M. Chingnell, *Expert Systems for Experts*, John Wiley & Sons, Inc., Toronto, 1988.

[67] RCA, *RCA CMOS Microprocessors, Memories and Peripherals*, RCA Solid State, Somerville, NJ, 1984.

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

[69] Rosenblum L. Y. and A. V. Yakovlev, “Signal Graphs: From Self-timed to Timed Ones,” in *Proceedings of the International Workshop on Timed Petri Nets*, pp. 199-207, IEEE Computer Society press, July 1985.

[70] Schnupp, P., *Productive Prolog Programming*, Prentice Hall, Englewood Cliffs, 1986.

[71] Shaw A. W., *Logic Circuit Design*, Saunders College Publishing, Toronto, 1993.

[72] Shaw, M., “Abstraction Techniques in Modern Programming Languages,” *IEEE Software*, Vol. 3x No. 2x pp. 10-26, August 1984.

[73] Shiva, S. G., *Computer Design and Architecture*, Harper Collins, New York, 1991.

[74] Shortliffe, E. H., *Computer-Based Medical Consultation: MYCIN*, Elsevier, New York, 1976.

[75] Siddall, M. F., *Expert Systems for Engineers*, Marcel Dekker, New York, 1990.

[76] Siewiorek, D. P., D. Giuse, W. P. Birmingham, “Proposal for Research on Demeter: A Design Methodology and Environment”, *Technical Report*, Carnegie-Mellon University, January 1983.

[77] Smith, M. F. and J. A. Bowen, “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.

[78] Staugaard, A. C., *6809 Microcomputer Programming & Interfacing*, Howard W. Sams & Co. Inc., Indianapolis, Indiana, 1981.

[79] Tanimoto, S., *The Elements of Artificial Intelligence, An Introduction using LISP*, Computer Science Press, Rockville, 1987.

[80] Texas Instruments, *TMS32020 User’s Guide*, Texas Instruments, USA, 1984.

[81] Thomas, D.E., and Phillip R. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, Boston, 1996.

[82] Thomson Components, *Memory Data Book*, Thomson Components, Carrollton, Texas, 1987.

[83] Vranesic, Z. G. and Safwat G. Zaky, *Microcomputer Structures*, Sounders College Publishing, Toronto, Ontario, 1989.

[84] Wagner, S. M., and W. H. Shaw Jr., “Expert Systems and Computer Aided Design: A Productive Merger”, *Proceeding of the 1988 IEEE Engineering Management Conference*, Dayton Ohio, USA, 24-26 Oct 1988, pp. 78-84.

[85] Wakerly J., *Digital Design Principles & Practice*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1994.

[86] Wiatrowski, C. A. and C. H. House, *Logic Circuits and Microcomputer Systems*, McGraw Hill Book Company, New York, 1980.

[87] Winston, P. H., *Artificial Intelligence*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1984.

[88] ZILOG, *ZILOG Component Data Book*, ZILOG Inc., 1984.

[89] XILINX, *The Programmable Logic Data Book*, XILINX Inc., San Jose, California, 1994.

Appendix A
Timing Templates for Modeling Data Transfer

This appendix presents a description of the different timing templates developed to model the data transfer protocol. The timing templates are divided into two types: non-interactive timings that have timing relations from the reference to the information signal, and interactive timings that have timing relations from the information signal to the reference signal in addition to timing relations from the reference to the information signal.

All signals shown in a signal timing template are marked with an O for output or an I for input, indicating the allowed direction of the signals with respect to the component.

The signal timings described in this section are illustrated by showing the timing links between events other than those shown in Figure 4-19. The range of the allowed values for the timing parameter of the timing links are shown using the symbol ◁——┤ or ├——▷ with respect to the reference event, as explained in Section 4.7.2, if the timing parameter is bounded by infinity on one side and an omp delay on the other. For a causal timing link, where the timing parameter value is bounded by 0 one side and + infinity on the other, the symbol ●——▷ is used.

For discussion purposes, all timing links in the signal timings presented are given names such as ‘setup time link’, ‘hold time link’, ‘response time link’, ‘acknowledge time link’ or ‘access time link’. The timing link names have slightly different meanings for the different timing templates, and must be discussed in the context of the timing template in which they are used.

A.1 Non-Interactive Timings

A.1.1 Strobe Timing

The Strobe Timing shown in Figure A-1 is used to give the timing of a signal that has what is often called non-multiplexed timing behavior (for example the address signal on a MC68000 microprocessor relative to the AS* signal, or the address signal on an 6116 relative to the CE* signal). For a Strobe Timing, the transition to a valid state of the information signal typically occurs before the initiate event (called the setup time of the information signal), while the valid state remains present until after the terminate event (called the hold time of the information signal). The information signal and the reference signals are either both outputs or both inputs. There are always-accompanied-by links between the reference events and the signal events as shown in Figure A-1.

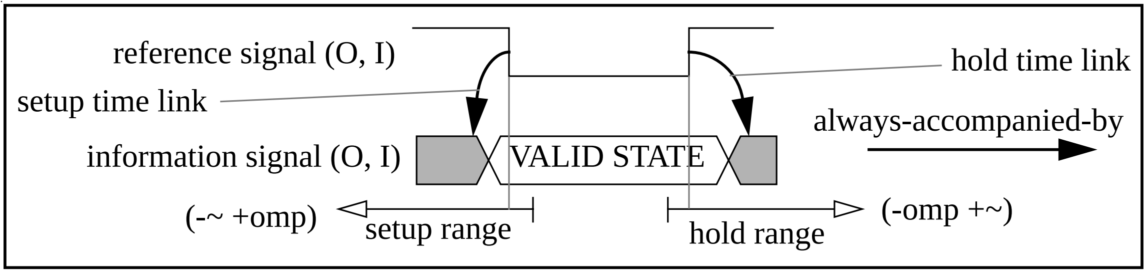


FIGURE A-1. Strobe Timing

This signal timing provides information about when the information signal is valid and stable. The first event of the information signal is an event such as a transition from a tristate/open or invalid to a valid state, while the second event is the complementary event of the first event. The setup timing parameter range is (-~ +omp), while the hold timing parameter range is (-omp +~).

A.1.2 Latch Timing

The Latch Timing is typically used to give the timing of a signal that is shared for different functions, and is commonly called a multiplexed signal timing behavior. For example, in a 8088 microprocessor, the address of a data transfer and the data itself are transferred by time multiplexing the address information and the data information onto the same signals. Time multiplexing onto the same signal means that at some point in time a signal will contain the address, while at another time the signal will contain the data.

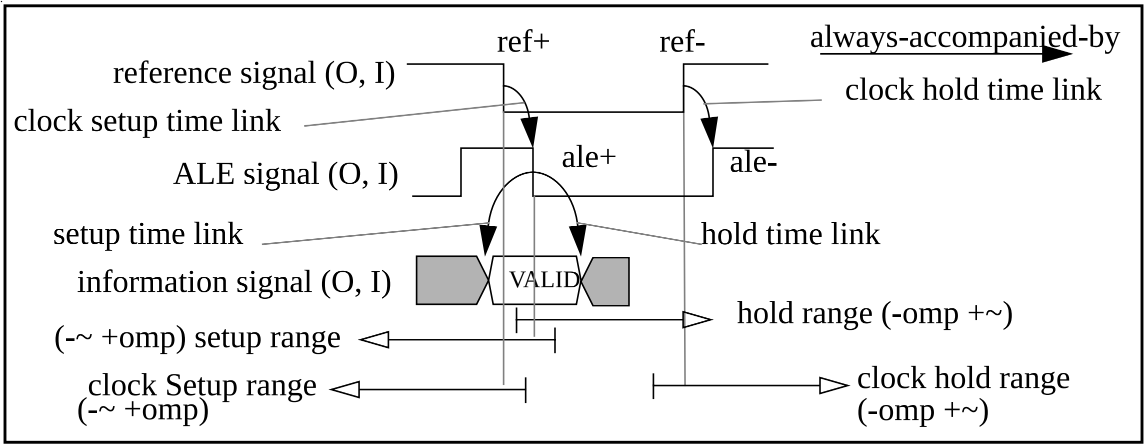


FIGURE A-2. Latch Timing

In a Latch Timing, the information signal has a timing link to a clock signal (called ALE or Address Latch Enable signal) using the setup time and hold time always-accompanied-by timing links as shown in Figure A-2. The ALE signal is related to the reference events through the clock setup time and clock hold time always-accompanied-by timing links. The information signal, the reference and the ALE signal are either all inputs or all outputs.

For a Latch Timing, the transition to a valid state of the information signal typically occurs before the ALE asserted event (setup time link), while the valid state remains present until after the ALE negated event (hold time link). The ALE asserted event typically occurs before the initiate event (clock setup time link), while the ALE negated event typically occurs after the terminate event (clock hold time link). The information signal and the reference signals are either both outputs or both inputs.

It should be noted that the ref+ event has an indirect always-accompanied-by timing relation to the information signal events through the ale+ event. The relationship is indirect since no direct timing links between the ref+ and the information signal events are given.

This signal timing provides information about when the information signal is valid and stable. The first event of the information signal is an event such as a transition from a tristate/open or invalid to a valid state, while the second event is the complementary event of the first event. The setup and clock setup timing parameter ranges are (-~ +omp), while the hold and clock hold timing parameter ranges are (-omp +~).

A.1.3 Follows Timing

The Follows Timing is used to give the timing of signal that is the response to a reference initiate event (for example the data signals on a 6116 memory device during a read). The information signal is always an output signal and the reference is always an input signal. There is a responds-with timing link between the reference events and the information signal events.

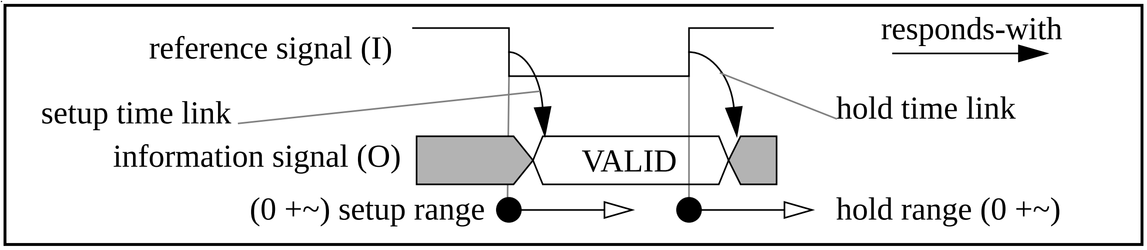


FIGURE A-3. Follows Timing

This signal timing provides information about when an information signal output is valid and stable in response to an input reference signal, implying causality. The first event of the information signal is an event such as a transition from a tristate/open or invalid to a valid state, while the second event is the complementary event of the first event. The setup and hold timing parameter ranges are (0 +~).

A.1.4 Pulse-Latch Timing

The Pulse-Latch Timing is used to give the timing of input signals that must be valid during the terminate reference event of a device relative to an input terminate reference event (for example a data signal on a 6116 memory during a read relative to the events on the `CE*` signal). In a Pulse-Latch Timing, a valid state of an information signal is present before the terminate event (setup time link), and the valid state stays until after the terminate event (hold time link). The reference signals and the information signal are both input signals. In a Pulse-Latch Timing, the information signal has timing links to a reference using the setup time and hold time always-accompanied-by timing links as shown in Figure A-4.

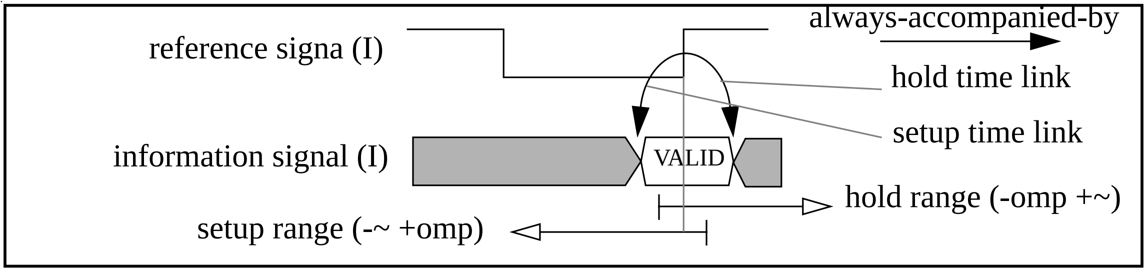


FIGURE A-4. Pulse-Latch Timing

This signal timing provides information about when an input information signal has to be valid relative to the input reference ref- event. The first event of the information signal is an event such as a transition from an open or invalid to a valid state, while the second event is the complementary event of the first event. The setup timing parameter range is (-~ +omp), while the hold timing parameter range is (-omp +~).

A.1.5 Follows-Latch Timing

The Follows-Latch Timing is used to give the timing of input signals that must be valid during the output terminate reference event of a device (for example a data signal on a MC68000 microprocessor relative to the UDS signal during a read). In a Follows-Latch Timing, a valid state of an information input signal is present before the terminate event (setup time link), and the valid state stays until after the terminate event (hold time link). The reference signal must be an output signal, while the information signal must be an

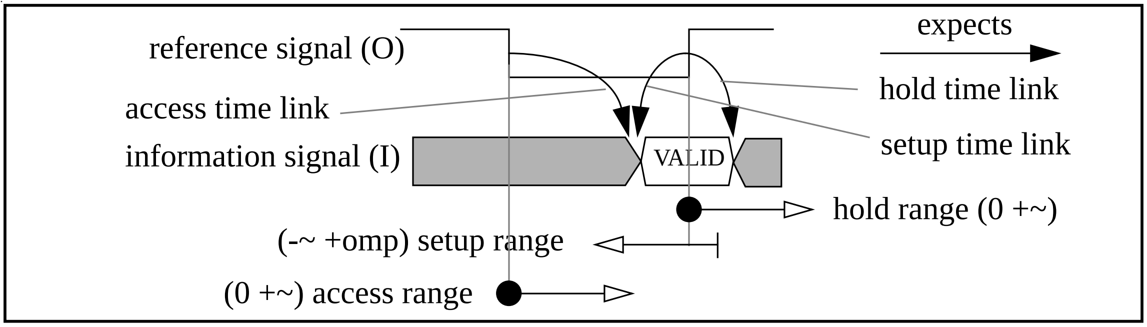


FIGURE A-5. Follows-Latch Timing

input signal. In a Follows-Latch Timing, the reference signal has expects time links to the information signal using the setup and hold time links shown in Figure A-5.

One of the distinguishing feature of the Follows-Latch Timing is the setup time of the expects timing link that is negative. The expects time link between the ref- and sig+ events is allowed to go negative for this timing since it is assumed there is a causal relationship between the ref+ event and the sig+ event (access time link).

This signal timing provides information about when an input information signal is valid and stable relative to an output reference ref- event. The first event of the information signal is an event such as a transition from an open or invalid to a valid state, while the second event is the complementary event of the first event. The access timing parameter range is (0 +~), the setup timing parameter range is (-~ +omp), while the hold timing parameter range is (0 +~).

A.1.6 Logic Timing

The Logic Timing is used to give the timing of information signals that have a timing similar to the reference. For example on a Z80 microprocessor, the MREQ* signal has a similar timing as the RD* signal. Figure A-6 illustrates the Logic Timing. The reference

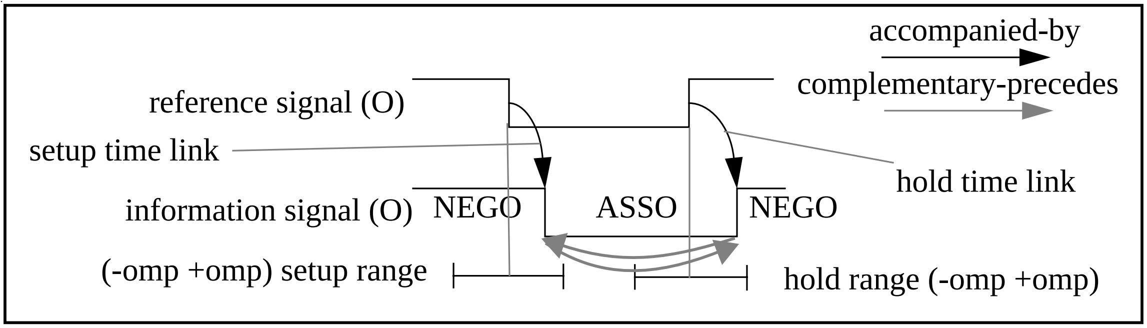


FIGURE A-6. Logic Timing

and the information signal both are outputs for a Logic Timing. The information signal events for a Logic Timing are detectable and are therefore restricted to state changes between ASSO and NEGO. The reference event to information signal event links are both accompanied-by timing links. Another notable aspect of this timing is the complementary-precedes timing link between the sig+ and sig- events. This link assures that the information signal event sequence will be glitch free. No events are allowed on the information signals other than those linked to the reference. The setup and hold timing parameters have a range of (-omp +omp).

The Logic Timing allows the description of any glitch free signal that behaves similar to the reference. By definition, a reference signal behaves similar to the reference. Therefore the Logic Timing makes it possible to give the signal timing of the reference signal itself by specifying the setup and hold timing parameters as (0).

The importance of being able to specify the timing of the reference signals is realized later in this work when every information transfer between components is given as timing information and state information. The use of the Logic Timing allows the reference information to be given in the same way as any other information, such as address information. Any techniques and heuristics developed for connecting information transfer between components can then also be applied to the reference information.

Figure A-7 shows a typical application of the Logic Timing. A MC68020 microprocessor has the DS* data strobe signal, which is used as the reference signal. The signal timing for the DS* signal is given as a Logic Timing with a setup and hold timing parameters of zero.

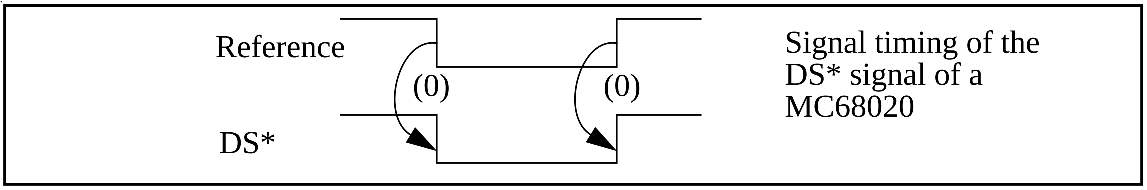


FIGURE A-7. Logic Timing Example

A.2 Interactive Timings

A.2.1 Handshake Timing

The Handshake Timing shown in Figure A-8 and Figure A-9 is an interactive timing that is used to give the timing of an information that interacts with the reference. The events on the information signals are detectable. The timing links are either responds-with or expects timing links as shown in Figure A-8 and Figure A-9. If the information signal is an output, the reference signal is an output, while if the information signal is an output, the reference signal will be an input. A typical example of the Handshake Timing is the signal relationship between the UDS*/LDS* signals and the DTACK* signal in a MC68000 microprocessor.

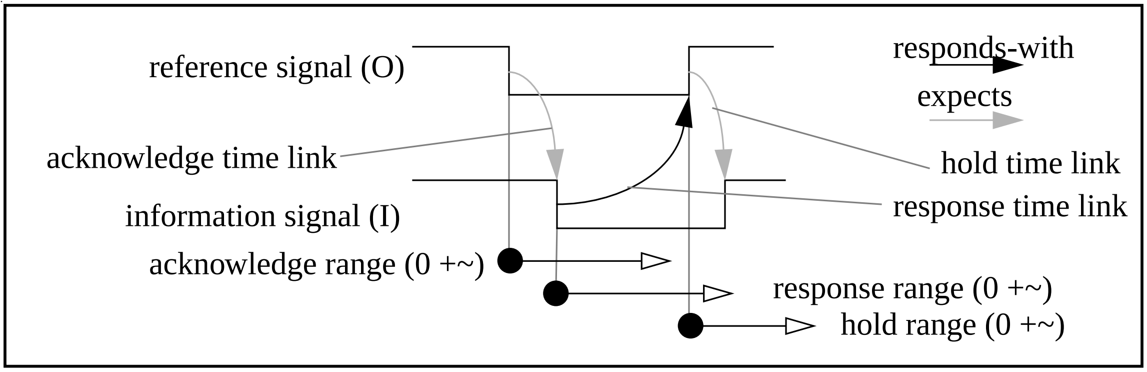


FIGURE A-8. Handshake Timing (Information Signal is Output)

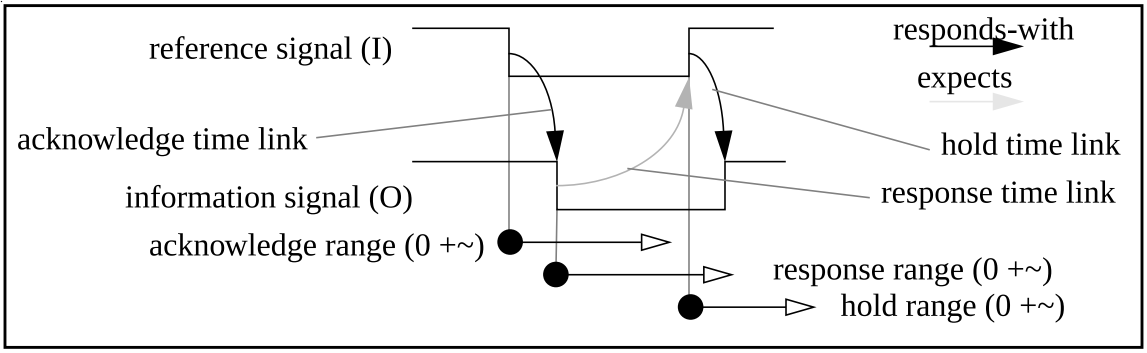


FIGURE A-9. Handshake Timing (Information Signal is Input)

The acknowledge, response and hold timing parameter range is (0 +~). The Handshake Timing presents the timing of an information signal that can be used to adjust the time from the reference initiate to terminate event by changing the acknowledge timing parameter.

A.2.2 Wait Timing

The Wait Timing shown in Figure A-10 and Figure A-11 is another interactive timing that is used to give the signal timing of an information signal that interacts with the reference. There are responds-with and expects timing links as shown in Figure A-10 and Figure A-11. There is an complementary-precedes link between the sig+ and sig- events (acknowledge time link). There is also a complementary-precedes link between the ref+ and ref- events. (minimum time link). The events on the information signals are detectable. If the reference signal is an input, the information signal is an output, while if the reference is an output, the information signal will be an input.

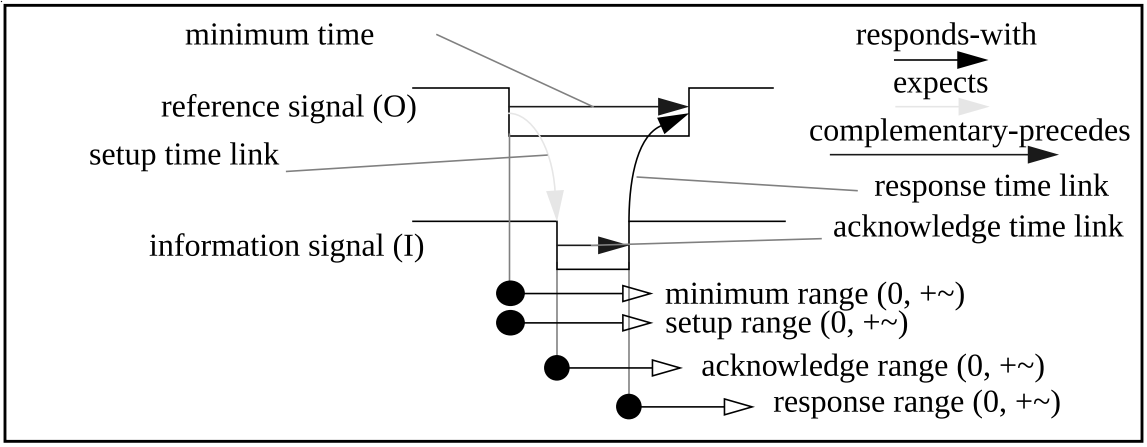


FIGURE A-10. Wait Timing (Information Signal is Output)

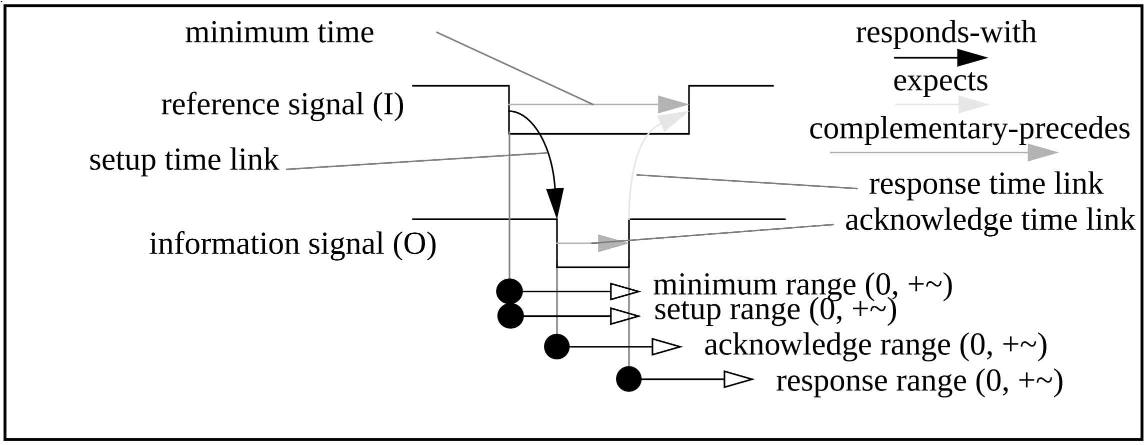


FIGURE A-11. Wait Timing (Information Signal is Input)

A typical example of the Wait Timing is the relationship between the MREQ* signal and the WAIT* signal on a Z80 microprocessor. The minimum, setup, response and the acknowledge timing parameter range is (0 +~). The acknowledge timing parameter can be used to adjust the time interval between the reference initiate to terminate events.

The Wait Timing from Figure A-10 has an interesting property: if the information signal does not have an event after the reference initiate event (i.e. the information signal stays negated), then the initiate to terminate time interval will be given by the timing parameter of the minimum time link.

A.2.3 Pulse Timing

The Pulse Timing shown in Figure A-12 is an interactive timing that is used to give the signal timing of the terminate reference event relative to the initiate reference event. The Pulse Timing is used in components where the time from the initiate to the terminate event is fixed. For example the E signal on a 68HC11 microprocessor provides a fixed time interval from the initiate to terminate reference event. The Pulse Timing has a complemetary-precedes link between the ref+ and ref- events (access time link). The access timing parameter range is (0 +~). The reference signal and the information signal are the same for this timing. The reference / information signal can be an input or an output.

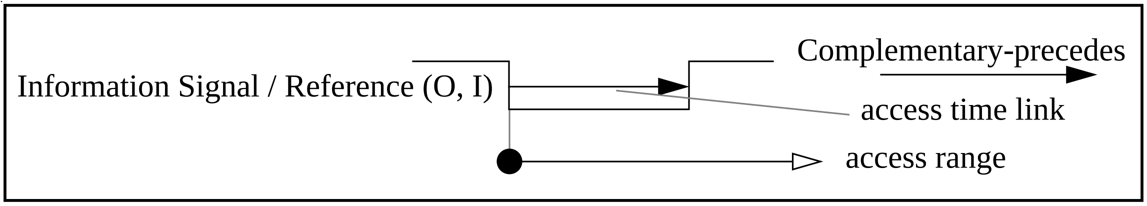


FIGURE A-12. Pulse Timing

Appendix B
The Component and Interface Frame Hierarchy

B.1 The Component Frames

A microprocessor system component, such as a MC68000, is represented by a component device frame. The component device frame will specify the component's characteristics, behavior and attributes. These characteristics include items such as the number and names of signals pins, power consumption, voltage requirements, tasks the device can perform and how the device will perform those tasks.

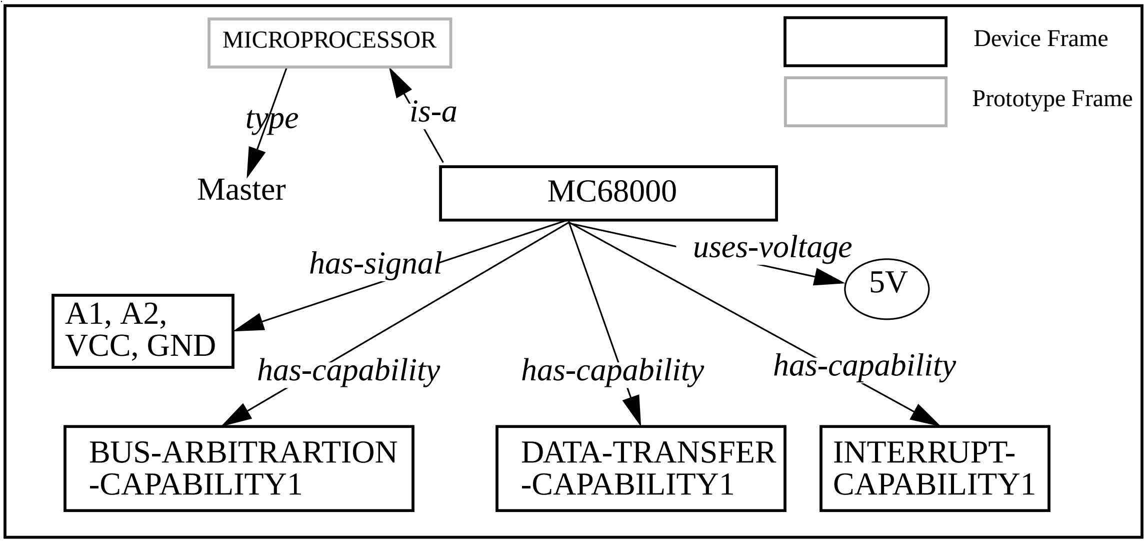


FIGURE B-1. The MC68000 Component Device Frame

Figure B-1 shows the organization of a MC68000 microprocessor device frame. The MC68000 inherits all the properties of a microprocessor (such as ^*type* 'Master') from the MICROPROCESSOR prototype frame. The MC68000 device has bus arbitration, data transfer and interrupt capabilities which are represented in the device frame with a ^*has-capability* link to the BUS-ARBITRATION-CAPABILITY1, DATA-TRANSFER-CAPABILITY1 and INTERRUPT-CAPABILITY1 device frames. MC68000 signals are represented by frames linked the MC68000 frame through the ^*has-signal* relation.

B.1.1 The Capability Device Frame

The protocol of a capability consists of a series of information transfers. Each information transfer is represented using a state timing specification device frame inside the capability device frame as shown in Figure B-2. The state timing specification frame is

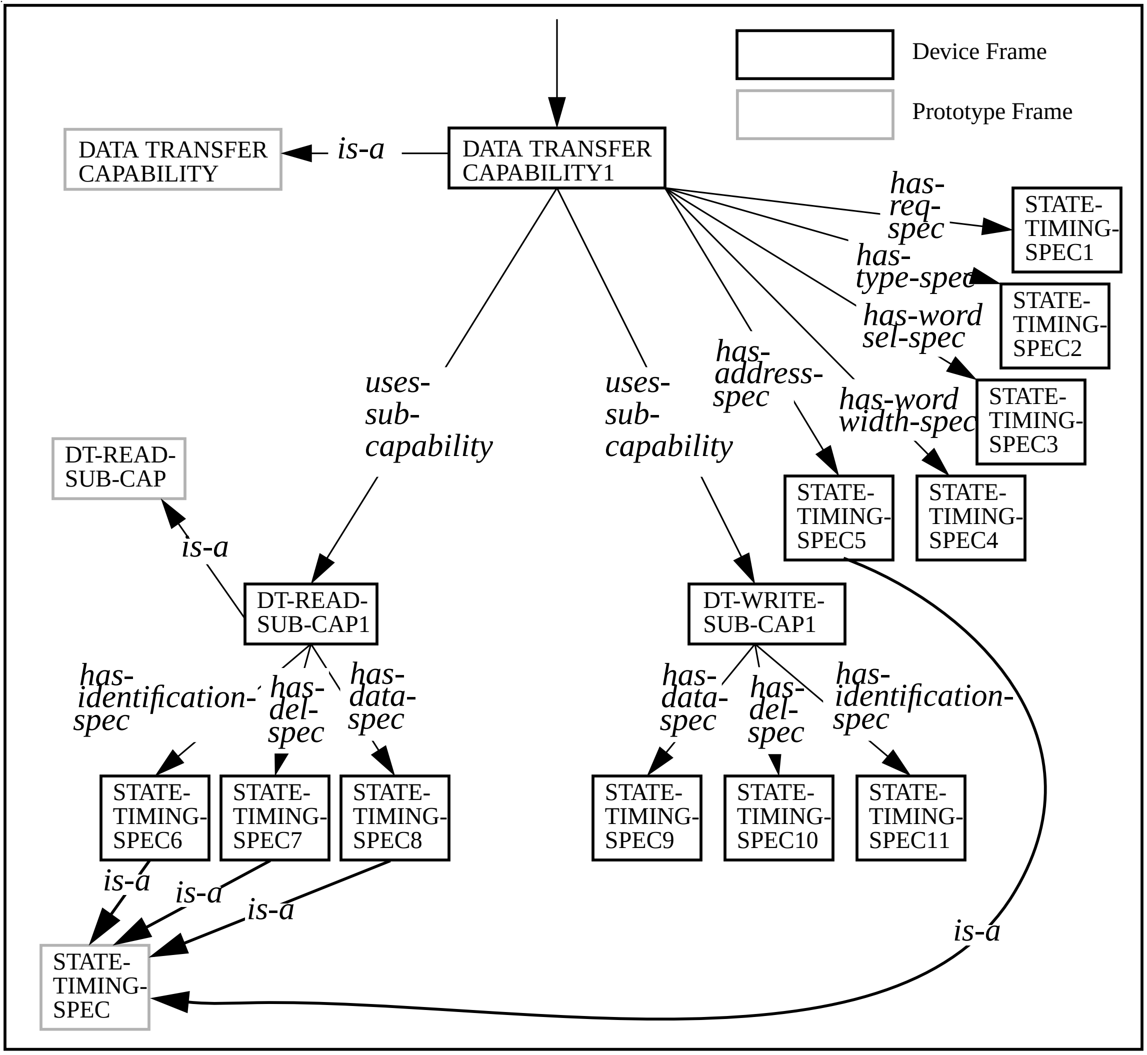


FIGURE B-2. The MC68000 Capability Device Frame

related to the capability frame using a ^*has-xxx-spec* relation, where xxx represents the class of information. For example, the *address* information is given using the ^*has-address-spec* relation and the *request* information is given using a ^*has-req-spec* relation as shown for the data transfer capability in Figure B-2. Each of the information transfer device frames, such as the STATE-TIMING-SPEC5 frame for the *address* information, is based on a STATE-TIMING-SPEC prototype frame as shown in Figure B-2.

A capability can often be classified further into sub-categories of the capability. For example, data transfer capability can be classified into read and write data transfer capabilities. Some information transfer protocols for the data transfer capability will be common to both read and write sub-capabilities, while others will be specific to the read and write sub-capabilities. The device frames developed for the capability model allow representation of sub-capabilities through the use of sub-capability frames such as DT-READ-SUB-CAP1 and DT-WRITE-SUB-CAP1 device frames shown in Figure B-2.

Any information transfers that are common to all the sub-capabilities are given in the capability frame, while any information transfer specific to a sub-capability is given in the sub-capability frame in the ^*uses-sub-capability* slot. For example in Figure B-2, the *address* information specification is common to both the read and write sub-capabilities and therefore is given in the data transfer capability, while the data information specification changes for the read and write sub-capabilities and therefore is given in the read and write sub-capabilities.

Table B-1 summarizes the relations used to give the state-timing device frames for each information class within a data transfer capability.

Information Class	**Relation**
Request	*^has-req-spec*
Address	*^has-address-spec*
Type	*^has-type-spec*
Word	*^has-word-select-spec*
Direction	*^uses-sub-capability / has-identification-spec*
Width	*^has-word-width-spec*
Delay	*^has-del-spec*
Data	*^has-data-spec*

TABLE B-1. Relations Used to give the State-Timing Frames for Data Transfer Capability

B.1.2 A Note About Choosing the Name of a Frame

The names for the device and prototype frames used to build up a component are chosen to assist the human user in identification of the function of a frame. For example DT-READ-SUB-CAP1 frame is the Data Transfer Read Sub-Capability frame in Figure B-1. Instantiation of prototype frames are usually given the name of the prototype frame with a unique number attached to the end of the name. For example in Figure B-1 the STATE-TIMING-SPEC5 frame is used to represent an instance of the state timing specification prototype frame STATE-TIMING-SPEC. Component device frames which represent actual components (such as the MC68000 frame in Figure B-1) are given a name indicating the component’s name from the manufacturer, instead of the name of the prototype frame extended by a number.

B.1.3 The State-Timing Specification Device Frame

Each information transfer is represented using a state-timing specification frame, which consists of a frame representing the timing information and a frame representing the state information of the information transfer. The example shown in Figure B-3 gives the address information specification STATE-TIMING-SPEC5 device frame which is linked to a capability frame using a ^*has-address-spec* relation. The STATE-TIMING-SPEC5 frame is broken into the timing information transfer and the state information transfer in the form of the TIMING3 device frame and the STATE3 device frame, which are related to the state-timing frame using ^*uses-timing* relation and ^*uses-state* relations respectively.

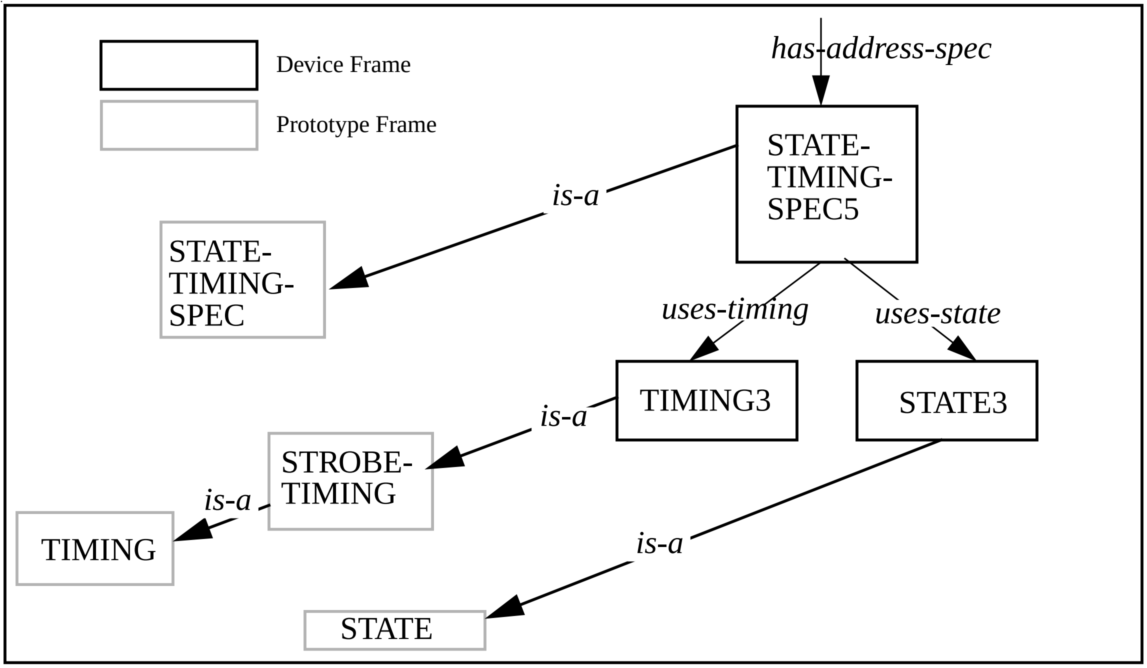
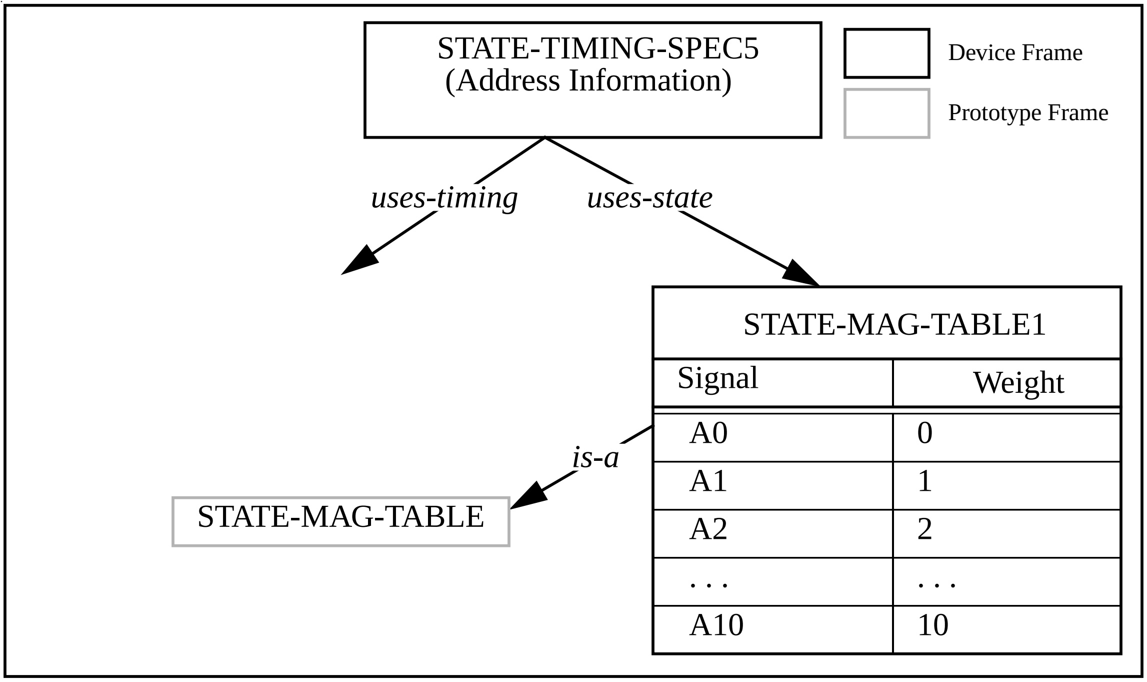


FIGURE B-3. State Timing Specification

B.1.4 The State Specification Device Frame

The purpose of a state specification device frame is to associate a set of states with some kind of abstract meaning or interpretation. If a state specification includes n binary signals, then the set of possible states will consist of 2^n states. Two different methods are used to associate a set of states to an interpretation of the set of states.

The first method applies when the set of states that must be represented by the state information is complete and regular: for n binary signals, 2^n states have meaning. *Address* state information falls into this category: 10 address signals can be used to represent 1024 states, each state indicating the address of a single memory location. A shorthand notation is used to represent this type of state information because of the large



Signal	Weight
A0	0
A1	1
A2	2
. . .	. . .
A10	10

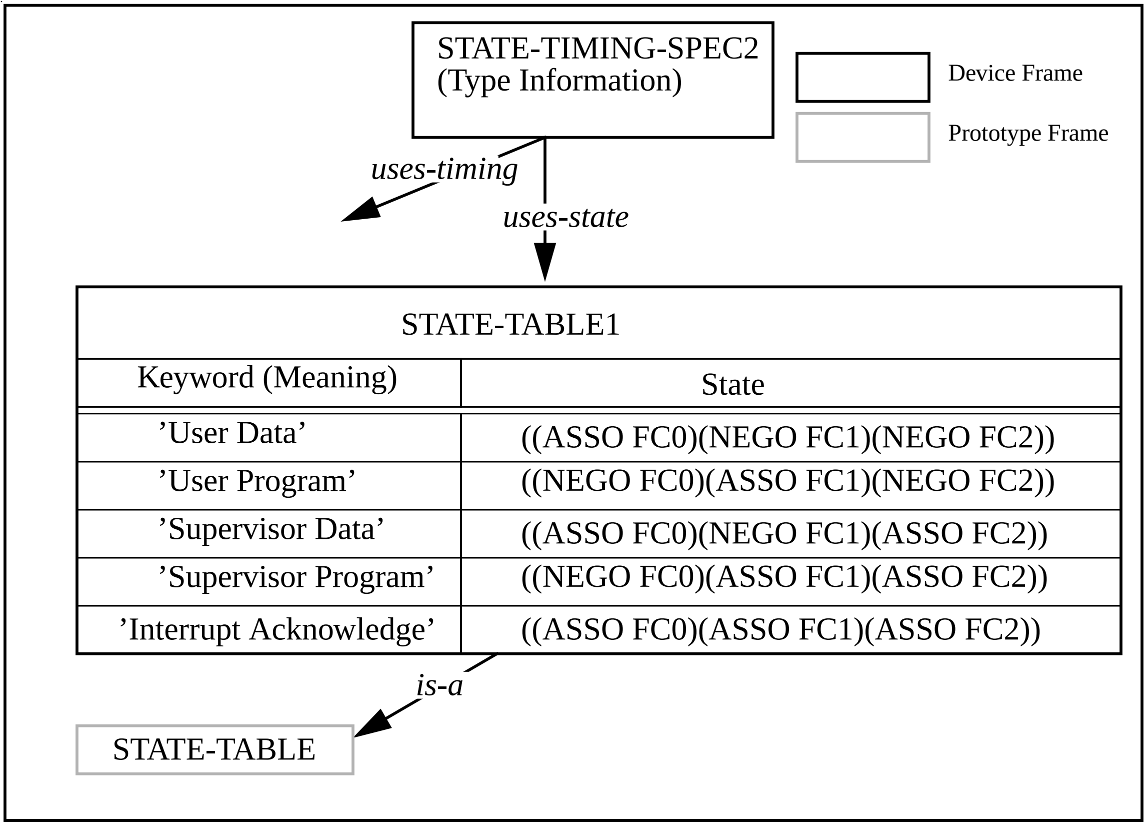
FIGURE B-4. State Information for Address Information Transfer

number of states: the binary weight of each signal is given in a magnitude state information device frame `STATE-MAG-TABLE1`, as shown in Figure B-4.

The second method for representing state information is used when there is more than one state associated with an information transfer, but the set of states is either not complete (i.e. not all binary states must be represented) or is irregular (i.e. each state in the set of states has a different meaning unrelated to the other states). An example of this kind of state information is shown in Figure B-5 for the *`type`* information of the MC68000 microprocessor: the binary `FC0`, `FC1` and `FC2` signals used to generate the states for the type information can represent eight unique states, but only five of which are used. The five that are used represent the supervisor and user program space, the supervisor and user data space, and the interrupt acknowledge data space. They are given in the state information device frame `STATE-TABLE1` as shown in Figure B-5.

B.1.5 The Timing Specification Device Frame

The purpose of the timing specification device frame is to provide the timing behavior of the signals used in the state specification device frame. A timing specification device frame represents the timings of signals as discussed in Section 4.7.4. The prototype of a



STATE-TABLE1	
Keyword (Meaning)	State
'User Data'	((ASSO FC0)(NEGO FC1)(NEGO FC2))
'User Program'	((NEGO FC0)(ASSO FC1)(NEGO FC2))
'Supervisor Data'	((ASSO FC0)(NEGO FC1)(ASSO FC2))
'Supervisor Program'	((NEGO FC0)(ASSO FC1)(ASSO FC2))
'Interrupt Acknowledge'	((ASSO FC0)(ASSO FC1)(ASSO FC2))

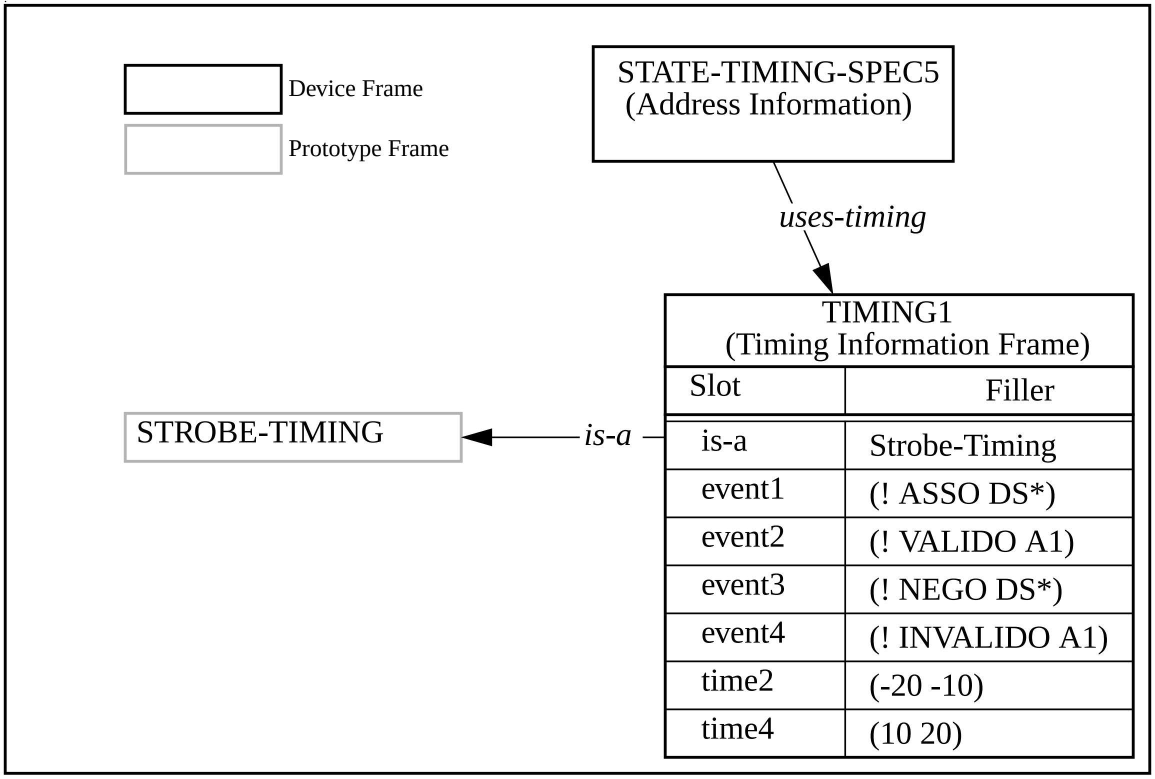
FIGURE B-5. State Information for MC68000 Type Information Transfer

timing device frame corresponds to the timing template on which the timing device frame is based. For example in Figure B-6, the `TIMING1` device frame is based on the `STROBE-TIMING` prototype.

The events and timing parameters found in the timing device frame are given in the slot and filler format as shown in Figure B-6. The slot name is the name of the event or timing parameter, while the filler gives the value assigned to the event or timing parameter. In this timing, event1 and event3 represent the reference events while event2 and event4 represent information signal events as shown in Figure B-7.

B.1.6 The Signal Device Frame

A component has electrical signal wires over which information is transferred or which supply power and ground reference voltages. The electrical connection points are called signals and the information about these signals is stored in a signal device frame, which is linked to the component device frame using a ^*has-signal* relation. Each individual signal will have electrical characteristics such as voltage levels and drive capability, a device pin number and a polarity associated with it. This information will be given in slots containing the appropriate information as shown in Figure B-8.



Slot	Filler
is-a	Strobe-Timing
event1	(! ASSO DS*)
event2	(! VALIDO A1)
event3	(! NEGO DS*)
event4	(! INVALIDO A1)
time2	(-20 -10)
time4	(10 20)

FIGURE B-6. Example Strobe Timing Information Frame

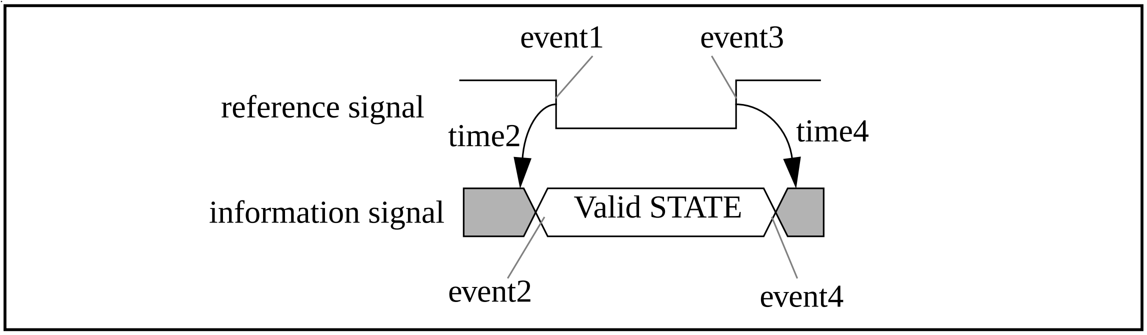


FIGURE B-7. Event Names for Strobe Timing

B.1.7 Overview of the Component Organization

A component is assembled by building up a hierarchical set of device frames which are created by instantiating prototype frames. Figure B-9 gives an overview of how the component model is organized by presenting some prototype, device and instance frames for a MC68000 microprocessor. The `MC68000` frame inherits the type ‘Master’ from the `MICROPROCESSOR` prototype frame. The data transfer capability is represented by the `DT-CAP-MICRO11` device frame which is based on the `DT-CAP-MICRO` prototype frame. The data transfer capability in turn has a read sub-capability called `DT-RD-CAP33`. The sub-capability has delay specification `STATE-TIMING5` which consists of

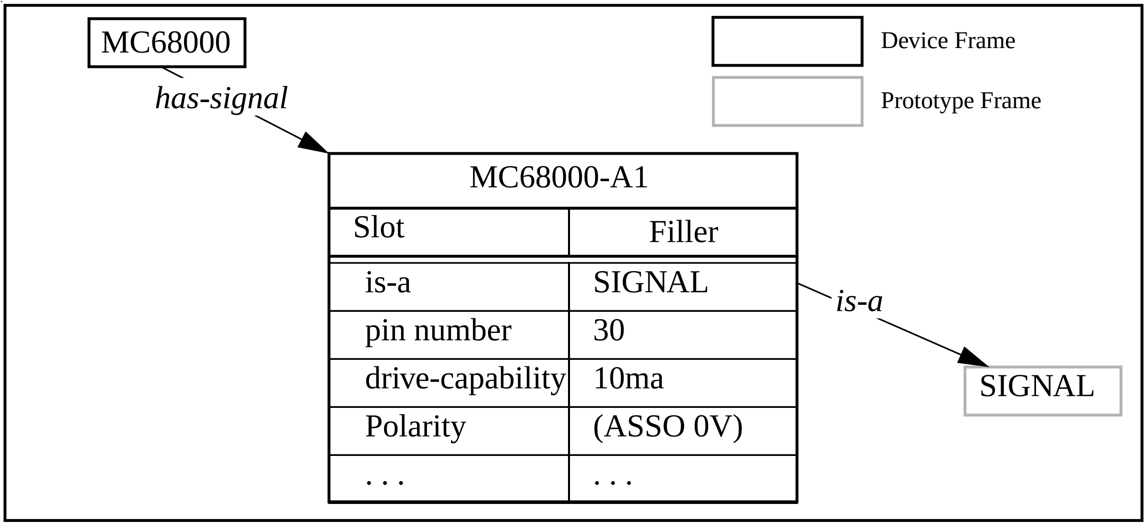


FIGURE B-8. Example Signal Frame

signal timing `TIMING14`. `TIMING14` is based on the `HANDSHAKE-TIMING` prototype frame.

The instance of a component such as `U1`, are created during the system design phase. As the system is designed, decisions are made about what components should be in the system, and the appropriate instances are created. Device frames such as the `MC68000` frame are created when components are entered into the component library data base, which is independent of the system design phase. Prototype frames are created when the rules are written that will accomplish the interface design. The prototype frames must be flexible and universal enough to be used as building blocks for any components that will be created and entered into the component library. The developer of the prototype frames must assure that any frames instantiated from the prototype contain all the relevant information required for interface design.

B.1.8 Examples of Component Frame Hierarchy

This section presents a more detailed overview of the frame hierarchy for the MC68000 microprocessor (Figure B-10) and the MK6116 static RAM (Figure B-11). It should be noted that these diagrams only show a partial set of the device frames as the full set is too complicated to show in a single figure. This is the hierarchy that was used for implementing the frames to represent components in Knowledge Craft.

B.1.9 Examples of Component Frames

This section presents some examples device and prototype frames as they were implemented in Knowledge Craft in the DAME Interface Designer.

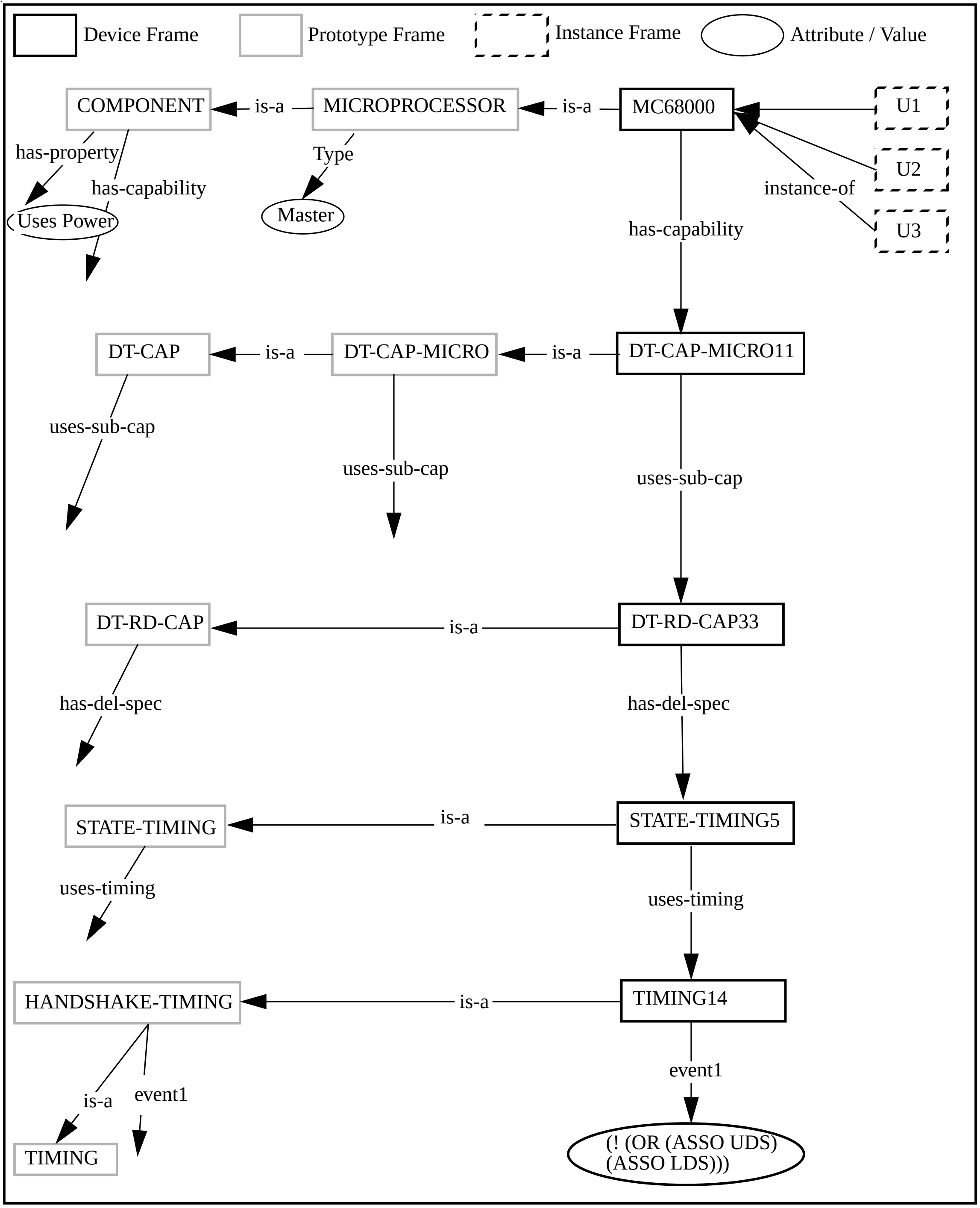


FIGURE B-9. Prototype, Device and Instance Hierarchy

B.1.9.1 Example of a Timing Information Frame

Table B-2 gives the frame representing the timing of the address signals for the MC68000 microprocessor. The address signal timing are based on a Strobe Timing. The reference signals, which consists of the `MC68000-LDS` and `MC68000-UDS` signals,

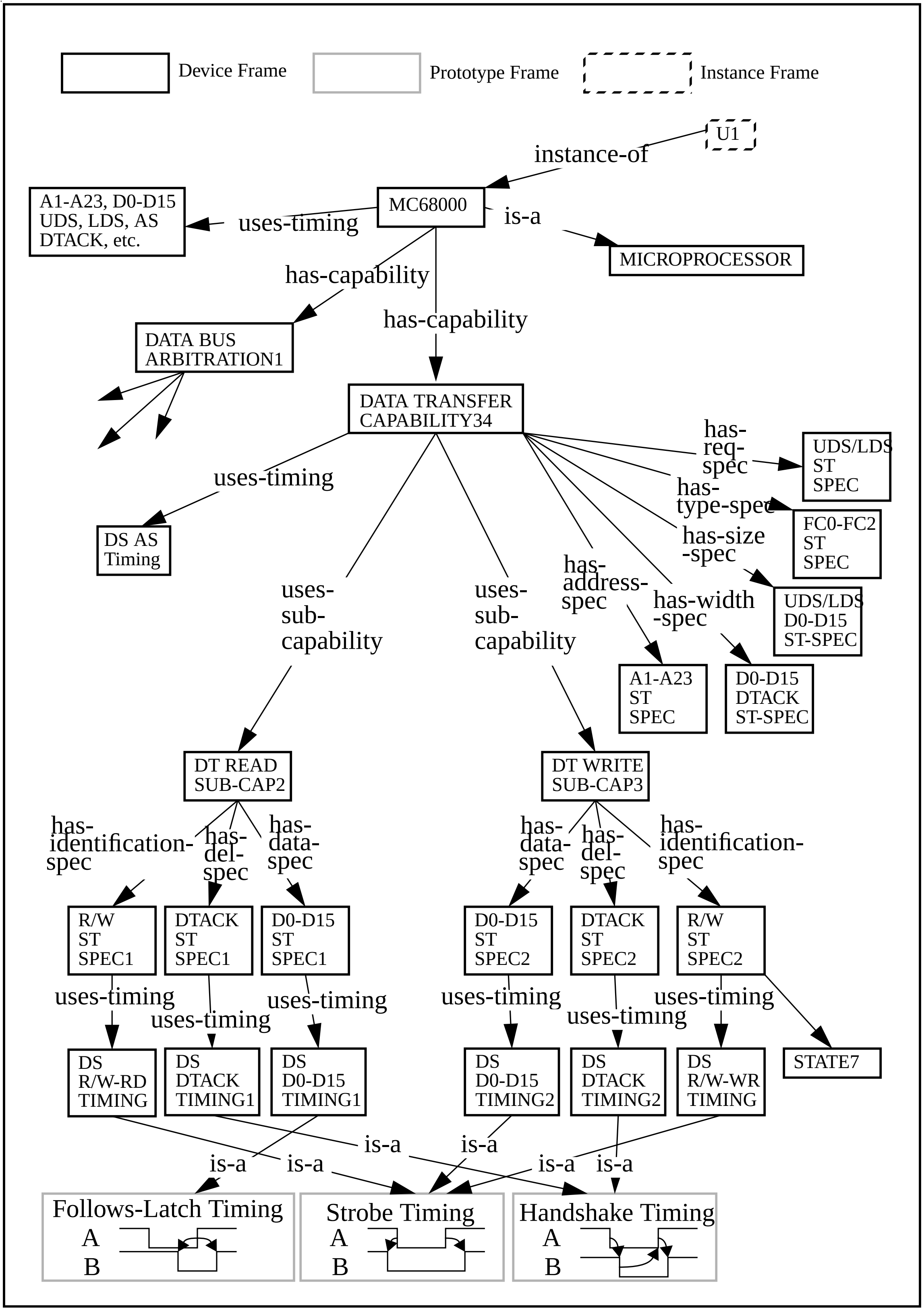


FIGURE B-10. Component Hierarchy for MC68000

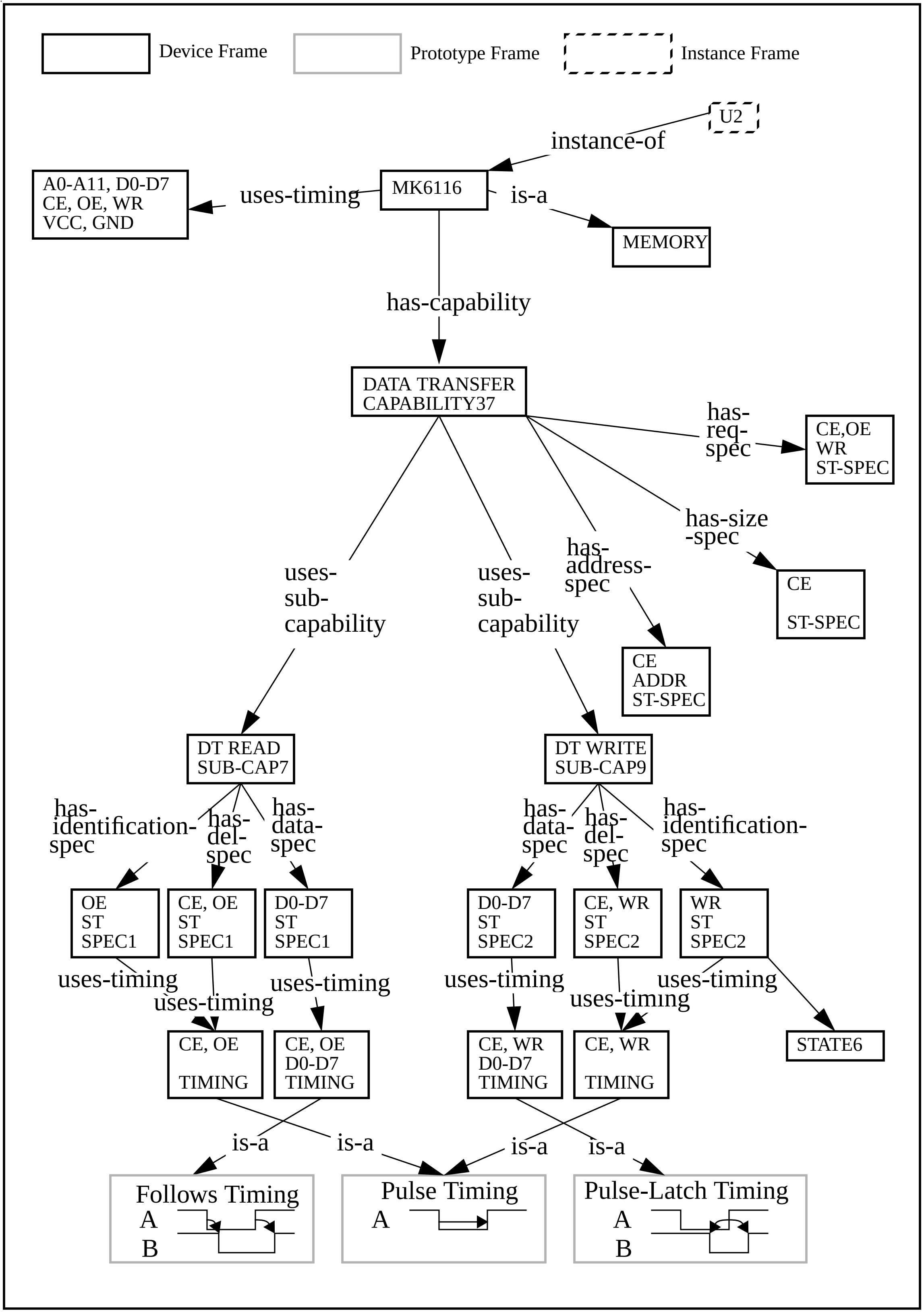


FIGURE B-11. Component Hierarchy for MK6116

are given in the ^*signal1* slot. The information signals consist of the MC68000-ADDRESS-BUS are given in the ^*signal2* slot. The reference events are given in the ^*event1* and ^*event3* slots, while the information signal events are given in the ^*event2* and ^*event4* slots. The timing parameters for the setup and hold times are given in the ^*time2* and ^*time4* slots. The ^*time2* timing parameter is associated with the ^*event1* -> ^event2 relation, while the ^*time4* timing parameter is associated with the ^*event3* -> ^*event4* relation. The events and timing parameters are represented using <*event expressions*> and <*time*> notation as developed in the Section 4.3 and Section 4.5. Figure B-12 gives a graphical representation of the timing shown in Table B-2.

```
(defschema MC68000-ADDRESS-TIMING
   (is-a STROBE-TIMING)
   (signal1        MC68000-UDS MC68000-LDS)
   (signal2        MC68000-ADDRESS-BUS)
   (event1        (! (OR ( ASSO MC68000-UDS) ( ASSO MC68000-LDS)))
   (event3         (! (NOT ((OR( ASSO MC68000-UDS) ( ASSO MC68000-LDS))))
   (event2         (! VALIDO MC68000-ADDRESS-BUS))
   (event4          (! INVALIDO MC68000-ADDRESS-BUS))
   (time2           (-~ -10))
   (time4           (10 +~)))
```

TABLE B-2. Example Frame for MC68000 Address Timing Information Frame

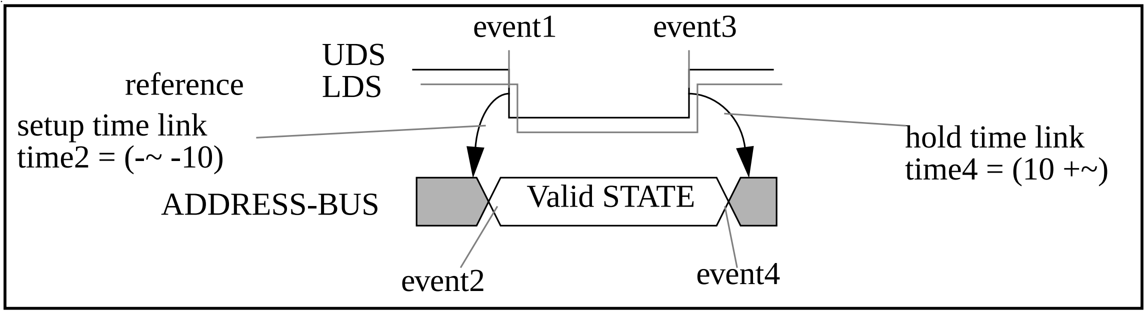


FIGURE B-12. Strobe Timing for MC68000 Address Signals

The MC68000-ADDRESS-TIMING device frame is based on the Strobe Timing prototype frame shown in Table B-3. The timing links between events and their allowed timing parameter range are stored in the ^*has-timing-relations* slot of the timing template.

```
(defschema STROBE-TIMING
  (is-a TIMING)
  (has-model STROBE-TIMING-MODEL)
  (has-timing-relations
    (event1 COMPLEMENTARY-PRECEDES event3 @ (0 +~))
    (event3 COMPLEMENTARY-PRECEDES event1 @ (0 +~))
    (event2 COMPLEMENTARY-PRECEDES event4 @ (0 +~))
    (event4 EVENTUALLY-PRECEDES event2 @ (0 +~))
    (event1 ALWAYS-ACCOMPANIED-BY event2 @ time2 (-~ +OMP))
    (event3 ALWAYS-ACCOMPANIED-BY event4 @ time4 (-OMP +~)))
  (signal1)
  (signal2)
  (event1)
  (event3)
  (event2)
  (event4)
  (time2)
  (time4))
```

TABLE B-3. Frame for Strobe Timing

For the Strobe Timing the allowed range for the setup timing parameter ^*time2* is given as (-~ +OMP) which means a range from negative infinity to a positive omp delay, while the hold timing parameter ^*time4* is given as (-OMP +~) which means a range from negative omp delay to positive infinity.

B.1.9.2 Example of a State Information Frame

A state information frame associates a set of states with a meaning or interpretation. For this work, the meaning of a state will be given as a key word, while the state itself will be given using the signal-state notation developed in Section 4.3. If the state involves n binary signals, a maximum of 2^n possible states exist.

To illustrate the concept of a state and its meaning, this section gives the type state information frame of the MC68000 microprocessor (Table B-4). There are three signals associated with the type information: `FC0`, `FC1` and `FC2`. These are binary signals that can only be asserted and negated. The three signals can be used to indicate access to five different type spaces: the user data space, the supervisor data space, the user program space, the supervisor program space and the interrupt acknowledge space. Each of the different type spaces is assigned a key word: *USER-DATA*, *SUP-DATA*, *USER-PROGRAM*, *SUP-PROGRAM* and *INT-ACK* which are listed in the ^*selects* slot. Each key word in Table B-4 has the appropriate state of the `FC0`, `FC1` and `FC2` signals associated with it in the ^*access-table* slot. The access-table can be viewed as a dictionary that associates a keyword with a signal state. The signal states are given using the notation for the <*state expression*> as given in Section 4.3.

```
(defschema MC68000-TYPE-STATE
 (is-a STATE-TABLE)
 (condition-signals   MC68000-FC0 MC68000-FC1 MC68000-FC2)
 (selects USER-DATA USER-PROGRAM SUP-DATA SUP-PROGRAM INT-ACK)
 (access-table
  (USER-DATA (AND (NEGO MC68000-FC2) (NEGO MC68000-FC1) (ASSO MC68000-FC0)))
  (USER-PROGRAM(AND (NEGO MC68000-FC2) (ASSO MC68000-FC1) (NEGO MC68000-FC0)))
  (SUP-DATA    (AND (ASSO MC68000-FC2) (NEGO MC68000-FC1) (ASSO MC68000-FC0)))
  (SUP-PROGRAM (AND (ASSO MC68000-FC2) (ASSO MC68000-FC1) (NEGO MC68000-FC0)))
  (INT-ACK     (AND (ASSO MC68000-FC2) (ASSO MC68000-FC1) (ASSO MC68000-FC0)))))
```

TABLE B-4. Example Frame for the MC68000 Type State Information

A state information frame as shown in Table B-4 will be utilized during interface design for determining the appropriate states of signals for specific conditions. For example, if a system is being designed with a memory bank in the supervisor data space, the *SUP-DATA* key yields the state expression:

(AND (ASSO MC68000-FC2) (NEGO MC68000-FC1) (ASSO MC68000-FC0)) (EQ 8-1)

This state expression can then be used to design a combinatorial decoder that can generate a signal that is active whenever the MC68000 accesses the supervisor data space.

B.2 The Interface Frames

This section gives some simple example frames for the interface and their ISBs. The frames shown are simplified for illustration purposes, and only important overall aspects are discussed.

The organization of the frames used to build the interface follows the hierarchy developed in Chapter 5. An IB frame is made up of more detailed ISB frames, which in turn are made up of ISBP frames. At each level of the frame hierarchy more detail is revealed about the interface. Each ISB will have a specific purpose attached to it. The organization for the IB frames can be seen in Figure B-13. A single IB frame, `IB-1,` rep-

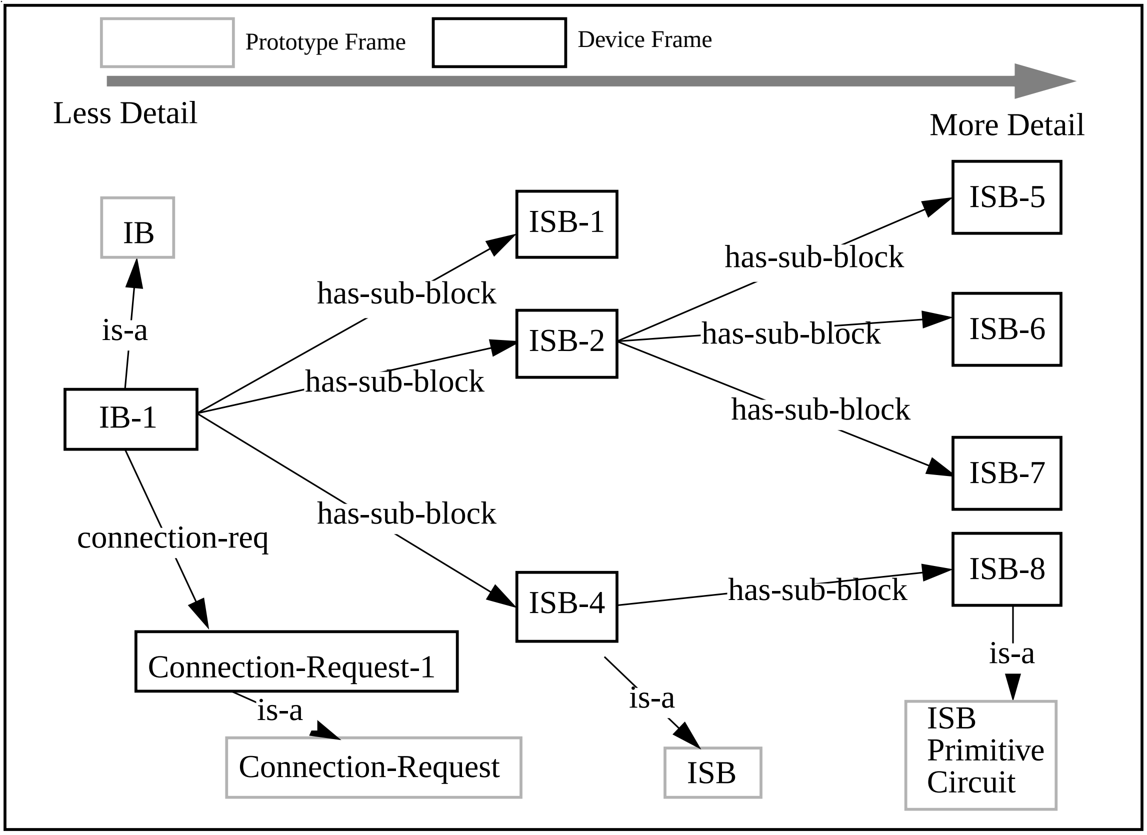


FIGURE B-13. Interface Block Organization

resents the interface designed for a specific connection request `Connection-Request-1`. A *connection request* is a simple frame that instructs the Interface Designer to initiate the interface design process. The IB frame can contain any number of ISB frames linked to the parent frame with a ^*has-sub-block* relation.

The frames that are used to build up the interface are called the interface device frames. IB and ISB device frames are created during the interface design process by instantiating the prototype IB and ISB frames. An interface device frame is related to a prototype frame through the ^*is-a* relation. For example, the `IB-1` device frame in Figure B-13 is created by instantiating the `IB` prototype frame. Similarly the connection request device frames are created during the interface design process by instantiating the `Connection-Request` prototype frame.

B.2.1 Frame Representation of the Interface Block

A typical frame representing an IB is given in Table B-5. This frame is presented to show the general organization of the IB frames. The ^*has-sub-block* slot will normally contain the frames making up the next more detailed level of the interface (`ISB_5` and

```
(defschema IB_1_RW_CONNECT
   (has-sub-block ISB_5 ISB_17)
   (has-internal-signal INT_SIG_1)
   (purpose DATA_TRANSFER CONNECT)
   (function DATA_CONN)
   (needs-function ADD_CONN)
   (component1 U1)
   (component2 U2 U3 U4 U5)
   (device1 MC68000)
   (device2 MK6116)
   (connection-req CONNECTION_REQUEST_1))
```

TABLE B-5. Interface Block Frame

ISB_17 in Table B-5). The ^*has-internal-signal* slot is used to store any signal internal to the IB after they are created during the design process (INT_SIG_1 in Table B-5). The ^*function* / ^*needs-function* slots are used to control the design of different aspects of the interface. For example, the ADD_CONN keyword in the ^*needs-function* slot indicates that the address must still be connected for this interface, while the DATA_CONN keyword in the ^*function* slot indicates that the data signals have been connected. Once the address signals are connected, the ADD_CONN key word will be moved from the ^*needs-function* to ^*function* slot. The ^*deviceX* slots indicate the components being connected (such as MK6116 and MC68000), while the ^*componentX* slots indicate the instances of the components being connected (such as U1, U2, U3 etc.). A link to the connection request that was used to create the IB is provided in the ^*connection-req* slot.

A VHDL representation of the frame from Table B-5 is shown in Table B-6. The VHDL representation will be generated by the Interface Designer once the IB and its ISBs have been designed completely. The Interface Designer will produce a structural architecture of the IB frame in terms of its ISBs. For example, the architecture of the IB_1_RW_CONNECT VHDL entity is given by instantiating the ISB_17 entity as

```
entity IB_1_RW_CONNECT is
        port ( MC68000_UDS : IN std_logic, etc....);
end IB_1_RW_CONNECT;

architecture ONLY of IB_1_RW_CONNECT is
   signal INT_SIG_1 : std_logic;
begin
   PART_1 : ISB_17
      port map (
             INT_SIG_1 => INT_SIG_1,
             MC68000_UDS => MC68000_UDS );
   PART_2 : etc...
end ONLY;
```

TABLE B-6. VHDL Representation of Example Interface Block Frame

PART_1. It should be noted that the VHDL frame above is highly simplified for illustration purpose.

B.2.2 Frame Representation of an ISBP

A typical frame representing a Combinatorial ISBP is shown in Table B-7. The ^*purpose* slot is provided to indicate the purpose of this specific ISB. This slot is currently used primarily for debugging purposes. A ^*hardware-function* slot indicates that this is a Combinatorial ISBP, with the propagation delay parameter given in the ^*parameters* slot. For this Combinatorial ISBP the propagation delay (6 12 9) indicates a minimum delay of 6 ns, a maximum delay of 12 ns and a typical delay of 9 ns. The ^*input* slot contains the input state expression which is the combinatorial equation that maps the inputs of the ISB to the outputs. Slots are provided for the ISB input and output signals and timing. The ^*output-state* slot is provided to indicate the required output state when the input state expression is true. The required information for filling in the combinatorial ISBs is found either by analyzing the different information state specification of the components being connected, and/or by analyzing the overall architecture of the complete design.

```
(defschema ISB_4_REQ_INT
   (instance INTERFACE_SUB_BLOCK
   (has-sub-block ISB_8 ISB_9)
   (purpose INTERNAL REQUEST GENERATE)
   (function REQUEST_IN)
   (needs-function))
   (hardware-function COMBINATORIAL)
   (parameters (pdelay (8 12 9)))
   (input-signals MC68000_LDS MC68000_UDS)
   (input-timings MC68000_UDS/LDS_TIMING)
   (input (OR (ASSO MC68000_UDS)(ASSO MC68000_LDS)))
   (output-timing REQ_INT_TIMING_1)
   (output-state (ASSO REQ_INT_SIGNAL))
   (output-signals REQ_INT_SIGNAL))
```

TABLE B-7. Combinatorial ISBP

The example in Table B-7 gives an ISBP that generates the internal request signal REQ_INT_SIGNAL. This Combinatorial ISBP has two inputs, MC68000_LDS and MC68000_UDS, and implements the state expressions: (OR (ASSO MC68000_UDS) (ASSO MC68000_LDS)). The output state slot indicates that the REQ_INT_SIGNAL is asserted whenever the state expression is true. A schematic representation of the ISBP frame of Table B-7 is shown in Figure B-14. A VHDL representation of the frame from Table B-7 is shown in Table B-8. The VHDL representation will be generated by the Interface Designer, once the ISBP frame has been completed, by systematically mapping the contents of frame slots to the VHDL entity and architecture. For example, the ^*input-sig-*

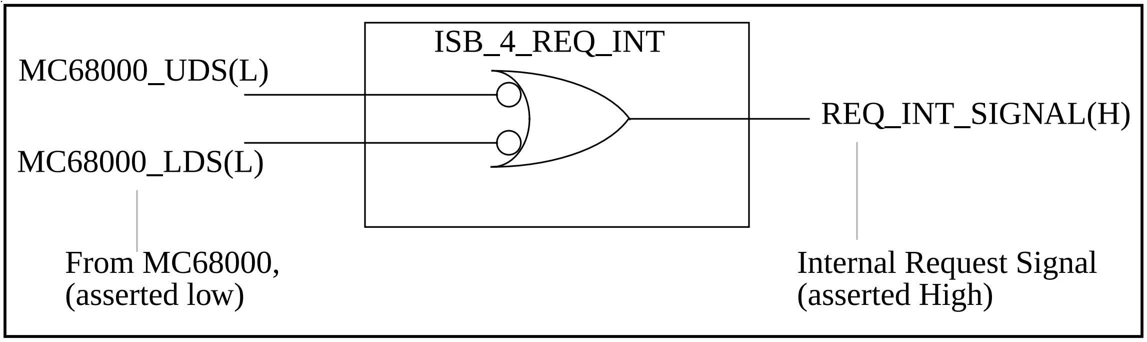


FIGURE B-14. Schematic Representation of Example ISBP Frame

```
entity ISB_4_REQ_INT is
        generic (TPD : time := 9 ns);
        port (MC68000_UDS_U1 : IN std_logic;
              MC68000_LDS_U1 : IN std_logic;
              REQ_INT_SIGNAL : OUT std_logic);
end ISB_4_REQ_INT;

architecture ONLY of ISB_4_REQ_INT is
begin
     REQ_INT_SIGNAL <=
         (not MC68000_UDS)or (not MC68000_LDS)after TPD;
end ONLY;
```

TABLE B-8. VHDL Representation of Example ISBP Frame

nals and ^*output-signals* are mapped to input ports and output ports of the VHDL entity. While the ^*input* state equation is mapped to a concurrent statement in the architecture body.

Appendix C
VHDL Code for ISBPs

C.1 Package Declaration for ISBPs

```
LIBRARY damelib;
USE ieee.std_logic_1164.all;
PACKAGE primitive IS
    CONSTANT time_prop_delay : TIME := 3 ns;
    CONSTANT time_en_delay : TIME := 2 ns;
    CONSTANT time_clock_delay : TIME := 2 ns;
    CONSTANT time_pure_delay : TIME := 55 ns;
    COMPONENT and2p
        GENERIC (tpd : TIME := time_prop_delay);
        PORT (in1, in2 : IN std_logic;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT or2p
        GENERIC (tpd : TIME := time_prop_delay);
        PORT (in1, in2 : IN std_logic;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT xor2p
        GENERIC (tpd : TIME := time_prop_delay);
        PORT (in1, in2 : IN std_logic;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT invp
        GENERIC (tpd : TIME := time_prop_delay);
        PORT (in1 : IN std_logic;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT dlatchp
        GENERIC (tpd : TIME := time_prop_delay;
                 tpd_en : TIME := time_en_delay);
        PORT (in1, latch_en : IN std_logic;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT edge_dffp
        GENERIC (tpd_clock, tpd_res : TIME := time_prop_delay);
        PORT (in1, clk, clr : IN std_logic;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT pure_delayp
        GENERIC (tpd : TIME := time_pure_delay);
        PORT (in1 : IN std_logic;
              sys_reset : IN std_logic;
              sys_clock : IN std_logic;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT leading_edge_delayp
        GENERIC (tpd_edge : TIME := time_pure_delay;
                 tpd : TIME := time_prop_delay);
        PORT (in1 : IN std_logic;
              sys_reset : IN std_logic := ‘0’;
              sys_clock : IN std_logic := ‘0’;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT trailing_edge_delayp
        GENERIC (tpd_edge : TIME := time_pure_delay;
                 tpd : TIME := time_prop_delay);
        PORT (in1 : IN std_logic;
              sys_reset : IN std_logic := ‘0’;
              sys_clock : IN std_logic := ‘0’;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT tri_state_bufferp
        GENERIC (tpd : TIME := time_prop_delay;
                 tpd_tri : TIME := time_en_delay);
        PORT (in1, tri_out : IN std_logic;
              out1 : OUT std_logic);
    END COMPONENT;
    COMPONENT oc_bufferp
        GENERIC (tpd : TIME := time_prop_delay);
        PORT (in1 : IN std_logic;
              out1 : OUT std_logic);
    END COMPONENT;
END primitive;
```

C.2 Entity and Architecture Declaration for ISBPs

C.2.1 2 Input AND Entity

```
USE ieee.std_logic_1164.all;
ENTITY and2p IS
        GENERIC (tpd : TIME );
        PORT (in1, in2 : IN std_logic;
              out1 : OUT std_logic);
END and2p;

ARCHITECTURE pcircuit OF and2p IS
BEGIN
        out1 <= in1 AND in2 AFTER tpd;
END pcircuit;
```

C.2.2 2 Input OR Entity

```
USE ieee.std_logic_1164.all;
ENTITY or2p IS
        GENERIC (tpd : TIME );
        PORT (in1, in2 : IN std_logic;
              out1 : OUT std_logic);
END or2p;

ARCHITECTURE pcircuit OF or2p IS
BEGIN
        out1 <= in1 OR in2 AFTER tpd;
END pcircuit;
```

C.2.3 2 Input XOR Entity

```
USE ieee.std_logic_1164.all;
ENTITY xor2p IS
        GENERIC (tpd : TIME );
        PORT (in1, in2 : IN std_logic;
              out1 : OUT std_logic);
END xor2p;

ARCHITECTURE pcircuit OF xor2p IS
BEGIN
        out1 <= in1 XOR in2 AFTER tpd;
END pcircuit;
```

C.2.4 Inverter Entity

```
USE ieee.std_logic_1164.all;
ENTITY invp IS
        GENERIC (tpd : TIME );
        PORT (in1 : IN std_logic;
              out1 : OUT std_logic);
END invp;

ARCHITECTURE pcircuit OF invp IS
BEGIN
        out1 <= NOT in1 AFTER tpd;
END pcircuit;
```

C.2.5 D-Latch Entity

```
USE ieee.std_logic_1164.all;
ENTITY dlatchp IS
        GENERIC (tpd, tpd_en : TIME );
        PORT (in1, latch_en : IN std_logic;
              out1 : OUT std_logic);
END dlatchp;

ARCHITECTURE pcircuit OF dlatchp IS
        SIGNAL del_sig : std_logic;
        SIGNAL en_sig : std_logic;
BEGIN
        en_sig <= latch_en after tpd_en;

        state_change0 : PROCESS (in1)
        BEGIN
          IF (in1 = ‘0’ OR in1 = ‘1’) THEN
            del_sig <= in1 after tpd;
          ELSE
            del_sig <= ‘X’ after tpd;
          END IF;
        END PROCESS;

        state_change1 : PROCESS (en_sig, del_sig)
        BEGIN
          IF ( To_bit(en_sig) = ‘1’) THEN
            out1 <= del_sig after 0 ns;
          END IF;
        END PROCESS;
END pcircuit;
```

C.2.6 D-Flip-Flop Entity

```
USE ieee.std_logic_1164.all;
ENTITY edge_dffp IS
        GENERIC (tpd_clock, tpd_res : TIME );
        PORT (in1, clk, clr : IN std_logic;
              out1 : OUT std_logic);
END edge_dffp;

ARCHITECTURE pcircuit OF edge_dffp IS
BEGIN
        state_change : PROCESS (clk,clr)
        BEGIN
           IF ( To_bit(clr) = ‘1’ ) THEN
             out1 <= ‘0’ AFTER tpd_res;
           ELSIF ( clk’event AND clk = ‘1’ ) THEN
            IF (in1 = ‘0’ OR in1 = ‘1’) THEN
               out1 <= in1 after tpd_clock;
            ELSE
               out1 <= ‘X’ after tpd_clock;
            END IF;
           END IF;
        END PROCESS;
END pcircuit;
```

C.2.7 Pure Delay Entity

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY damelib;
USE damelib.primitive.edge_dffp;
ENTITY pure_delayp IS
        GENERIC (tpd : TIME );
        PORT (in1 : IN std_logic;
              sys_reset : IN std_logic;
              sys_clock : IN std_logic;
              out1 : OUT std_logic);
END pure_delayp;
```

```
ARCHITECTURE pcircuit OF pure_delayp IS
BEGIN
        out1 <= in1 after tpd;
END pcircuit;
```

C.2.7.1 D-Flip-Flop Implemenation of 50 ns Pure delay

```
ARCHITECTURE del50ns OF pure_delayp IS
   FOR ALL : edge_dffp
          USE entity damelib.edge_dffp(pcircuit);
   SIGNAL sig1 : std_logic;
BEGIN
   part1 : edge_dffp
        PORT MAP (in1, sys_clock, sys_reset, sig1);
   part2 : edge_dffp
        PORT MAP (sig1, sys_clock, sys_reset, out1);
END del50ns;
```

C.2.8 Leading Edge Delay Entity

```
USE ieee.std_logic_1164.all;
LIBRARY damelib;
USE damelib.primitive.and2p;
USE damelib.primitive.pure_delayp;
ENTITY leading_edge_delayp IS
        GENERIC (tpd_edge, tpd : TIME );
        PORT (in1 : IN std_logic;
              sys_reset : IN std_logic;
              sys_clock : IN std_logic;
              out1 : OUT std_logic);
END leading_edge_delayp;

ARCHITECTURE pcircuit OF leading_edge_delayp IS
        FOR all : and2p
           use entity damelib.and2p(pcircuit);
        FOR all : pure_delayp
           use entity damelib.pure_delayp(pcircuit);
        SIGNAL idel : std_logic;
BEGIN
        n1 : and2p
           GENERIC MAP (tpd => tpd)
           PORT MAP (in1 => idel, in2 => in1, out1 => out1);
        n2 : pure_delayp
           GENERIC MAP (tpd => tpd_edge)
           PORT MAP (in1 => in1,
                     sys_reset => sys_reset,
                     sys_clock => sys_clock,
                     out1 => idel);
END pcircuit;

ARCHITECTURE del50ns OF leading_edge_delayp IS
        FOR all : and2p
           use entity damelib.and2p(pcircuit);
        FOR all : pure_delayp
           use entity damelib.pure_delayp(del50ns);
        SIGNAL idel : std_logic;
BEGIN
        n1 : and2p
           GENERIC MAP (tpd => tpd)
           PORT MAP (in1 => idel, in2 => in1, out1 => out1);
        n2 : pure_delayp
           GENERIC MAP (tpd => tpd_edge)
           PORT MAP (in1 => in1,
                     sys_reset => sys_reset,
                     sys_clock => sys_clock,
                     out1 => idel);
END del50ns;
```

C.2.9 Trailing Edge Delay Entity

```
USE ieee.std_logic_1164.all;
LIBRARY damelib;
USE damelib.primitive.and2p;
USE damelib.primitive.invp;
USE damelib.primitive.pure_delayp;
ENTITY trailing_edge_delayp IS
        GENERIC (tpd_edge, tpd : TIME );
        PORT (in1 : IN std_logic;
              sys_reset : IN std_logic;
              sys_clock : IN std_logic;
              out1 : OUT std_logic);
END trailing_edge_delayp;

ARCHITECTURE pcircuit OF trailing_edge_delayp IS
        FOR all : and2p
           use entity damelib.and2p(pcircuit);
        FOR all : invp
           use entity damelib.invp(pcircuit);
        FOR all : pure_delayp
           use entity damelib.pure_delayp(pcircuit);
        SIGNAL idel, inv_idel : std_logic;
BEGIN
        n1 : and2p
           GENERIC MAP (tpd => tpd)
           PORT MAP (in1 => inv_idel, in2 => in1, out1 => out1);
        n2 : pure_delayp
           GENERIC MAP (tpd => tpd_edge)
           PORT MAP (in1 => in1,
                     sys_reset => sys_reset,
                     sys_clock => sys_clock,
                     out1 => idel);
        n3 : invp
           GENERIC MAP (tpd => tpd)
           PORT MAP (in1 => idel,  out1 => inv_idel);
END pcircuit;

ARCHITECTURE del50ns OF trailing_edge_delayp IS
        FOR all : and2p
           use entity damelib.and2p(pcircuit);
        FOR all : invp
           use entity damelib.invp(pcircuit);
        FOR all : pure_delayp
           use entity damelib.pure_delayp(del50ns);
        SIGNAL idel, inv_idel : std_logic;
BEGIN
        n1 : and2p
           GENERIC MAP (tpd => tpd)
           PORT MAP (in1 => inv_idel, in2 => in1, out1 => out1);
        n2 : pure_delayp
           GENERIC MAP (tpd => tpd_edge)
           PORT MAP (in1 => in1,
                     sys_reset => sys_reset,
                     sys_clock => sys_clock,
                     out1 => idel);
        n3 : invp
           GENERIC MAP (tpd => tpd)
           PORT MAP (in1 => idel,  out1 => inv_idel);
END del50ns;
```

C.2.10 Tri-Sate Buffer Entity

```
USE ieee.std_logic_1164.all;
ENTITY tri_state_bufferp IS
        GENERIC (tpd : TIME; tpd_tri : TIME);
        PORT (in1, tri_out : IN std_logic;
              out1 : OUT std_logic);
END tri_state_bufferp;

ARCHITECTURE pcircuit OF tri_state_bufferp IS
```

```
        SIGNAL del_sig : std_logic;
        SIGNAL tri_sig : std_logic;
BEGIN
        tri_sig <= tri_out after tpd_tri;

        state_change0 : PROCESS (in1)
        BEGIN
          IF (in1 = ‘0’ OR in1 = ‘1’) THEN
            del_sig <= in1 after tpd;
          ELSE
            del_sig <= ‘X’ after tpd;
          END IF;
        END PROCESS;

        state_change1 : PROCESS (tri_sig, del_sig)
        BEGIN
          IF ( To_bit(tri_sig) = ‘1’) THEN
            out1 <= del_sig after 0 ns;
          ELSE
            out1 <= ‘Z’ after 0 ns;
          END IF;
        END PROCESS;
END pcircuit;
```

C.2.11 Open Collector Buffer Entity

```
USE ieee.std_logic_1164.all;
ENTITY oc_bufferp IS
        GENERIC (tpd : TIME );
        PORT (in1 : IN std_logic;
              out1 : OUT std_logic);
END oc_bufferp;

ARCHITECTURE pcircuit OF oc_bufferp IS
BEGIN
        state_change : PROCESS (in1)
        BEGIN
          IF ( To_bit(in1) = ‘0’ ) THEN
            out1 <= ‘0’ after tpd;
          ELSE
            out1 <= ‘Z’ after tpd;
          END IF;
        END PROCESS;
END pcircuit;Abstraction Levels for Information Transfers
```

Appendix D
CRL Frames for Design Example from Section 7.4

D.1 CRL Frames for the Motorola MC68000 Microprocessor

The component frames are divided into two parts. First, the body frames only contain the timing independent frames. The body frames are the same for all speed versions of a component. Second the timing frames contain the frames that are specific to a certian speed component.

D.1.1 CRL Frames MC68000 Body

Note: some address and data signals have been deleted for brevity.

```
(defschema m68000
         :parallel
         (is-a microprocessor)
         (has-capability
                        m68000_data_transfer_cap m68000_bus_arb
                        m68000_interrupt)
         (has-signal
                        m68000_as m68000_uds m68000_lds
                        m68000_dtak m68000_rd
                        m68000_fc0 m68000_fc1 m68000_fc2
                        m68000_br m68000_bg m68000_bgack
                        m68000_d0 m68000_d1 m68000_d2 m68000_d3
                        m68000_d4 m68000_d5 m68000_d6 m68000_d7
                        m68000_d8 m68000_d9 m68000_d10 m68000_d11
                        m68000_d12 m68000_d13 m68000_d14 m68000_d15
                        m68000_a1 m68000_a2 m68000_a3
                        m68000_a4 m68000_a5 m68000_a6 m68000_a7
                        m68000_a8 m68000_a9 m68000_a10 m68000_a11
                        m68000_a12 m68000_a13 m68000_a14 m68000_a15
                        m68000_a16 m68000_a17 m68000_a18 m68000_a19
                        m68000_a20 m68000_a21 m68000_a22 m68000_a23)
         (has-bus
                        m68000_data_bus m68000_address_bus
                        m68000_control_bus m68000_data_transfer_bus))
(defschema m68000_as
         :parallel
         (is-a signal)
         (pin-number 6)
         (polarity (ass 0))
         (sim-name )
         (sim-timing m68000_ds_as_timing)
         (sim-function s)
         (i-o  output))
(defschema m68000_dtak
         :parallel
         (is-a signal)
         (pin-number 10)
         (polarity (ass 0))
         (driver-type open_collector)
         (sim-name )
         (sim-timing m68000_control_timing)
         (sim-function k)
         (i-o  input))
(defschema m68000_uds
         :parallel
         (is-a signal)
         (pin-number 7)
         (polarity (ass 0))
         (sim-name )
         (sim-timing m68000_uds_lds_timing)
         (sim-function s)
         (i-o output))
(defschema m68000_lds
         :parallel
         (is-a signal)
         (pin-number 8)
         (polarity (ass 0))
         (sim-name )
         (sim-timing m68000_uds_lds_timing)
         (sim-function s)
         (i-o  output))
(defschema m68000_rw
         :parallel
         (is-a signal)
         (pin-number 9)
         (polarity (ass 0))
         (sim-name )
         (sim-timing m68000_ds_rw_timing)
         (sim-function w)
         (i-o  output))
(defschema m68000_fc0
         :parallel
         (is-a signal)
         (pin-number 28)
         (polarity (ass 1))
         (sim-name )
         (sim-timing m68000_ds_fc_timing)
         (sim-function 0a)
         (i-o  output))
(defschema m68000_fc1
         :parallel
         (is-a signal)
         (pin-number 27)
         (polarity (ass 1))
         (sim-name )
         (sim-timing m68000_ds_fc_timing)
         (sim-function 1a)
         (i-o  output))
(defschema m68000_fc2
         :parallel
         (is-a signal)
         (pin-number 26)
         (polarity (ass 1))
         (sim-name )
         (sim-timing m68000_ds_fc_timing)
         (sim-function 0a)
         (i-o  output))
(defschema m68000_a1
         :parallel
         (is-a signal)
         (pin-number 29)
         (polarity (ass 1))
         (sim-name (m68000_a 1))
         (sim-timing m68000_address_timing)
         (sim-function a1)
         (i-o  output))
......
(defschema m68000_a2 to m68000_a22 ....
......
(defschema m68000_a23
         :parallel
         (is-a signal)
         (pin-number 52)
         (polarity (ass 1))
         (sim-name (m68000_a 23))
         (sim-timing m68000_address_timing)
         (sim-function a23)
         (i-o  output))
(defschema m68000_d0
         :parallel
         (is-a signal)
         (pin-number 5)
         (polarity (ass 1))
         (sim-name (m68000_ld 0))
         (sim-timing m68000_write_data_timing)
         (sim-function d0)
         (i-o  input output))
......
(defschema m68000_d1 to m68000_d15
......
(defschema m68000_data_transfer_bus
                 :parallel
                 (is-a signal_bus)
                 (has-signal
                  m68000_d0 m68000_d1 m68000_d2 m68000_d3
                  m68000_d4 m68000_d5 m68000_d6 m68000_d7
                  m68000_d8 m68000_d9 m68000_d10 m68000_d11
                  m68000_d12 m68000_d13 m68000_d14 m68000_d15
                  m68000_a1 m68000_a2 m68000_a3
                  m68000_a4 m68000_a5 m68000_a6 m68000_a7
                  m68000_a8 m68000_a9 m68000_a10 m68000_a11
                  m68000_a12 m68000_a13 m68000_a14 m68000_a15
                  m68000_a16 m68000_a17 m68000_a18 m68000_a19
                  m68000_a20 m68000_a21 m68000_a22 m68000_a23
                  m68000_rw m68000_as m68000_dtak m68000_uds m68000_lds
                  ))
(defschema m68000_data_bus
         :parallel
         (is-a signal_bus)
         (sim-timing m68000_write_data_timing m68000_read_data_timing)
         (has-signal
                  m68000_d0 m68000_d1 m68000_d2 m68000_d3
                  m68000_d4 m68000_d5 m68000_d6 m68000_d7
                  m68000_d8 m68000_d9 m68000_d10 m68000_d11
                  m68000_d12 m68000_d13 m68000_d14 m68000_d15))
(defschema m68000_address_bus
         :parallel
         (is-a signal_bus)
         (sim-timing m68000_address_timing)
         (has-signal
                  m68000_a1 m68000_a2 m68000_a3
                  m68000_a4 m68000_a5 m68000_a6 m68000_a7
                  m68000_a8 m68000_a9 m68000_a10 m68000_a11
                  m68000_a12 m68000_a13 m68000_a14 m68000_a15
                  m68000_a16 m68000_a17 m68000_a18 m68000_a19
                  m68000_a20 m68000_a21 m68000_a22 m68000_a23))
(defschema m68000_control_bus
         :parallel
         (is-a signal_bus)
         (has-signal m68000_rw m68000_lds m68000_uds m68000_dtak m68000_as
                                         m68000_br m68000_bg m68000_bgack))
(defschema m68000_data_transfer_cap
         :parallel
         (is-a data_transfer_cap_micro)
         (has-dt-type-spec m68000_dt_type_spec)
         (has-dt-req-spec m68000_dt_req_spec)
         (has-word-select-spec m68000_word_select_spec)
         (has-word-width-spec none)
         (has-address-spec m68000_address_spec)
         (uses-protocol m68000_dt_write_protocol m68000_dt_read_protocol)
         (uses-timing m68000_ds_as_timing))
(defschema m68000_dt_read_protocol
         :parallel
         (is-a   read_protocol)
         (has-identification-spec     m68000_read_spec)
         (has-del-spec                m68000_del_spec)
         (has-data-spec               m68000_read_data_spec))
(defschema m68000_dt_write_protocol
         :parallel
         (is-a   write_protocol)
         (has-identification-spec     m68000_write_spec)
         (has-del-spec            m68000_del_spec)
         (has-data-spec               m68000_write_data_spec))
(defschema m68000_read_spec
         :parallel
         (is-a                  state_timing_spec)
         (uses-state            m68000_read_state)
         (uses-timing           m68000_ds_rw_timing))
(defschema m68000_read_state
         :parallel
         (is-a                  state_spec)
         (state-signals         m68000_rw)
         (state                 (nego m68000_rw)))
(defschema m68000_write_spec
         :parallel
         (is-a                  state_timing_spec)
         (uses-state            m68000_write_state)
         (uses-timing           m68000_ds_rw_timing))
(defschema m68000_write_state
         :parallel
         (is-a                  state_spec)
         (state-signals         m68000_rw)
         (state                 (asso m68000_rw)))
(defschema m68000_data_magnitude
         :parallel
         (is-a lookup_table)
         (selects 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
         (access-table   (0 m68000_d0) (1 m68000_d1) (2 m68000_d2)
                         (3 m68000_d3) (4 m68000_d4) (5 m68000_d5)
                         (6 m68000_d6) (7 m68000_d7) (8 m68000_d8)
                         (9 m68000_d9) (10 m68000_d10) (11 m68000_d11)
                         (12 m68000_d12) (13 m68000_d13) (14 m68000_d14)
                         (15 m68000_d15)))
(defschema m68000_read_data_spec
         :parallel
         (is-a           state_timing_spec)
         (uses-state     m68000_data_magnitude)
         (uses-timing    m68000_read_data_timing))
(defschema m68000_write_data_spec
         :parallel
         (is-a                  state_timing_spec)
         (uses-state            m68000_data_magnitude)
         (uses-timing           m68000_write_data_timing))
(defschema m68000_address_magnitude
         :parallel
         (is-a lookup_table)
         (selects 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
         (access-table  (1 m68000_a1)
                        (2 m68000_a2) (3 m68000_a3) (4 m68000_a4)
                        (5 m68000_a5) (6 m68000_a6) (7 m68000_a7)
                        (8 m68000_a8) (9 m68000_a9) (10 m68000_a10)
                        (11 m68000_a11) (12 m68000_a12) (13 m68000_a13)
                        (14 m68000_a14) (15 m68000_a15) (16 m68000_a16)
                        (17 m68000_a17) (18 m68000_a18) (19 m68000_a19)
                        (20 m68000_a20) (21 m68000_a21) (22 m68000_a22)
                        (23 m68000_a23)))
(defschema m68000_address_spec
         :parallel
         (is-a          state_timing_spec)
         (uses-state    m68000_address_magnitude)
         (uses-timing   m68000_address_timing))
(defschema m68000_dt_type_spec
         :parallel
         (is-a                    state_timing_spec)
         (uses-state              m68000_type_state)
         (uses-timing             m68000_ds_fc_timing))
(defschema m68000_type_state
         :parallel
         (is-a lookup_table)
         (condition-signals       m68000_fc0 m68000_fc1 m68000_fc2)
         (selects                 user_data user_program sup_data
                                  sup_program int_ack)
         (access-table  (user_data   (and (nego m68000_fc2) (nego m68000_fc1)
                                                           (asso m68000_fc0)))
                        (user_program(and (nego m68000_fc2) (asso m68000_fc1)
                                                         (nego m68000_fc0)))
                        (sup_data    (and (asso m68000_fc2) (nego m68000_fc1)
                                                         (asso m68000_fc0)))
                        (sup_program (and (asso m68000_fc2) (asso m68000_fc1)
                                                         (nego m68000_fc0)))
                        (int_ack     (and (asso m68000_fc2) (asso m68000_fc1)
                                                         (asso m68000_fc0)))))
(defschema m68000_dt_req_spec
         :parallel
         (is-a                    state_timing_spec)
         (uses-state              m68000_dt_req_state)
         (uses-timing             m68000_uds_lds_timing m68000_ds_as_timing))
(defschema m68000_dt_req_state
         :parallel
         (is-a                    state_spec)
         (state-signals           m68000_uds m68000_lds)
         (state                 (and (asso m68000_as)
                                (or (asso m68000_uds) (asso m68000_lds)))))
(defschema m68000_word_select_spec
         :parallel
         (is-a                    state_timing_spec)
         (uses-state              m68000_word_select_state)
         (uses-timing             m68000_uds_lds_timing))
(defschema m68000_word_select_state
         :parallel
         (is-a                    double_lookup_table)
         (condition-signals       m68000_uds m68000_lds)
         (select1                 (0 7) (0 15))
         (select2                 (0 7) (8 15))
         (access-table            ((0 15) (((0 7) (asso m68000_lds))
                                           ((8 15) (asso m68000_uds))))
                                  ((0 7) (((0 7) (asso m68000_lds))
                                          ((8 15) (asso m68000_uds))))))
(defschema m68000_del_spec
         :parallel
         (is-a                    state_timing_spec)
         (uses-state              m68000_del_state)
         (uses-timing             m68000_control_timing))
(defschema m68000_del_state
         :parallel
         (is-a                    state_spec)
         (state-signals           m68000_dtak)
         (state                   (assi m68000_dtak)))
```

D.1.2 CRL Frames MC68000 Timing (8Mhz)

```
(defschema m68000_ds_rw_timing
         :parallel
         (is-a                    strobe_timing)
         (signal1                 m68000_uds m68000_lds)
         (signal2                 m68000_rw)
         (event1                 (! (or (asso m68000_uds) (asso m68000_lds))))
         (event3                 (! (not (or (asso m68000_uds)
                                             (asso m68000_lds)))))
         (event2                  (valo ! valo m68000_rw))
         (event4                  (valo ! valo m68000_rw))
         (time2                   (-~ -60))
         (time4                   (40 +~)))
(defschema m68000_control_timing
         :parallel
         (is-a                    handshake_timing)
         (signal1                 m68000_uds m68000_lds)
         (signal2                 m68000_dtak)
         (event1                 (! (or (asso m68000_uds) (asso m68000_lds))))
         (event3                 (! (not (or (asso m68000_uds)
                                             (asso m68000_lds)))))
         (event2                  (! assi m68000_dtak))
         (event4                  (! negi m68000_dtak))
```

```
        (time2                    (variable))
        (time3                    (125 320))
        (time4                    (0 245)))
(defschema m68000_read_data_timing
        :parallel
        (is-a                     pulse_latch_timing)
        (signal1                  m68000_uds m68000_lds)
        (signal2                  m68000_data_bus)
        (event1                  (! (or (asso m68000_uds) (asso m68000_lds))))
        (event3                  (! (not (or (asso m68000_uds)
                                             (asso m68000_lds)))))
        (event2                   (float ! vali m68000_data_bus))
        (event4                   (vali ! float m68000_data_bus))
        (time2                    (-~ -35))
        (time4                    (0 +~)))
(defschema m68000_write_data_timing
        :parallel
        (is-a                     strobe_timing)
        (signal1                  m68000_uds m68000_lds)
        (signal2                  m68000_data_bus)
        (event1                  (! (or (asso m68000_uds) (asso m68000_lds))))
        (event3                  (! (not (or (asso m68000_uds)
                                             (asso m68000_lds)))))
        (event2                   (open ! valo m68000_data_bus))
        (event4                   (valo ! open m68000_data_bus))
        (time2                    (-~ -30))
        (time4                    (30 +~)))
(defschema m68000_address_timing
        :parallel
        (is-a                     strobe_timing)
        (signal1                  m68000_uds m68000_lds)
        (signal2                  m68000_address_bus)
        (event1                  (! (or (asso m68000_uds) (asso m68000_lds))))
        (event3                  (! (not (or (asso m68000_uds)
                                             (asso m68000_lds)))))
        (event2                   (! valo m68000_address_bus))
        (event4                   (! ivalo m68000_address_bus))
        (time2                    (-~ -30))
        (time4                    (30 +~)))
(defschema m68000_ds_as_timing
        :parallel
        (is-a                     strobe_timing)
        (signal1                  m68000_uds m68000_lds)
        (signal2                  m68000_as)
        (event1                  (! (or (asso m68000_uds) (asso m68000_lds))))
        (event3                  (! (not (or (asso m68000_uds)
                                             (asso m68000_lds)))))
        (event2                   (valo ! asso m68000_as))
        (event4                   (asso ! valo m68000_as))
        (time2                    (-~ 0))
        (time4                    (0 +~)))
(defschema m68000_ds_fc_timing
        :parallel
        (is-a                     strobe_timing)
        (signal1                  m68000_uds m68000_lds)
        (signal2                  m68000_fc0 m68000_fc1 m68000_fc2)
        (event1                  (! (or (asso m68000_uds) (asso m68000_lds))))
        (event3                  (! (not (or (asso m68000_uds)
                                             (asso m68000_lds)))))
        (event2                   (! valo sig2))
        (event4                   (! ivalo sig2))
        (time2                    (-~ -60))
        (time4                    (40 +~)))
(defschema m68000_uds_lds_timing
        :parallel
        (is-a                     logic_timing)
        (signal1                  m68000_lds m68000_uds)
        (signal2                  m68000_lds m68000_uds)
        (event1                  (! (or (asso m68000_uds) (asso m68000_lds))))
        (event3                  (! (not (or (asso m68000_uds)
                                             (asso m68000_lds)))))
        (event2                   (! asso sig2))
```

```
         (event4                  (! nego sig2))
         (time2                   (0))
         (time4                   (0)))
```

D.2 CRL Frames for Component Instances and the Connection Request

```
(DEFSCHEMA U1
  :PARALLEL
  (INSTANCE M68000)
  (PACKAGE DIP))
(DEFSCHEMA U2
  :PARALLEL
  (INSTANCE M6116)
  (PACKAGE DIP))
(DEFSCHEMA U3
  :PARALLEL
  (INSTANCE M6116)
  (PACKAGE DIP))
(DEFSCHEMA U4
  :PARALLEL
  (INSTANCE M6116)
  (PACKAGE DIP))
(DEFSCHEMA U5
  :PARALLEL
  (INSTANCE M6116)
  (PACKAGE DIP))
(defschema connection_request_1
  :parallel
  (is-a connection_request)
  (has-sub-request connection_sub_request_1 connection_sub_request_2
                   connection_sub_request_3 connection_sub_request_4)
  (purpose data_transfer)
  (status new)
  (component1 u1)
  (component2 u2 u3 u4 u5)
  (direction bidir)
  (usage1 (user_data sup_data user_program sup_program))
  (usage2 (T))
  (add-decode1
     (0000 (and (negi 23) (negi 22) (negi 21) (negi 20) (negi 19) (negi 18)
               (negi 17) (negi 16) (negi 15) (negi 14)
               (negi 13) (negi 12)))
     (8000 (and (negi 23) (negi 22) (negi 21) (negi 20) (negi 19) (negi 18)
               (negi 17) (negi 16) (assi 15) (negi 14)
               (negi 13) (negi 12))))
  (data-decode1 (0 7) (8 15))
  (data-decode2 (0 7))
  (add1 (1 11))
  (add2 (0 10))
  (interface-width (0 15)))
(defschema connection_sub_request_1
  :parallel
  (is-a connection_sub_request)
  (sub-request-of )
  (component1 u1)
  (component2 u2)
  (add-decode1 0000)
  (data-decode1 (0 7))
  (data-decode2 (0 7)))
(defschema connection_sub_request_2
  :parallel
  (is-a connection_sub_request)
  (sub-request-of )
  (component1 u1)
  (component2 u3)
  (add-decode1 0000)
  (data-decode1 (8 15))
  (data-decode2 (0 7)))
(defschema connection_sub_request_3
  :parallel
  (is-a connection_sub_request)
  (sub-request-of )
  (component1 u1)
  (component2 u4)
  (add-decode1 8000)
  (data-decode1 (0 7))
  (data-decode2 (0 7)))

(defschema connection_sub_request_4
  :parallel
  (is-a connection_sub_request)
  (sub-request-of )
  (component1 u1)
  (component2 u5)
  (add-decode1 8000)
  (data-decode1 (8 15))
  (data-decode2 (0 7)))
```

Appendix E
VHDL Code for Design Example from Section 7.4

The VHDL code is broken into three modules: the VHDL code for the ISBs, the IB and the test bench.

E.1 VHDL ISBs for Design Example

Note: some address and data signals have been deleted for brevity.

```
-- START of vhdl code for IB_1_RW_CONNECT
-- Device 1 : IB_1_RW_CONNECT
-- Generated on: 15:00:20 8-2-1997      Version 1.0
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library DAMELIB;

entity ISB_270_M68000_A11_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_A11_U1 : IN std_logic;
             M6116_A10_U2345 : OUT std_logic );
end ISB_270_M68000_A11_INT;

architecture ONLY of ISB_270_M68000_A11_INT is
   use DAMELIB.PRIMITIVE.ALL;
   for all : TRI_STATE_BUFFERP
       use entity DAMELIB.TRI_STATE_BUFFERP(pcircuit);
   signal SIG_ENAB : std_logic;
   signal SIG_ISIG : std_logic;
begin
   PART1 : TRI_STATE_BUFFERP
      generic map (
             TPD => TPD,
             TPD_TRI => TPD_EN )
      port map (
             IN1 => SIG_ISIG,
             TRI_OUT => SIG_ENAB,
             OUT1 => M6116_A10_U2345 );
     SIG_ISIG <=
                (
                     M68000_A11_U1
                ) after 0 ns;
--   Complexity was =  1
     SIG_ENAB <=
                (
                ‘1’
                ) after 0 ns;
--   Complexity was =  0
end ONLY;
entity ISB_250_M68000_A1_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_A1_U1 : IN std_logic;
             M6116_A0_U2345 : OUT std_logic );
end ISB_250_M68000_A1_INT;

architecture ONLY of ISB_250_M68000_A1_INT is
   use DAMELIB.PRIMITIVE.ALL;
   for all : TRI_STATE_BUFFERP
```

```
       use entity DAMELIB.TRI_STATE_BUFFERP(pcircuit);
   signal SIG_ENAB : std_logic;
   signal SIG_ISIG : std_logic;
begin
   PART1 : TRI_STATE_BUFFERP
      generic map (
             TPD => TPD,
             TPD_TRI => TPD_EN )
      port map (
             IN1 => SIG_ISIG,
             TRI_OUT => SIG_ENAB,
             OUT1 => M6116_A0_U2345 );
     SIG_ISIG <=
                (
                     M68000_A1_U1
                ) after 0 ns;
--   Complexity was =  1
     SIG_ENAB <=
                (
                ‘1’
                ) after 0 ns;
--   Complexity was =  0
end ONLY;

entity ISB_137_M6116_D0_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M6116_D0_U35 : IN std_logic;
             ISB_55_DATA_ACC_EN_INT_SIGNAL : IN std_logic;
             ISB_38_READ_INT_SIGNAL : IN std_logic;
             M68000_D8_U1 : OUT std_logic );
end ISB_137_M6116_D0_INT;

architecture ONLY of ISB_137_M6116_D0_INT is
   use DAMELIB.PRIMITIVE.ALL;
   for all : TRI_STATE_BUFFERP
       use entity DAMELIB.TRI_STATE_BUFFERP(pcircuit);
   signal SIG_ENAB : std_logic;
   signal SIG_ISIG : std_logic;
begin
   PART1 : TRI_STATE_BUFFERP
      generic map (
             TPD => TPD,
             TPD_TRI => TPD_EN )
      port map (
             IN1 => SIG_ISIG,
             TRI_OUT => SIG_ENAB,
             OUT1 => M68000_D8_U1 );
     SIG_ISIG <=
                (
                     M6116_D0_U35
                ) after 0 ns;
--   Complexity was =  1
     SIG_ENAB <=
                (
                    (
                     ISB_55_DATA_ACC_EN_INT_SIGNAL
                    )
                and (
                     ISB_38_READ_INT_SIGNAL
                    )
                ) after TPD;
--   Complexity was =  3
end ONLY;

entity ISB_121_M68000_D8_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_D8_U1 : IN std_logic;
             ISB_55_DATA_ACC_EN_INT_SIGNAL : IN std_logic;
             ISB_35_WRITE_INT_SIGNAL : IN std_logic;
             M6116_D0_U35 : OUT std_logic );
end ISB_121_M68000_D8_INT;

architecture ONLY of ISB_121_M68000_D8_INT is
   use DAMELIB.PRIMITIVE.ALL;
   for all : TRI_STATE_BUFFERP
       use entity DAMELIB.TRI_STATE_BUFFERP(pcircuit);
   signal SIG_ENAB : std_logic;
   signal SIG_ISIG : std_logic;
begin
   PART1 : TRI_STATE_BUFFERP
      generic map (
             TPD => TPD,
             TPD_TRI => TPD_EN )
      port map (
             IN1 => SIG_ISIG,
             TRI_OUT => SIG_ENAB,
             OUT1 => M6116_D0_U35 );
     SIG_ISIG <=
                (
                     M68000_D8_U1
                ) after 0 ns;
--   Complexity was =  1
     SIG_ENAB <=
                (
                    (
                     ISB_55_DATA_ACC_EN_INT_SIGNAL
                    )
                and (
                     ISB_35_WRITE_INT_SIGNAL
                    )
                ) after TPD;
--   Complexity was =  3
end ONLY;

entity ISB_105_M6116_D0_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M6116_D0_U24 : IN std_logic;
             ISB_55_DATA_ACC_EN_INT_SIGNAL : IN std_logic;
             ISB_38_READ_INT_SIGNAL : IN std_logic;
             M68000_D0_U1 : OUT std_logic );
end ISB_105_M6116_D0_INT;

architecture ONLY of ISB_105_M6116_D0_INT is
   use DAMELIB.PRIMITIVE.ALL;
   for all : TRI_STATE_BUFFERP
       use entity DAMELIB.TRI_STATE_BUFFERP(pcircuit);
   signal SIG_ENAB : std_logic;
   signal SIG_ISIG : std_logic;
begin
   PART1 : TRI_STATE_BUFFERP
      generic map (
             TPD => TPD,
             TPD_TRI => TPD_EN )
      port map (
             IN1 => SIG_ISIG,
             TRI_OUT => SIG_ENAB,
             OUT1 => M68000_D0_U1 );
     SIG_ISIG <=
                (
                     M6116_D0_U24
                ) after 0 ns;
--   Complexity was =  1
```

```
     SIG_ENAB <=
                (
                    (
                     ISB_55_DATA_ACC_EN_INT_SIGNAL
                    )
                and (
                     ISB_38_READ_INT_SIGNAL
                    )
                ) after TPD;
--   Complexity was =  3
end ONLY;

entity ISB_89_M68000_D0_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_D0_U1 : IN std_logic;
             ISB_55_DATA_ACC_EN_INT_SIGNAL : IN std_logic;
             ISB_35_WRITE_INT_SIGNAL : IN std_logic;
             M6116_D0_U24 : OUT std_logic );
end ISB_89_M68000_D0_INT;

architecture ONLY of ISB_89_M68000_D0_INT is
   use DAMELIB.PRIMITIVE.ALL;
   for all : TRI_STATE_BUFFERP
       use entity DAMELIB.TRI_STATE_BUFFERP(pcircuit);
   signal SIG_ENAB : std_logic;
   signal SIG_ISIG : std_logic;
begin
   PART1 : TRI_STATE_BUFFERP
      generic map (
             TPD => TPD,
             TPD_TRI => TPD_EN )
      port map (
             IN1 => SIG_ISIG,
             TRI_OUT => SIG_ENAB,
             OUT1 => M6116_D0_U24 );
     SIG_ISIG <=
                (
                     M68000_D0_U1
                ) after 0 ns;
--   Complexity was =  1
     SIG_ENAB <=
                (
                    (
                     ISB_55_DATA_ACC_EN_INT_SIGNAL
                    )
                and (
                     ISB_35_WRITE_INT_SIGNAL
                    )
                ) after TPD;
--   Complexity was =  3
end ONLY;

entity ISB_71_M6116_WR_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_35_WRITE_INT_SIGNAL : IN std_logic;
             M6116_WR_U2345 : OUT std_logic );
end ISB_71_M6116_WR_INT;

architecture ONLY of ISB_71_M6116_WR_INT is
begin
     M6116_WR_U2345 <=
            not (
                     ISB_35_WRITE_INT_SIGNAL
                ) after TPD;
```

```
--   Complexity was =  2
end ONLY;

entity ISB_69_M6116_CE_INT_0_0_7 is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_44_TYPE_INT_SIGNAL : IN std_logic;
             ISB_16_ADD_INT_SIGNAL_0 : IN std_logic;
             ISB_10_WORD_INT_SIGNAL_0_7 : IN std_logic;
             ISB_4_REQUEST_INT_SIGNAL : IN std_logic;
             M6116_CE_U2 : OUT std_logic );
end ISB_69_M6116_CE_INT_0_0_7;

architecture ONLY of ISB_69_M6116_CE_INT_0_0_7 is
begin
     M6116_CE_U2 <=
            not (
                    (
                     ISB_44_TYPE_INT_SIGNAL
                    )
                and (
                     ISB_16_ADD_INT_SIGNAL_0
                    )
                and (
                     ISB_10_WORD_INT_SIGNAL_0_7
                    )
                and (
                     ISB_4_REQUEST_INT_SIGNAL
                    )
                ) after TPD;
--   Complexity was =  6
end ONLY;

entity ISB_69_M6116_CE_INT_0_8_15 is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_44_TYPE_INT_SIGNAL : IN std_logic;
             ISB_16_ADD_INT_SIGNAL_0 : IN std_logic;
             ISB_10_WORD_INT_SIGNAL_8_15 : IN std_logic;
             ISB_4_REQUEST_INT_SIGNAL : IN std_logic;
             M6116_CE_U3 : OUT std_logic );
end ISB_69_M6116_CE_INT_0_8_15;

architecture ONLY of ISB_69_M6116_CE_INT_0_8_15 is
begin
     M6116_CE_U3 <=
            not (
                    (
                     ISB_44_TYPE_INT_SIGNAL
                    )
                and (
                     ISB_16_ADD_INT_SIGNAL_0
                    )
                and (
                     ISB_10_WORD_INT_SIGNAL_8_15
                    )
                and (
                     ISB_4_REQUEST_INT_SIGNAL
                    )
                ) after TPD;
--   Complexity was =  6
end ONLY;

entity ISB_69_M6116_CE_INT_8000_0_7 is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_44_TYPE_INT_SIGNAL : IN std_logic;
             ISB_16_ADD_INT_SIGNAL_8000 : IN std_logic;
             ISB_10_WORD_INT_SIGNAL_0_7 : IN std_logic;
             ISB_4_REQUEST_INT_SIGNAL : IN std_logic;
             M6116_CE_U4 : OUT std_logic );
end ISB_69_M6116_CE_INT_8000_0_7;

architecture ONLY of ISB_69_M6116_CE_INT_8000_0_7 is
begin
     M6116_CE_U4 <=
            not (
                    (
                     ISB_44_TYPE_INT_SIGNAL
                    )
                and (
                     ISB_16_ADD_INT_SIGNAL_8000
                    )
                and (
                     ISB_10_WORD_INT_SIGNAL_0_7
                    )
                and (
                     ISB_4_REQUEST_INT_SIGNAL
                    )
                ) after TPD;
--   Complexity was =  6
end ONLY;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library DAMELIB;

entity ISB_69_M6116_CE_INT_8000_8_15 is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_44_TYPE_INT_SIGNAL : IN std_logic;
             ISB_16_ADD_INT_SIGNAL_8000 : IN std_logic;
             ISB_10_WORD_INT_SIGNAL_8_15 : IN std_logic;
             ISB_4_REQUEST_INT_SIGNAL : IN std_logic;
             M6116_CE_U5 : OUT std_logic );
end ISB_69_M6116_CE_INT_8000_8_15;

architecture ONLY of ISB_69_M6116_CE_INT_8000_8_15 is
begin
     M6116_CE_U5 <=
            not (
                    (
                     ISB_44_TYPE_INT_SIGNAL
                    )
                and (
                     ISB_16_ADD_INT_SIGNAL_8000
                    )
                and (
                     ISB_10_WORD_INT_SIGNAL_8_15
                    )
                and (
                     ISB_4_REQUEST_INT_SIGNAL
                    )
                ) after TPD;
--   Complexity was =  6
end ONLY;

entity ISB_67_M6116_OE_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_38_READ_INT_SIGNAL : IN std_logic;
             M6116_OE_U2345 : OUT std_logic );
end ISB_67_M6116_OE_INT;

architecture ONLY of ISB_67_M6116_OE_INT is
begin
     M6116_OE_U2345 <=
            not (
                     ISB_38_READ_INT_SIGNAL
                ) after TPD;
--   Complexity was =  2
end ONLY;

entity ISB_65_M68000_DTAK_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_61_DELAY_INT_SIGNAL : IN std_logic;
             M68000_DTAK_U1 : OUT std_logic );
end ISB_65_M68000_DTAK_INT;

architecture ONLY of ISB_65_M68000_DTAK_INT is
   use DAMELIB.PRIMITIVE.ALL;
   for all : OC_BUFFERP
       use entity DAMELIB.OC_BUFFERP(pcircuit);
   signal SIG_OSIG : std_logic;
begin
     SIG_OSIG <=
            not (
                     ISB_61_DELAY_INT_SIGNAL
                ) after TPD;
--   Complexity was =  2
   PART1_OC : OC_BUFFERP
      generic map (
             TPD => TPD )
      port map (
             IN1 => SIG_OSIG,
             OUT1 => M68000_DTAK_U1 );
end ONLY;

entity ISB_61_DELAY_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_58_ACC_DEL_INT_SIGNAL_S_1 : IN std_logic;
             ISB_61_DELAY_INT_SIGNAL : OUT std_logic );
end ISB_61_DELAY_INT;

architecture ONLY of ISB_61_DELAY_INT is
begin
     ISB_61_DELAY_INT_SIGNAL <=
                (
                     ISB_58_ACC_DEL_INT_SIGNAL_S_1
                ) after 0 ns;
--   Complexity was =  1
end ONLY;

entity ISB_63_CONV_SS is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_58_ACC_DEL_INT_SIGNAL : IN std_logic;
             SYS_RESET : IN std_logic;
             SYS_CLOCK : IN std_logic;
             ISB_58_ACC_DEL_INT_SIGNAL_S_1 : OUT std_logic );
```

```
end ISB_63_CONV_SS;

architecture ONLY of ISB_63_CONV_SS is
   use DAMELIB.PRIMITIVE.ALL;
   for all : LEADING_EDGE_DELAYP
       use entity DAMELIB.LEADING_EDGE_DELAYP(DEL100NS);
   signal SIG_ISIG : std_logic;
   signal SIG_OSIG : std_logic;
begin
   PART1 : LEADING_EDGE_DELAYP
      generic map (
             TPD_EDGE => 76 ns,
             TPD => TPD )
      port map (
             IN1 => SIG_ISIG,
             SYS_RESET => SYS_RESET,
             SYS_CLOCK => SYS_CLOCK,
             OUT1 => SIG_OSIG );
     SIG_ISIG <=
                (
                     ISB_58_ACC_DEL_INT_SIGNAL
                ) after 0 ns;
--   Complexity was =  1
     ISB_58_ACC_DEL_INT_SIGNAL_S_1 <=
                (
                     SIG_OSIG
                ) after 0 ns;
--   Complexity was =  1
end ONLY;

entity ISB_58_ACC_DEL_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_49_ACCESS_INT_SIGNAL : IN std_logic;
             ISB_58_ACC_DEL_INT_SIGNAL : OUT std_logic );
end ISB_58_ACC_DEL_INT;

architecture ONLY of ISB_58_ACC_DEL_INT is
begin
     ISB_58_ACC_DEL_INT_SIGNAL <=
                (
                     ISB_49_ACCESS_INT_SIGNAL
                ) after 0 ns;
--   Complexity was =  1
end ONLY;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library DAMELIB;

entity ISB_55_DATA_ACC_EN_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_49_ACCESS_INT_SIGNAL : IN std_logic;
             ISB_55_DATA_ACC_EN_INT_SIGNAL : OUT std_logic );
end ISB_55_DATA_ACC_EN_INT;

architecture ONLY of ISB_55_DATA_ACC_EN_INT is
begin
     ISB_55_DATA_ACC_EN_INT_SIGNAL <=
                (
                     ISB_49_ACCESS_INT_SIGNAL
                ) after 0 ns;
--   Complexity was =  1
end ONLY;
```

```
entity ISB_49_ACCESS_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_31_BLOCK_ADD_INT_SIGNAL : IN std_logic;
             ISB_4_REQUEST_INT_SIGNAL : IN std_logic;
             ISB_44_TYPE_INT_SIGNAL : IN std_logic;
             ISB_49_ACCESS_INT_SIGNAL : OUT std_logic );
end ISB_49_ACCESS_INT;

architecture ONLY of ISB_49_ACCESS_INT is
begin
     ISB_49_ACCESS_INT_SIGNAL <=
                (
                    (
                     ISB_31_BLOCK_ADD_INT_SIGNAL
                    )
                and (
                     ISB_4_REQUEST_INT_SIGNAL
                    )
                and (
                     ISB_44_TYPE_INT_SIGNAL
                    )
                ) after TPD;
--   Complexity was =  4
end ONLY;

entity ISB_44_TYPE_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_FC2_U1 : IN std_logic;
             M68000_FC1_U1 : IN std_logic;
             M68000_FC0_U1 : IN std_logic;
             ISB_44_TYPE_INT_SIGNAL : OUT std_logic );
end ISB_44_TYPE_INT;

architecture ONLY of ISB_44_TYPE_INT is
begin
     ISB_44_TYPE_INT_SIGNAL <=
                (
                    (
                    (
                      not M68000_FC2_U1
                    )
                and (
                      not M68000_FC1_U1
                    )
                and (
                     M68000_FC0_U1
                    )
                    )
                or (
                    (
                     M68000_FC2_U1
                    )
                and (
                      not M68000_FC1_U1
                    )
                and (
                     M68000_FC0_U1
                    )
                   )
                or (
                    (
                      not M68000_FC2_U1
                    )
                and (
                     M68000_FC1_U1
                    )
                and (
                      not M68000_FC0_U1
                    )
                   )
                or (
                    (
                     M68000_FC2_U1
                    )
                and (
                     M68000_FC1_U1
                    )
                and (
                      not M68000_FC0_U1
                    )
                   )
                ) after 3*TPD;
--   Complexity was =  17
end ONLY;

entity ISB_38_READ_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_RW_U1 : IN std_logic;
             ISB_38_READ_INT_SIGNAL : OUT std_logic );
end ISB_38_READ_INT;

architecture ONLY of ISB_38_READ_INT is
begin
     ISB_38_READ_INT_SIGNAL <=
                (
                     M68000_RW_U1
                ) after 0 ns;
--   Complexity was =  1
end ONLY;

entity ISB_35_WRITE_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_RW_U1 : IN std_logic;
             ISB_35_WRITE_INT_SIGNAL : OUT std_logic );
end ISB_35_WRITE_INT;

architecture ONLY of ISB_35_WRITE_INT is
begin
     ISB_35_WRITE_INT_SIGNAL <=
                (
                      not M68000_RW_U1
                ) after TPD;
--   Complexity was =  2
end ONLY;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library DAMELIB;

entity ISB_31_BLOCK_ADD_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             ISB_16_ADD_INT_SIGNAL_8000 : IN std_logic;
             ISB_16_ADD_INT_SIGNAL_0 : IN std_logic;
             ISB_31_BLOCK_ADD_INT_SIGNAL : OUT std_logic );
```

```
end ISB_31_BLOCK_ADD_INT;

architecture ONLY of ISB_31_BLOCK_ADD_INT is
begin
     ISB_31_BLOCK_ADD_INT_SIGNAL <=
                (
                    (
                     ISB_16_ADD_INT_SIGNAL_8000
                    )
                or (
                     ISB_16_ADD_INT_SIGNAL_0
                   )
                ) after TPD;
--   Complexity was =  3
end ONLY;

entity ISB_16_ADD_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_A23_U1 : IN std_logic;
             M68000_A22_U1 : IN std_logic;
             M68000_A21_U1 : IN std_logic;
             M68000_A20_U1 : IN std_logic;
             M68000_A19_U1 : IN std_logic;
             M68000_A18_U1 : IN std_logic;
             M68000_A17_U1 : IN std_logic;
             M68000_A16_U1 : IN std_logic;
             M68000_A15_U1 : IN std_logic;
             M68000_A14_U1 : IN std_logic;
             M68000_A13_U1 : IN std_logic;
             M68000_A12_U1 : IN std_logic;
             ISB_16_ADD_INT_SIGNAL_8000 : OUT std_logic;
             ISB_16_ADD_INT_SIGNAL_0 : OUT std_logic );
end ISB_16_ADD_INT;

architecture ONLY of ISB_16_ADD_INT is
begin
     ISB_16_ADD_INT_SIGNAL_8000 <=
                (
                    (
                      not M68000_A23_U1
                    )
                and (
                      not M68000_A22_U1
                    )
                and (
                      not M68000_A21_U1
                    )
                and (
                      not M68000_A20_U1
                    )
                and (
                      not M68000_A19_U1
                    )
                and (
                      not M68000_A18_U1
                    )
                and (
                      not M68000_A17_U1
                    )
                and (
                      not M68000_A16_U1
                    )
                and (
                     M68000_A15_U1
                    )
                and (
                      not M68000_A14_U1
                    )
                and (
                      not M68000_A13_U1
                    )
                and (
                      not M68000_A12_U1
                    )
                ) after 3*TPD;
--   Complexity was =  14
     ISB_16_ADD_INT_SIGNAL_0 <=
                (
                    (
                      not M68000_A23_U1
                    )
                and (
                      not M68000_A22_U1
                    )
                and (
                      not M68000_A21_U1
                    )
                and (
                      not M68000_A20_U1
                    )
                and (
                      not M68000_A19_U1
                    )
                and (
                      not M68000_A18_U1
                    )
                and (
                      not M68000_A17_U1
                    )
                and (
                      not M68000_A16_U1
                    )
                and (
                      not M68000_A15_U1
                    )
                and (
                      not M68000_A14_U1
                    )
                and (
                      not M68000_A13_U1
                    )
                and (
                      not M68000_A12_U1
                    )
                ) after 3*TPD;
--   Complexity was =  14
end ONLY;

entity ISB_10_WORD_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_UDS_U1 : IN std_logic;
             M68000_LDS_U1 : IN std_logic;
             ISB_10_WORD_INT_SIGNAL_8_15 : OUT std_logic;
             ISB_10_WORD_INT_SIGNAL_0_7 : OUT std_logic );
end ISB_10_WORD_INT;

architecture ONLY of ISB_10_WORD_INT is
begin
     ISB_10_WORD_INT_SIGNAL_8_15 <=
                (
                      not M68000_UDS_U1
                ) after TPD;
--   Complexity was =  2
     ISB_10_WORD_INT_SIGNAL_0_7 <=
                (
                      not M68000_LDS_U1
                ) after TPD;
--   Complexity was =  2
end ONLY;

entity ISB_4_REQUEST_INT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_AS_U1 : IN std_logic;
             M68000_UDS_U1 : IN std_logic;
             M68000_LDS_U1 : IN std_logic;
             ISB_4_REQUEST_INT_SIGNAL : OUT std_logic );
end ISB_4_REQUEST_INT;

architecture ONLY of ISB_4_REQUEST_INT is
begin
     ISB_4_REQUEST_INT_SIGNAL <=
                (( not M68000_AS_U1 ) and
 (( not M68000_UDS_U1 ) or ( not M68000_LDS_U1 ))
                ) after TPD;
--   Complexity was =  6
end ONLY;


-- END of vhdl code for IB_1_RW_CONNECT
-- Generated on: 15:02:57 8-2-1997      Version 1.0
```

E.2 VHDL IB for Design Example

Note: some address and data signals have been deleted for brevity.

```
-- START of vhdl code for IB_1_RW_CONNECT
-- Device 1 : IB_1_RW_CONNECT
-- Generated on: 15:00:20 8-2-1997      Version 1.0


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library DAMELIB;

entity IB_1_RW_CONNECT is
        generic (
             TPD : time;
             TPD_EN : time;
             TPD_EDGE : time );
        port (
             M68000_AS_U1 : IN std_logic;
             M68000_LDS_U1 : IN std_logic;
             M68000_UDS_U1 : IN std_logic;
             M68000_A12_U1 : IN std_logic;
             M68000_A23_U1 : IN std_logic;
             M68000_RW_U1 : IN std_logic;
             M68000_FC0_U1 : IN std_logic;
             M68000_FC1_U1 : IN std_logic;
             M68000_FC2_U1 : IN std_logic;
             SYS_CLOCK : IN std_logic;
             SYS_RESET : IN std_logic;
             M68000_A1_U1 : IN std_logic;
             M68000_A11_U1 : IN std_logic;
             M68000_DTAK_U1 : OUT std_logic;
             M6116_OE_U2345 : OUT std_logic;
             M6116_CE_U5 : OUT std_logic;
             M6116_CE_U4 : OUT std_logic;
             M6116_CE_U3 : OUT std_logic;
             M6116_CE_U2 : OUT std_logic;
             M6116_WR_U2345 : OUT std_logic;
             M6116_A0_U2345 : OUT std_logic;
             M6116_A10_U2345 : OUT std_logic;
             M68000_D0_U1 : INOUT std_logic;
             M6116_D0_U24 : INOUT std_logic;
             M68000_D8_U1 : INOUT std_logic;
             M6116_D0_U35 : INOUT std_logic);
end IB_1_RW_CONNECT;

architecture ONLY of IB_1_RW_CONNECT is
   use WORK.IB_IB_1_RW_CONNECT_PACKAGE.ALL;
   signal ISB_4_REQUEST_INT_SIGNAL : std_logic;
   signal ISB_10_WORD_INT_SIGNAL_8_15 : std_logic;
   signal ISB_10_WORD_INT_SIGNAL_0_7 : std_logic;
   signal ISB_16_ADD_INT_SIGNAL_8000 : std_logic;
   signal ISB_16_ADD_INT_SIGNAL_0 : std_logic;
   signal ISB_31_BLOCK_ADD_INT_SIGNAL : std_logic;
   signal ISB_35_WRITE_INT_SIGNAL : std_logic;
   signal ISB_38_READ_INT_SIGNAL : std_logic;
   signal ISB_44_TYPE_INT_SIGNAL : std_logic;
   signal ISB_49_ACCESS_INT_SIGNAL : std_logic;
   signal ISB_55_DATA_ACC_EN_INT_SIGNAL : std_logic;
   signal ISB_58_ACC_DEL_INT_SIGNAL : std_logic;
   signal ISB_58_ACC_DEL_INT_SIGNAL_S_1 : std_logic;
   signal ISB_61_DELAY_INT_SIGNAL : std_logic;
begin
   PART_1 : ISB_4_REQUEST_INT
      generic map (
             TPD => TPD,
             TPD_EN => TPD_EN,
             TPD_EDGE => TPD_EDGE )
      port map (
             M68000_AS_U1 => M68000_AS_U1,
             M68000_UDS_U1 => M68000_UDS_U1,
             M68000_LDS_U1 => M68000_LDS_U1,
             ISB_4_REQUEST_INT_SIGNAL => ISB_4_REQUEST_INT_SIGNAL );
   PART_2 : ISB_10_WORD_INT
      generic map (
             TPD => TPD,
             TPD_EN => TPD_EN,
             TPD_EDGE => TPD_EDGE )
      port map (
             M68000_UDS_U1 => M68000_UDS_U1,
             M68000_LDS_U1 => M68000_LDS_U1,
             ISB_10_WORD_INT_SIGNAL_8_15 => ISB_10_WORD_INT_SIGNAL_8_15,
             ISB_10_WORD_INT_SIGNAL_0_7 => ISB_10_WORD_INT_SIGNAL_0_7 );
   PART_3 : ISB_16_ADD_INT
      generic map (
             TPD => TPD,
             TPD_EN => TPD_EN,
             TPD_EDGE => TPD_EDGE )
      port map (
             M68000_A23_U1 => M68000_A23_U1,
             M68000_A22_U1 => M68000_A22_U1,
             M68000_A21_U1 => M68000_A21_U1,
             M68000_A20_U1 => M68000_A20_U1,
             M68000_A19_U1 => M68000_A19_U1,
             M68000_A18_U1 => M68000_A18_U1,
             M68000_A17_U1 => M68000_A17_U1,
             M68000_A16_U1 => M68000_A16_U1,
             M68000_A15_U1 => M68000_A15_U1,
             M68000_A14_U1 => M68000_A14_U1,
             M68000_A13_U1 => M68000_A13_U1,
             M68000_A12_U1 => M68000_A12_U1,
             ISB_16_ADD_INT_SIGNAL_8000 => ISB_16_ADD_INT_SIGNAL_8000,
             ISB_16_ADD_INT_SIGNAL_0 => ISB_16_ADD_INT_SIGNAL_0 );
   PART_4 : ISB_31_BLOCK_ADD_INT
      generic map (
             TPD => TPD,
             TPD_EN => TPD_EN,
             TPD_EDGE => TPD_EDGE )
      port map (
             ISB_16_ADD_INT_SIGNAL_8000 => ISB_16_ADD_INT_SIGNAL_8000,
             ISB_16_ADD_INT_SIGNAL_0 => ISB_16_ADD_INT_SIGNAL_0,
             ISB_31_BLOCK_ADD_INT_SIGNAL => ISB_31_BLOCK_ADD_INT_SIGNAL );
```

```
PART_5 : ISB_35_WRITE_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          M68000_RW_U1 => M68000_RW_U1,
          ISB_35_WRITE_INT_SIGNAL => ISB_35_WRITE_INT_SIGNAL );
PART_6 : ISB_38_READ_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          M68000_RW_U1 => M68000_RW_U1,
          ISB_38_READ_INT_SIGNAL => ISB_38_READ_INT_SIGNAL );
PART_7 : ISB_44_TYPE_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          M68000_FC2_U1 => M68000_FC2_U1,
          M68000_FC1_U1 => M68000_FC1_U1,
          M68000_FC0_U1 => M68000_FC0_U1,
          ISB_44_TYPE_INT_SIGNAL => ISB_44_TYPE_INT_SIGNAL );
PART_8 : ISB_49_ACCESS_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_31_BLOCK_ADD_INT_SIGNAL => ISB_31_BLOCK_ADD_INT_SIGNAL,
          ISB_4_REQUEST_INT_SIGNAL => ISB_4_REQUEST_INT_SIGNAL,
          ISB_44_TYPE_INT_SIGNAL => ISB_44_TYPE_INT_SIGNAL,
          ISB_49_ACCESS_INT_SIGNAL => ISB_49_ACCESS_INT_SIGNAL );
PART_9 : ISB_55_DATA_ACC_EN_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_49_ACCESS_INT_SIGNAL => ISB_49_ACCESS_INT_SIGNAL,
          ISB_55_DATA_ACC_EN_INT_SIGNAL => ISB_55_DATA_ACC_EN_INT_SIGNAL );
PART_10 : ISB_58_ACC_DEL_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_49_ACCESS_INT_SIGNAL => ISB_49_ACCESS_INT_SIGNAL,
          ISB_58_ACC_DEL_INT_SIGNAL => ISB_58_ACC_DEL_INT_SIGNAL );
PART_11 : ISB_63_CONV_SS
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_58_ACC_DEL_INT_SIGNAL => ISB_58_ACC_DEL_INT_SIGNAL,
          SYS_RESET => SYS_RESET,
          SYS_CLOCK => SYS_CLOCK,
          ISB_58_ACC_DEL_INT_SIGNAL_S_1 => ISB_58_ACC_DEL_INT_SIGNAL_S_1 );
PART_12 : ISB_61_DELAY_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_58_ACC_DEL_INT_SIGNAL_S_1 => ISB_58_ACC_DEL_INT_SIGNAL_S_1,
          ISB_61_DELAY_INT_SIGNAL => ISB_61_DELAY_INT_SIGNAL );
PART_13 : ISB_65_M68000_DTAK_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_61_DELAY_INT_SIGNAL => ISB_61_DELAY_INT_SIGNAL,
          M68000_DTAK_U1 => M68000_DTAK_U1 );
PART_14 : ISB_67_M6116_OE_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_38_READ_INT_SIGNAL => ISB_38_READ_INT_SIGNAL,
          M6116_OE_U2345 => M6116_OE_U2345 );
PART_15 : ISB_69_M6116_CE_INT_8000_8_15
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_44_TYPE_INT_SIGNAL => ISB_44_TYPE_INT_SIGNAL,
          ISB_16_ADD_INT_SIGNAL_8000 => ISB_16_ADD_INT_SIGNAL_8000,
          ISB_10_WORD_INT_SIGNAL_8_15 => ISB_10_WORD_INT_SIGNAL_8_15,
          ISB_4_REQUEST_INT_SIGNAL => ISB_4_REQUEST_INT_SIGNAL,
          M6116_CE_U5 => M6116_CE_U5 );
PART_16 : ISB_69_M6116_CE_INT_8000_0_7
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_44_TYPE_INT_SIGNAL => ISB_44_TYPE_INT_SIGNAL,
          ISB_16_ADD_INT_SIGNAL_8000 => ISB_16_ADD_INT_SIGNAL_8000,
          ISB_10_WORD_INT_SIGNAL_0_7 => ISB_10_WORD_INT_SIGNAL_0_7,
          ISB_4_REQUEST_INT_SIGNAL => ISB_4_REQUEST_INT_SIGNAL,
          M6116_CE_U4 => M6116_CE_U4 );
PART_17 : ISB_69_M6116_CE_INT_0_8_15
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_44_TYPE_INT_SIGNAL => ISB_44_TYPE_INT_SIGNAL,
          ISB_16_ADD_INT_SIGNAL_0 => ISB_16_ADD_INT_SIGNAL_0,
          ISB_10_WORD_INT_SIGNAL_8_15 => ISB_10_WORD_INT_SIGNAL_8_15,
          ISB_4_REQUEST_INT_SIGNAL => ISB_4_REQUEST_INT_SIGNAL,
          M6116_CE_U3 => M6116_CE_U3 );
PART_18 : ISB_69_M6116_CE_INT_0_0_7
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_44_TYPE_INT_SIGNAL => ISB_44_TYPE_INT_SIGNAL,
          ISB_16_ADD_INT_SIGNAL_0 => ISB_16_ADD_INT_SIGNAL_0,
          ISB_10_WORD_INT_SIGNAL_0_7 => ISB_10_WORD_INT_SIGNAL_0_7,
          ISB_4_REQUEST_INT_SIGNAL => ISB_4_REQUEST_INT_SIGNAL,
          M6116_CE_U2 => M6116_CE_U2 );
PART_19 : ISB_71_M6116_WR_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
          ISB_35_WRITE_INT_SIGNAL => ISB_35_WRITE_INT_SIGNAL,
          M6116_WR_U2345 => M6116_WR_U2345 );
PART_20 : ISB_89_M68000_D0_INT
   generic map (
          TPD => TPD,
          TPD_EN => TPD_EN,
          TPD_EDGE => TPD_EDGE )
   port map (
             M68000_D0_U1 => M68000_D0_U1,
             ISB_55_DATA_ACC_EN_INT_SIGNAL => ISB_55_DATA_ACC_EN_INT_SIGNAL,
             ISB_35_WRITE_INT_SIGNAL => ISB_35_WRITE_INT_SIGNAL,
             M6116_D0_U24 => M6116_D0_U24 );
   PART_28 : ISB_105_M6116_D0_INT
      generic map (
             TPD => TPD,
             TPD_EN => TPD_EN,
             TPD_EDGE => TPD_EDGE )
      port map (
             M6116_D0_U24 => M6116_D0_U24,
             ISB_55_DATA_ACC_EN_INT_SIGNAL => ISB_55_DATA_ACC_EN_INT_SIGNAL,
             ISB_38_READ_INT_SIGNAL => ISB_38_READ_INT_SIGNAL,
             M68000_D0_U1 => M68000_D0_U1 );
   PART_36 : ISB_121_M68000_D8_INT
      generic map (
             TPD => TPD,
             TPD_EN => TPD_EN,
             TPD_EDGE => TPD_EDGE )
      port map (
             M68000_D8_U1 => M68000_D8_U1,
             ISB_55_DATA_ACC_EN_INT_SIGNAL => ISB_55_DATA_ACC_EN_INT_SIGNAL,
             ISB_35_WRITE_INT_SIGNAL => ISB_35_WRITE_INT_SIGNAL,
             M6116_D0_U35 => M6116_D0_U35 );
   PART_44 : ISB_137_M6116_D0_INT
      generic map (
             TPD => TPD,
             TPD_EN => TPD_EN,
             TPD_EDGE => TPD_EDGE )
      port map (
             M6116_D0_U35 => M6116_D0_U35,
             ISB_55_DATA_ACC_EN_INT_SIGNAL => ISB_55_DATA_ACC_EN_INT_SIGNAL,
             ISB_38_READ_INT_SIGNAL => ISB_38_READ_INT_SIGNAL,
             M68000_D8_U1 => M68000_D8_U1 );
   PART_52 : ISB_250_M68000_A1_INT
      generic map (
             TPD => TPD,
             TPD_EN => TPD_EN,
             TPD_EDGE => TPD_EDGE )
      port map (
             M68000_A1_U1 => M68000_A1_U1,
             M6116_A0_U2345 => M6116_A0_U2345 );
   PART_62 : ISB_270_M68000_A11_INT
      generic map (
             TPD => TPD,
             TPD_EN => TPD_EN,
             TPD_EDGE => TPD_EDGE )
      port map (
             M68000_A11_U1 => M68000_A11_U1,
             M6116_A10_U2345 => M6116_A10_U2345 );
end ONLY;


-- END of vhdl code for IB_1_RW_CONNECT
-- Generated on: 15:02:57 8-2-1997      Version 1.0
```

E.3 VHDL Test Bench for Design Example

Note: some address and data signals have been deleted for brevity.

```
-- START of vhdl code for COMPLETE
-- Device 1 : COMPLETE
-- Generated on: 15:02:59 8-2-1997      Version 1.0
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library DAMELIB;

entity TEST_BENCH_SYSTEM is
end TEST_BENCH_SYSTEM;
```

```
architecture TEST_BENCH_SYSTEM_COMPLETE of TEST_BENCH_SYSTEM is
   constant SCALE : TIME := 1 ns;
   use WORK.INTERFACE_SYSTEM_BLOCK_PACKAGE.ALL;
   signal M68000_AS_U1 : std_logic;
   signal M68000_LDS_U1 : std_logic;
   signal M68000_UDS_U1 : std_logic;
   signal M68000_RW_U1 : std_logic;
   signal M68000_FC0_U1 : std_logic;
   signal M68000_FC1_U1 : std_logic;
   signal M68000_FC2_U1 : std_logic;
   signal SYS_CLOCK : std_logic;
   signal SYS_RESET : std_logic;
   signal M68000_DTAK_U1 : std_logic;
   signal M6116_OE_U2345 : std_logic;
   signal M6116_CE_U5 : std_logic;
   signal M6116_CE_U4 : std_logic;
   signal M6116_CE_U3 : std_logic;
   signal M6116_CE_U2 : std_logic;
   signal M6116_WR_U2345 : std_logic;
   signal M68000_A_U1 : std_logic_vector(23 downto 1);
   signal M6116_D_U24 : std_logic_vector(7 downto 0);
   signal M68000_LD_U1 : std_logic_vector(7 downto 0);
   signal M6116_D_U35 : std_logic_vector(7 downto 0);
   signal M68000_UD_U1 : std_logic_vector(7 downto 0);
   signal M6116_A_U2345 : std_logic_vector(10 downto 0);
begin
   DUT : COMPLETE
      generic map (
             TPD => 3 ns,
             TPD_EN => 15 ns,
             TPD_EDGE => 2 ns )
      port map (
             M68000_AS_U1 => M68000_AS_U1,
             M68000_LDS_U1 => M68000_LDS_U1,
             M68000_UDS_U1 => M68000_UDS_U1,
             M68000_A12_U1 => M68000_A_U1(12),
             M68000_A23_U1 => M68000_A_U1(23),
             M68000_RW_U1 => M68000_RW_U1,
             M68000_FC0_U1 => M68000_FC0_U1,
             M68000_FC1_U1 => M68000_FC1_U1,
             M68000_FC2_U1 => M68000_FC2_U1,
             SYS_CLOCK => SYS_CLOCK,
             SYS_RESET => SYS_RESET,
             M68000_D0_U1 => M68000_LD_U1(0),
             M68000_D7_U1 => M68000_LD_U1(7),
             M6116_D0_U24 => M6116_D_U24(0),
             M6116_D7_U24 => M6116_D_U24(7),
             M68000_D8_U1 => M68000_UD_U1(0),
             M68000_D15_U1 => M68000_UD_U1(7),
             M6116_D0_U35 => M6116_D_U35(0),
             M6116_D7_U35 => M6116_D_U35(7),
             M68000_A1_U1 => M68000_A_U1(1),
             M68000_A11_U1 => M68000_A_U1(11),
             M68000_DTAK_U1 => M68000_DTAK_U1,
             M6116_OE_U2345 => M6116_OE_U2345,
             M6116_CE_U5 => M6116_CE_U5,
             M6116_CE_U4 => M6116_CE_U4,
             M6116_CE_U3 => M6116_CE_U3,
             M6116_CE_U2 => M6116_CE_U2,
             M6116_WR_U2345 => M6116_WR_U2345,
             M6116_A0_U2345 => M6116_A_U2345(0),
             M6116_A10_U2345 => M6116_A_U2345(10) );
   STIMULUS_1 : process
    begin
--      it-delay: 270  address: 0
        M68000_AS_U1 <= ‘1’;  wait for 200 * SCALE;
        M68000_AS_U1 <= ‘0’;  wait for 270 * SCALE;
        M68000_AS_U1 <= ‘1’;  wait for 200 * SCALE;

--      it-delay: 270  address: 0
        M68000_AS_U1 <= ‘1’;  wait for 200 * SCALE;
        M68000_AS_U1 <= ‘0’;  wait for 270 * SCALE;
        M68000_AS_U1 <= ‘1’;  wait for 200 * SCALE;
   end process STIMULUS_1;

   STIMULUS_2 : process
    begin
--      it-delay: 270  address: 0
        M68000_LDS_U1 <= ‘1’;  wait for 250 * SCALE;
        M68000_LDS_U1 <= ‘0’;  wait for 220 * SCALE;
        M68000_LDS_U1 <= ‘1’;  wait for 200 * SCALE;
--      it-delay: 270  address: 0
        M68000_LDS_U1 <= ‘1’;  wait for 200 * SCALE;
        M68000_LDS_U1 <= ‘1’;  wait for 270 * SCALE;
        M68000_LDS_U1 <= ‘1’;  wait for 200 * SCALE;
   end process STIMULUS_2;

   STIMULUS_3 : process
    begin
--      it-delay: 270  address: 0
        M68000_UDS_U1 <= ‘1’;  wait for 250 * SCALE;
        M68000_UDS_U1 <= ‘0’;  wait for 220 * SCALE;
        M68000_UDS_U1 <= ‘1’;  wait for 200 * SCALE;
--      it-delay: 270  address: 0
        M68000_UDS_U1 <= ‘1’;  wait for 200 * SCALE;
        M68000_UDS_U1 <= ‘0’;  wait for 270 * SCALE;
        M68000_UDS_U1 <= ‘1’;  wait for 200 * SCALE;
   end process STIMULUS_3;

   STIMULUS_4 : process
    begin
--      it-delay: 270  address: 0
        M68000_RW_U1 <= ‘1’;  wait for 140 * SCALE;
        M68000_RW_U1 <= ‘0’;  wait for 370 * SCALE;
        M68000_RW_U1 <= ‘1’;  wait for 160 * SCALE;
--      it-delay: 270  address: 0
        M68000_RW_U1 <= ‘1’;  wait for 140 * SCALE;
        M68000_RW_U1 <= ‘1’;  wait for 370 * SCALE;
        M68000_RW_U1 <= ‘1’;  wait for 160 * SCALE;
   end process STIMULUS_4;

   STIMULUS_5 : process
    begin
--      it-delay: 270  address: 0
        M68000_FC0_U1 <= ‘0’;  wait for 140 * SCALE;
        M68000_FC0_U1 <= ‘0’;  wait for 370 * SCALE;
        M68000_FC0_U1 <= ‘0’;  wait for 160 * SCALE;
--      it-delay: 270  address: 0
        M68000_FC0_U1 <= ‘0’;  wait for 140 * SCALE;
        M68000_FC0_U1 <= ‘0’;  wait for 370 * SCALE;
        M68000_FC0_U1 <= ‘0’;  wait for 160 * SCALE;
   end process STIMULUS_5;

   STIMULUS_6 : process
    begin
--      it-delay: 270  address: 0
        M68000_FC1_U1 <= ‘0’;  wait for 140 * SCALE;
        M68000_FC1_U1 <= ‘1’;  wait for 370 * SCALE;
        M68000_FC1_U1 <= ‘0’;  wait for 160 * SCALE;
--      it-delay: 270  address: 0
        M68000_FC1_U1 <= ‘0’;  wait for 140 * SCALE;
        M68000_FC1_U1 <= ‘1’;  wait for 370 * SCALE;
        M68000_FC1_U1 <= ‘0’;  wait for 160 * SCALE;
   end process STIMULUS_6;

   STIMULUS_7 : process
    begin
--      it-delay: 270  address: 0
        M68000_FC2_U1 <= ‘0’;  wait for 140 * SCALE;
        M68000_FC2_U1 <= ‘0’;  wait for 370 * SCALE;
        M68000_FC2_U1 <= ‘0’;  wait for 160 * SCALE;
--      it-delay: 270  address: 0
        M68000_FC2_U1 <= ‘0’;  wait for 140 * SCALE;
        M68000_FC2_U1 <= ‘0’;  wait for 370 * SCALE;
        M68000_FC2_U1 <= ‘0’;  wait for 160 * SCALE;
   end process STIMULUS_7;

   STIMULUS_CLOCK : process
    begin
        SYS_CLOCK <= ‘0’;  wait for 25 ns;
        SYS_CLOCK <= ‘1’;  wait for 25 ns;
        SYS_CLOCK <= ‘0’;  wait for 0 ns;
   end process STIMULUS_CLOCK;

   STIMULUS_RESET : process
    begin
        SYS_RESET <= ‘0’;  wait for 1 ns;
        SYS_RESET <= ‘1’;  wait for 14 ns;
        SYS_RESET <= ‘0’;  wait for 1000000 ns;
   end process STIMULUS_RESET;

   STIMULUS_1000 : process
    begin
--      it-delay: 270  address: 0
--      M68000_LD_U1      76543210
        M68000_LD_U1 <= (“ZZZZZZZZ”);  wait for 170 * SCALE;
        M68000_LD_U1 <= (“00111100”);  wait for 330 * SCALE;
        M68000_LD_U1 <= (“ZZZZZZZZ”);  wait for 170 * SCALE;
--      it-delay: 270  address: 0
--      M68000_LD_U1      76543210
        M68000_LD_U1 <= (“ZZZZZZZZ”);  wait for 170 * SCALE;
        M68000_LD_U1 <= (“ZZZZZZZZ”);  wait for 330 * SCALE;
        M68000_LD_U1 <= (“ZZZZZZZZ”);  wait for 170 * SCALE;
   end process STIMULUS_1000;

   STIMULUS_1001 : process
    begin
--      it-delay: 270  address: 0
--      M6116_D_U24      76543210
        M6116_D_U24 <= (“ZZZZZZZZ”);  wait for 200 * SCALE;
        M6116_D_U24 <= (“ZZZZZZZZ”);  wait for 150 * SCALE;
        M6116_D_U24 <= (“ZZZZZZZZ”);  wait for 135 * SCALE;
        M6116_D_U24 <= (“ZZZZZZZZ”);  wait for 185 * SCALE;
--      it-delay: 270  address: 0
--      M6116_D_U24      76543210
        M6116_D_U24 <= (“ZZZZZZZZ”);  wait for 200 * SCALE;
        M6116_D_U24 <= (“ZZZZZZZZ”);  wait for 150 * SCALE;
        M6116_D_U24 <= (“00100100”);  wait for 135 * SCALE;
        M6116_D_U24 <= (“ZZZZZZZZ”);  wait for 185 * SCALE;

   end process STIMULUS_1001;

   STIMULUS_1002 : process
    begin
--      it-delay: 270  address: 0
--      M68000_UD_U1      76543210
        M68000_UD_U1 <= (“ZZZZZZZZ”);  wait for 170 * SCALE;
        M68000_UD_U1 <= (“00100000”);  wait for 330 * SCALE;
        M68000_UD_U1 <= (“ZZZZZZZZ”);  wait for 170 * SCALE;
--      it-delay: 270  address: 0
--      M68000_UD_U1      76543210
        M68000_UD_U1 <= (“ZZZZZZZZ”);  wait for 170 * SCALE;
        M68000_UD_U1 <= (“ZZZZZZZZ”);  wait for 330 * SCALE;
        M68000_UD_U1 <= (“ZZZZZZZZ”);  wait for 170 * SCALE;
   end process STIMULUS_1002;

   STIMULUS_1003 : process
    begin
--      it-delay: 270  address: 0
--      M6116_D_U35      76543210
        M6116_D_U35 <= (“ZZZZZZZZ”);  wait for 200 * SCALE;
        M6116_D_U35 <= (“ZZZZZZZZ”);  wait for 150 * SCALE;
        M6116_D_U35 <= (“ZZZZZZZZ”);  wait for 135 * SCALE;
        M6116_D_U35 <= (“ZZZZZZZZ”);  wait for 185 * SCALE;
--      it-delay: 270  address: 0
--      M6116_D_U35      76543210
```

```
        M6116_D_U35 <= (“ZZZZZZZZ”);  wait for 200 * SCALE;
        M6116_D_U35 <= (“ZZZZZZZZ”);  wait for 150 * SCALE;
        M6116_D_U35 <= (“01000000”);  wait for 135 * SCALE;
        M6116_D_U35 <= (“ZZZZZZZZ”);  wait for 185 * SCALE;
   end process STIMULUS_1003;

   STIMULUS_1004 : process
    begin
--      it-delay: 270  address: 0
--      M68000_A_U1      32109876543210987654321
        M68000_A_U1 <= (“11111111111111111111111”);  wait for 170 * SCALE;
        M68000_A_U1 <= (“00000000100000000010010”);  wait for 330 * SCALE;
        M68000_A_U1 <= (“11111111111111111111111”);  wait for 170 * SCALE;
--      it-delay: 270  address: 0
--      M68000_A_U1      32109876543210987654321
        M68000_A_U1 <= (“11111111111111111111111”);  wait for 170 * SCALE;
        M68000_A_U1 <= (“00000000000000100011010”);  wait for 330 * SCALE;
        M68000_A_U1 <= (“11111111111111111111111”);  wait for 170 * SCALE;
   end process STIMULUS_1004;
end TEST_BENCH_SYSTEM_COMPLETE;

configuration TEST_BENCH_SYSTEM_CONFIG of TEST_BENCH_SYSTEM is
   for TEST_BENCH_SYSTEM_COMPLETE
   end for;
end TEST_BENCH_SYSTEM_CONFIG;
-- END of vhdl code for COMPLETE
```

Appendix F
Other Interface Design Examples

F.1 Interface Design Example: i8086

The Interface Designer was given the following design problem:

- Microprocessor: Intel i8086A-2 (8Mhz)
- RAM: Four RCA cmd6116-3 2Kx8 (150ns) with 16-bit datapath interface mapped at address 0x00000 and 0x08000 in the 20-bit address space
- ROM: Two Mostek etc2716-1 2Kx8 (350ns) EPROMs with 16-bit datapath interface mapped at address 0x0e400
- PIO: Intel i8255a (400ns) parallel IO device with 8-bit datapath interface mapped at address 0x0c000

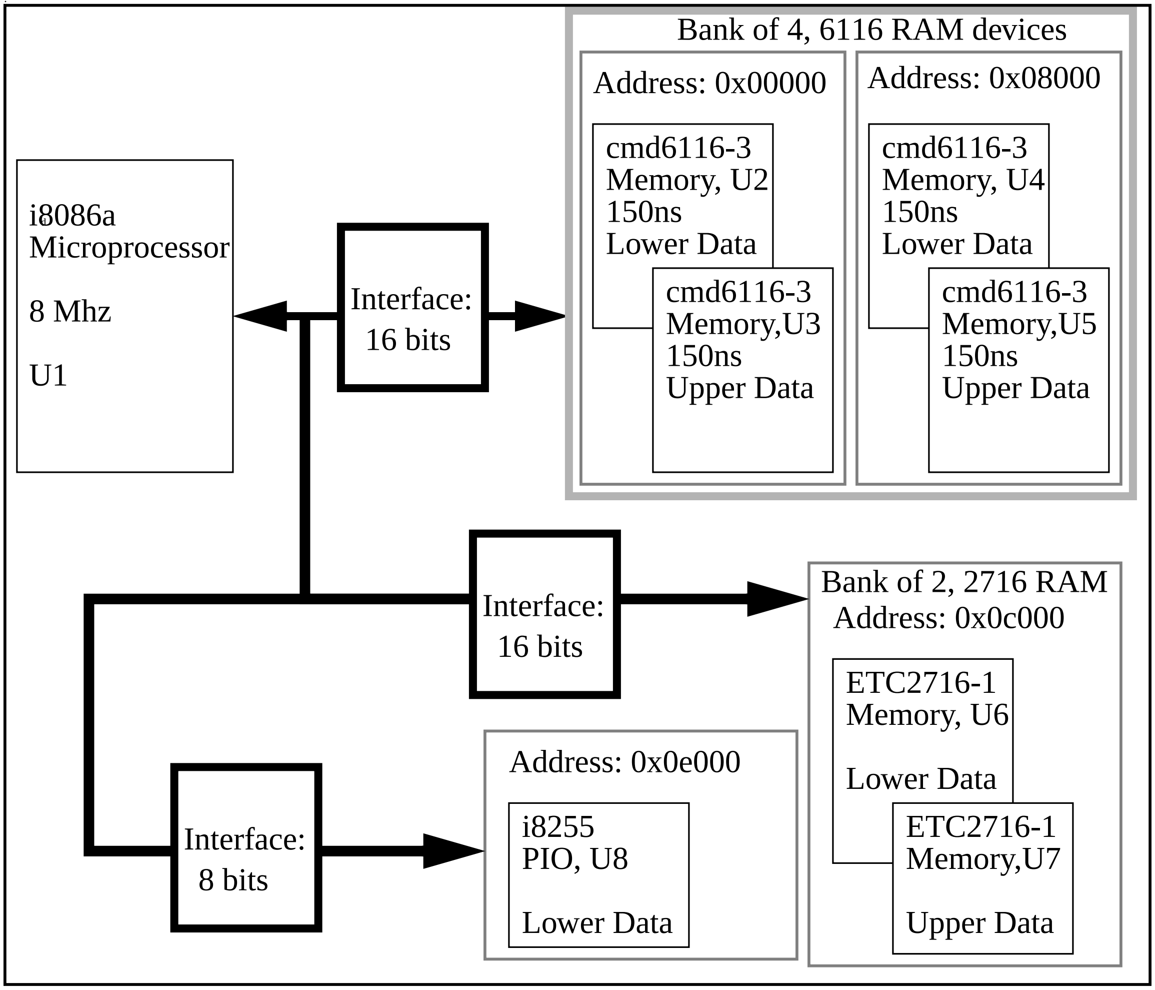


FIGURE F-1. i8086 System

Some of the features illustrated by this design example are: microprocessor system design using more than one type of component, address decoding using multiplexed address signals, interfacing 16-bit memory and 8-bit memory mapped IO devices to a 16-bit microprocessor allowing both 16-bit and 8-bit data transfer, interfacing components of different speed using wait signal generation, and connection of multiplexed to non-multiplexed address signals. A design problem specification block diagram is shown in Figure F-1. The block diagram indicates that separate IBs must be generated for the RAM, ROM and the PIO.

The VHDL simulation output for this interface is shown in Figure F-2. The simulation shows three data transfer cycles: a 16-bit write cycle to the RAM at address 0x08024, an 8-bit read cycle from the ROM at address 0x0c61a and an 8-bit read cycle from the PIO at address 0x0e420. The timing diagram illustrates the correct operation of the interface, specifically the correct generation of enable and write signals, demultiplexing of the address signals and generation of the `WAIT` signal. The timing parameters from the simulation output were verified against the timing parameters given by the component manufacturers and were found to provide a positive margin, indicating a valid design. This design shows that the Interface Designer is able to correctly design a more complex microprocessor system utilizing more than one type of different speed component.

F.2 Interface Design Example: 68020

The Interface Designer was given the following design problem:

- Microprocessor: Motorola mc68020-12.5 (12.5Mhz)
- RAM: Four Motorola mcm6164-45 8kx8 (45ns) with 32-bit datapath interface mapped at address 0x00008000 in the 32-bit address space
- ROM: Two Intel 27128a-2 16kx8 (200ns) EPROMs with 16-bit datapath interface mapped at address 0x00000000
- RAM: one Motorola mcm6810 (450ns) 128 byte scratch pad RAM mapped at address 0x0001f000

This design example illustrates a design that uses the dynamic bus sizing feature of the 68020: the RAM, ROM and PIO to 68020 data transfer interface width are 32-bit, 16-bit and 8-bit respectively using memory devices of different speed. The VHDL simulation of the interface as shown in Figure F-3, illustrates data transfer cycles for each of the three different memories.

The `U1 DTAK0/1` signals are both asserted for the 6164 RAM write cycle (t=100ns) indicating that the RAM is capable of transferring 32-bit data. However only 24 bits are

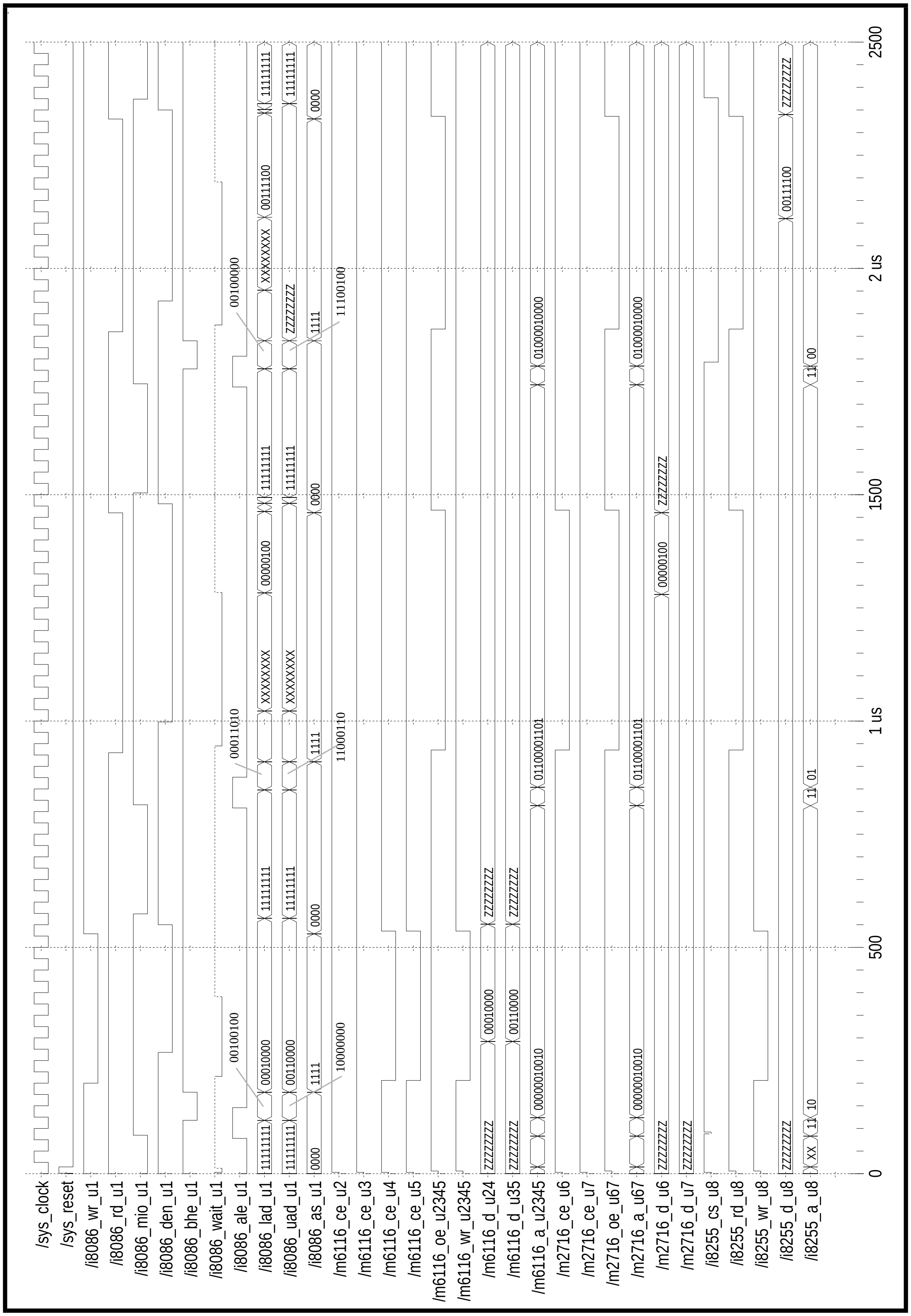


FIGURE F-2. i8086 Design - VHDL Simulation

transferred, as can be seen by the activation of 3 of the 6164 RAM CE1 signals (/ m6164_CE1_U2, /m6164_CE1_U3, /m6164_CE1_U4), since the 32-bit write cycle is on an odd address (0x000080F5).

The U1 DTAK1 signal is asserted for the 27128 RAM read cycle (t=750ns) indicating that the ROM is capable of transferring 16-bit data. However only 8 bits are transferred, as can be seen by the activation of the /m27128_CE_U6 signal, since this 16-bit read cycle is on an odd address (0x00000027).

For the 6810 read cycle at t=1470ns only the U1 DTAK0 signal is asserted indicating that scratch pad RAM is capable of transferring 8-bit data. The 8 bits are transferred from the /m6810_d_u8 signals to the /m68020_uud_u1 signals as can be seen in the simulation timing diagram (note: the naming convention for the 68020 uses ‘uud’ for D24:D31, ‘umd’ for D16:D23, ‘lmd’ for D8:D15 and lld for D0:D7).

The DTAK0/1 signals of the 68020 serve two purposes in this design: they indicate the data path width of the slave device and they are also used as the handshake signal that terminates the data transfer. The VHDL simulation shows that the asserted DTAK0/1 event is delayed depending on the speed of the slave component, about 100ns for the 6164, 250ns for the 27128, and about 450ns for the 6810, as required from the component manufacturer’s data sheet, indicating a valid interface.

The VHDL code was manually inspected to verify that the correct address signals on the slave devices are connected to the address signals on the microprocessor. As expected, the 32-bit 6164 RAM memory bank A0:A12 signals are connected to the 68020 A2:A14 signals, the 16-bit 27128 ROM memory bank A0:A13 signals are connected to the 68020 A1:A14 signals and the 8-bit 6810 scratch pad RAM A0:A6 signals are connected to the 68020 A0:A6 signals. The result of these connections can also be observed in the simulation of Figure F-3.

The timing diagram illustrates the correct operation of the interface, specifically the correct generation of enable and write signals for the dynamically sized data bus and generation of the DTAK signals. The timing parameters from the simulation output were verified against the timing parameters given by the component manufacturers and were found to provide a positive margin, indicating a valid design.

This design example illustrates a more difficult design. Unless intimately familiar with the data sheet of the 68020, a human designer will have to spend time investigating the dynamic bus sizing signals and the signal protocol of the 68020. The human designer has to become familiar with the relationship between the DTACK0, DTACK1, SIZE0,

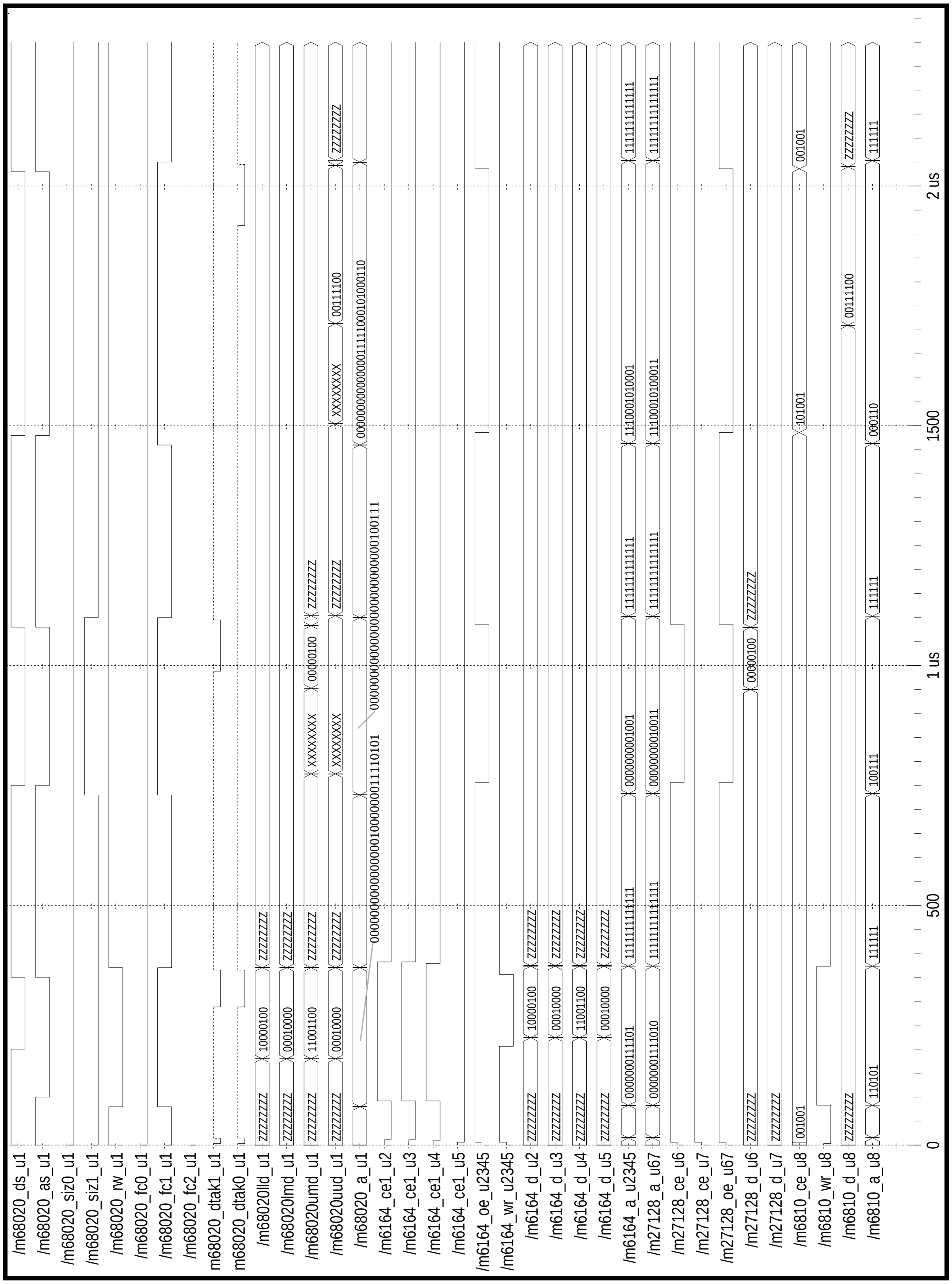


FIGURE F-3. 68020 Design - VHDL Simulation

SIZE, A0 and A1 signals and the 32-bit data bus for 32-bit, 16-bit and 8-bit data transfers, and then translate the information learned into a working design. The Interface Designer allows the design to be generated automatically, reducing the design time and also eliminating errors introduced into the design due to mistakes in the interpretation or transcription of information from the data books.

F.3 6809 Interface Example

The Interface Designer was given the following design problem:

- Microprocessor: Motorola mcm6809e (1Mhz)
- IO: one Motorola mcm6821 (450ns) Parallel IO Controller mapped at address 0xec00.
- IO: one Motorola mcm6845 (450ns) CRT Controller mapped at address 0xe800.
- IO: one Motorola mcm6850 (450ns) UART mapped at address 0xe000.

This design example tests the design of a simple 8 bit microprocessor to IO device interface. In the VHDL simulation shown in Figure F-4. A data transfer cycle for each of the three different IO devices is shown. The 6809 does not have an acknowledge signal or a wait signal, so fixed E signal pulse width of 450ns is used in the simulation.

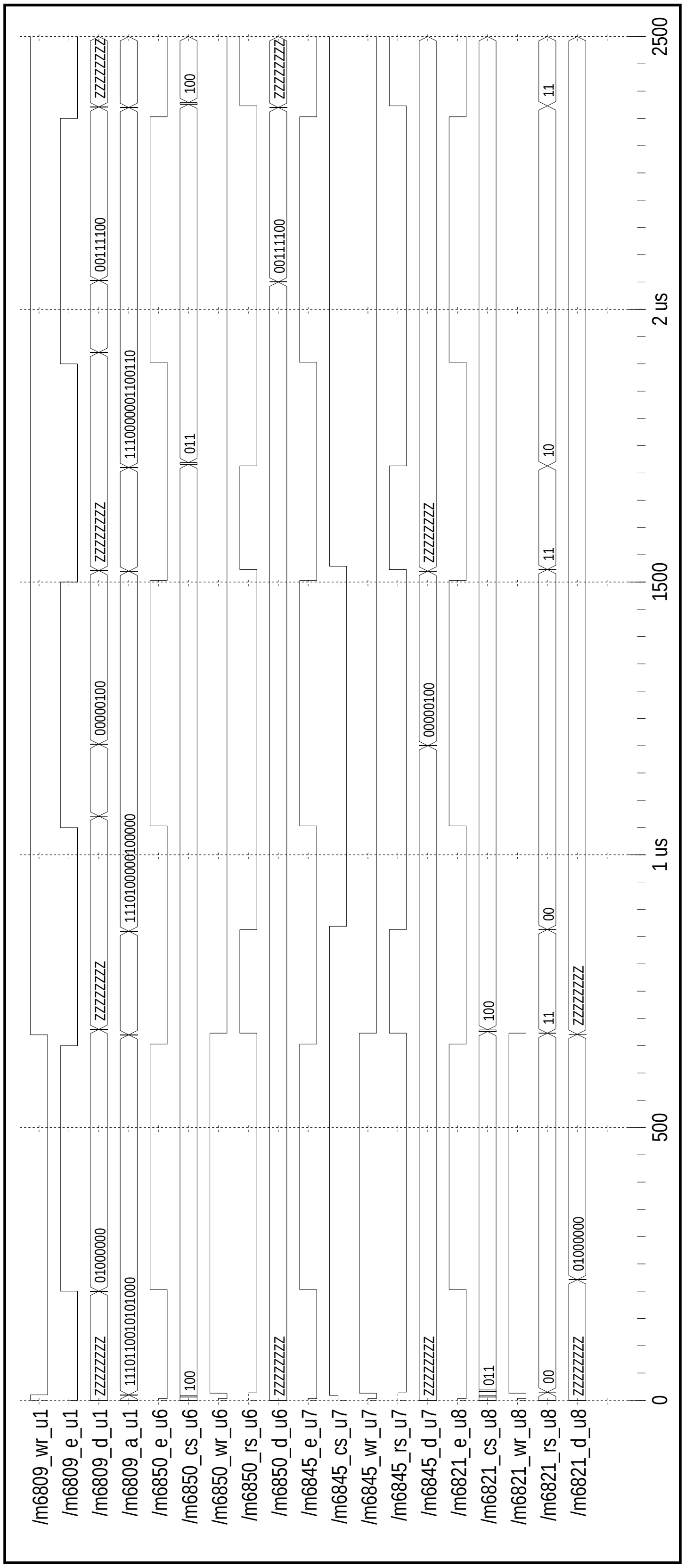


FIGURE F-4. m6809 Design - VHDL Simulation

F.4 t32020 Interface Example

The Interface Designer was given the following design problem:

- Microprocessor: Texas Instruments tms32020 (20Mhz).
- RAM: Two Motorola mcm6164-45 8kx8 (45ns) with 16-bit datapath interface
  Mapped at address 0x0000.
- ROM: Two Intel 2764a-1 16kx8 (180ns) EPROMs with 16-bit datapath interface
  Mapped at address 0xe000.

This design example tests the design of the interface for a 16-bit DSP microprocessor to a high speed memory. In the VHDL simulation shown in Figure F-5, two data transfer cycles are shown. The first data transfer cycle is a write to m6164 memory at address 0x0000, while the second data transfer cycle is a read from the m2764 at address 0xe000.

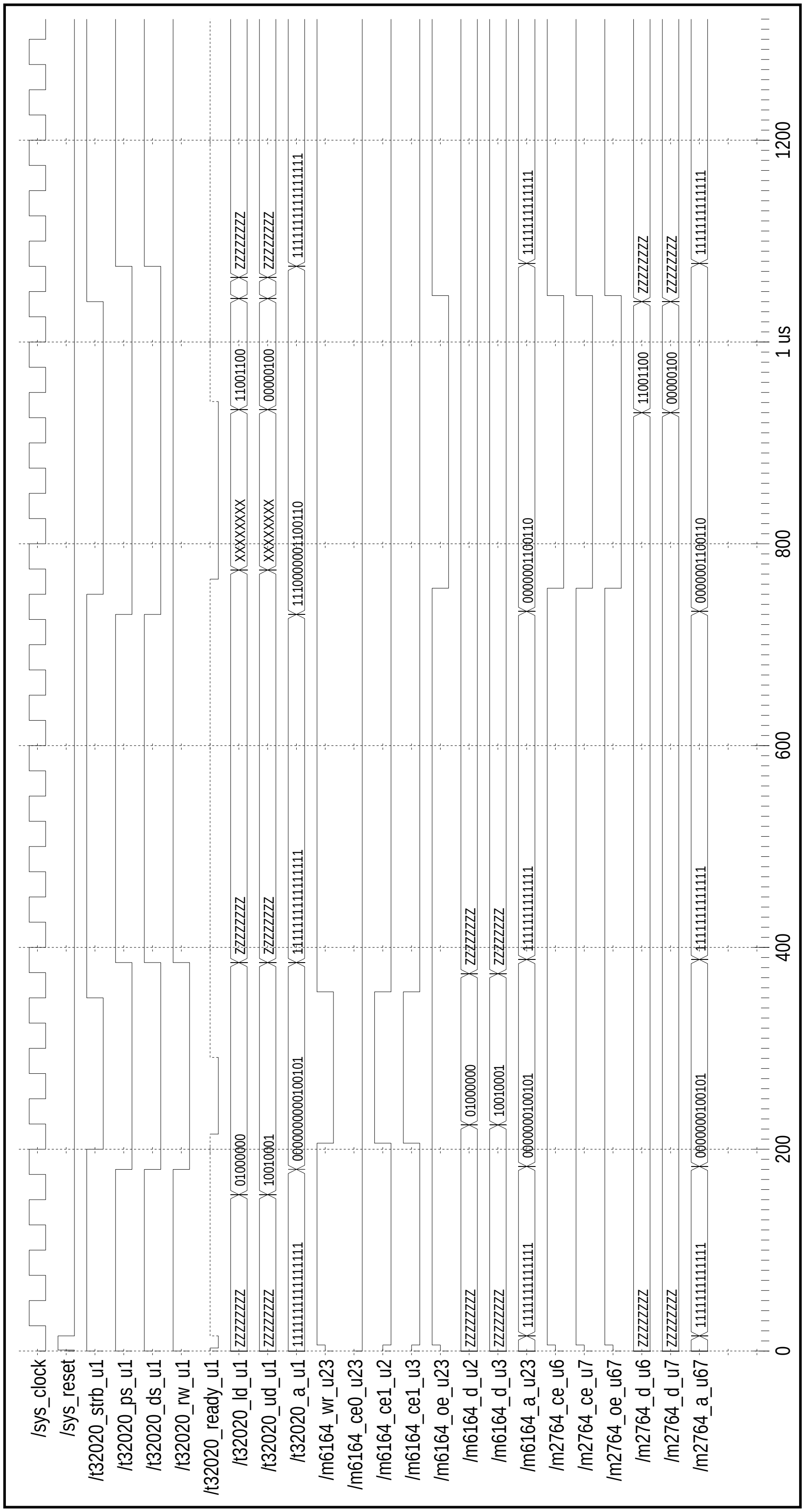


FIGURE F-5. t32020 Design - VHDL Simulation

Appendix G
The Model Frame

When entering the component data structures into the component library, a set of device frames is created that will represent the component. For example, when entering the 68000 microprocessor, a device frame will be created that is based on the microprocessor frame prototype. The created frames will be linked to its prototype through a ^*is-a* relation. The frame will have a series of slots that must be filled in. The content of a slot can be static value such as the number 68 for the number of pins, or it can be another frame. In any case, for every slot there will be a limited number of possible types of entries: either a value such as a number with a certain range, or some other frame. Under some circumstance a slot may be left empty. What is allowed for the content of a slot is determined by the rules used to design with the component. For example consider the ^*has-capability* slot for a component: only those capabilities for which the expert system rule base was written are allowed for the content of the ^*has-capability* slot. For now only the data transfer interface and the bus arbitration interface are considered by the rule base, so only bus arbitration and data transfer capability frames are allowed in the ^*has-capability* slot.

A method is required to represent the permitted or required content of a slot in a device frame. This is required for several reasons: First it allows documentation of the possible devices for which the interface rules developed so far will work. Second it allows a small expert system to be written, that uses the data structure to help and guide a knowledge engineer in entering a new device, assuring that no mistakes are made. Third it allows an expert system to be written which can test an entered device for validity: The expert system can verify if the content of a slot is valid and if all the required information to complete the design is present.

The data structure that represents the permitted or required content of a slot is called a *model frame*. Every prototype frame will have a model frame that indicates the permitted or allowed values for the slots of a device frame based on the prototype frame. The model frame will guide the design engineer that is entering the component, giving him hints on what possible object or value to put as the content of a slot. This ’guiding’ is either manual (i.e. the design engineer looks at the model frames and figures out what to put in the device frame slot) or automated using an intelligent component editor (i.e. a small expert system that tells the design engineer what the contents of a slot should/must be).

Using the model frame when entering a device, it would be possible to present the user of the component entry expert system with a choice of possible options for the contents of a slot. For example, if the user is currently entering a new XYZ microprocessor component, he will create a frame which will be based on the microprocessor prototype. After creating a device frame based on the microprocessor prototype and calling it XYZ, slots must be created in the XYZ frame and filled in. One of these slots is the ^*has-capability* slot. When filling in the ^*has-capability* slot of the XYZ microprocessor, the user should be presented with a choice of creating a data transfer, interrupt or bus arbitration capability frame, assuming rules have been developed in the expert system to design the data transfer, interrupt or bus arbitration interface. No other frames or values are allowed for the ^*has-capability* slot. If the user decides that the device under construction has data transfer capability, a new data transfer capability device frame is created and entered in the ^*has-capability* slot. This new frame must then be filled in according to information obtained from its prototype frame.

The model frames of a prototype frames will have their own hierarchy as shown in Figure G-1. They give the possible device hierarchy, since every model frame slot will contain any allowed model frame. The hierarchy of the model frame serves two purposes: First the hierarchy of prototype models is used to give a complete device hierarchy since every slot in the model has a permitted model frame or data value. Second, every slot in the model frame can have a comment or note attached which can be shown to the user of the system who is trying to enter a component into the library of components, and thus gives a method of assisting the user in the entry of components.

The flexibility of the model frame hierarchy allows device models of any detail to be entered. The limit is not the information the model can show, but how much information the design engineer who enters the prototypes into the frame library is willing to enter into the model frames.

The complete overall device frame layout including the model frame is shown in Figure G-2 for a 68000 microprocessor. The frames found in the component model can be organized into four different classes: prototype, model, device and instance. The prototype class contains all the information that should be inherited by a device frame. The model class contains all the possible and permitted contents of the slots of device frame that is based on a prototype frame. The device class contains the actual frame that make up a device. The instance class refers to actual devices in a system and will inherit all information from a device class frame.

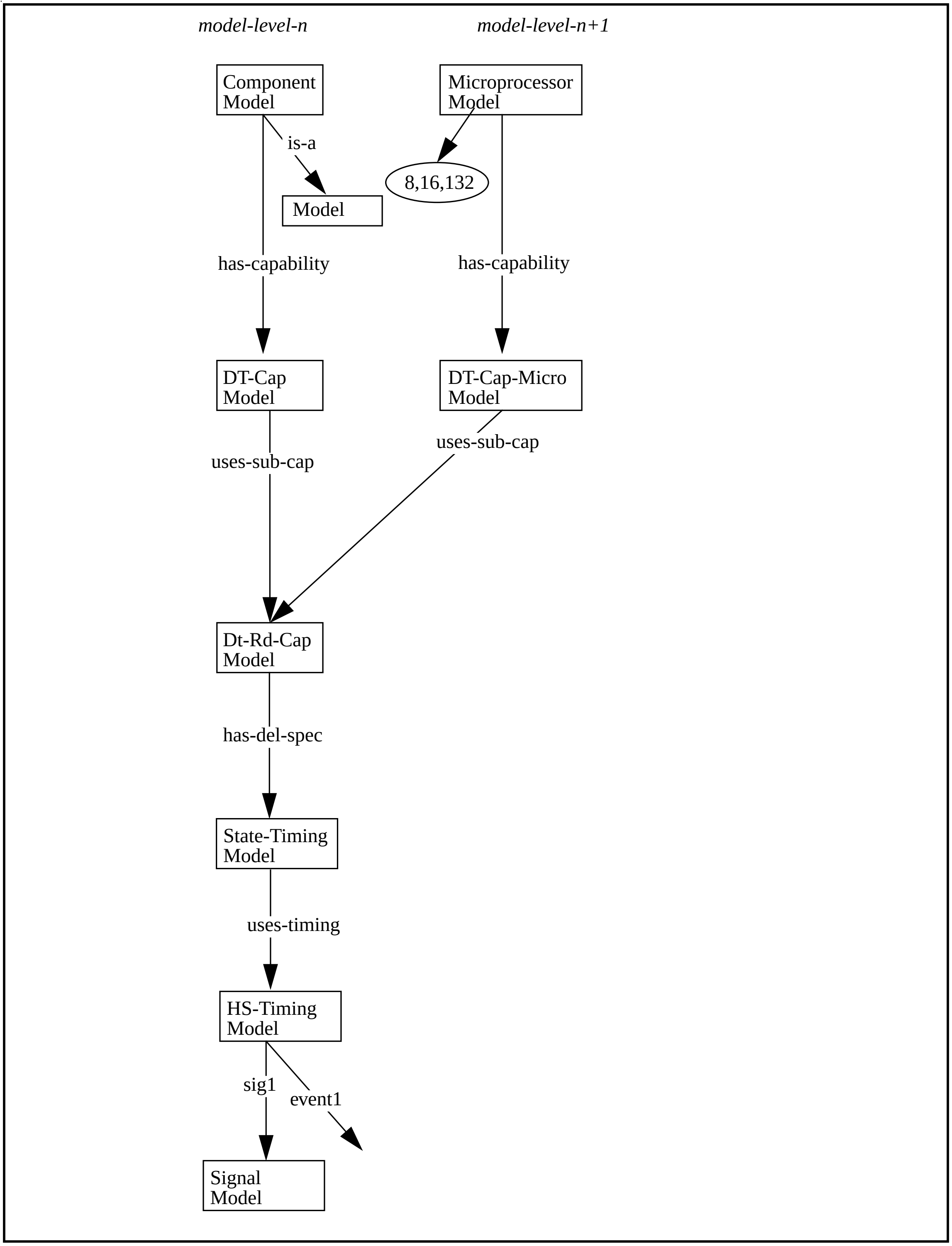


FIGURE G-1. The Model Hierarchy

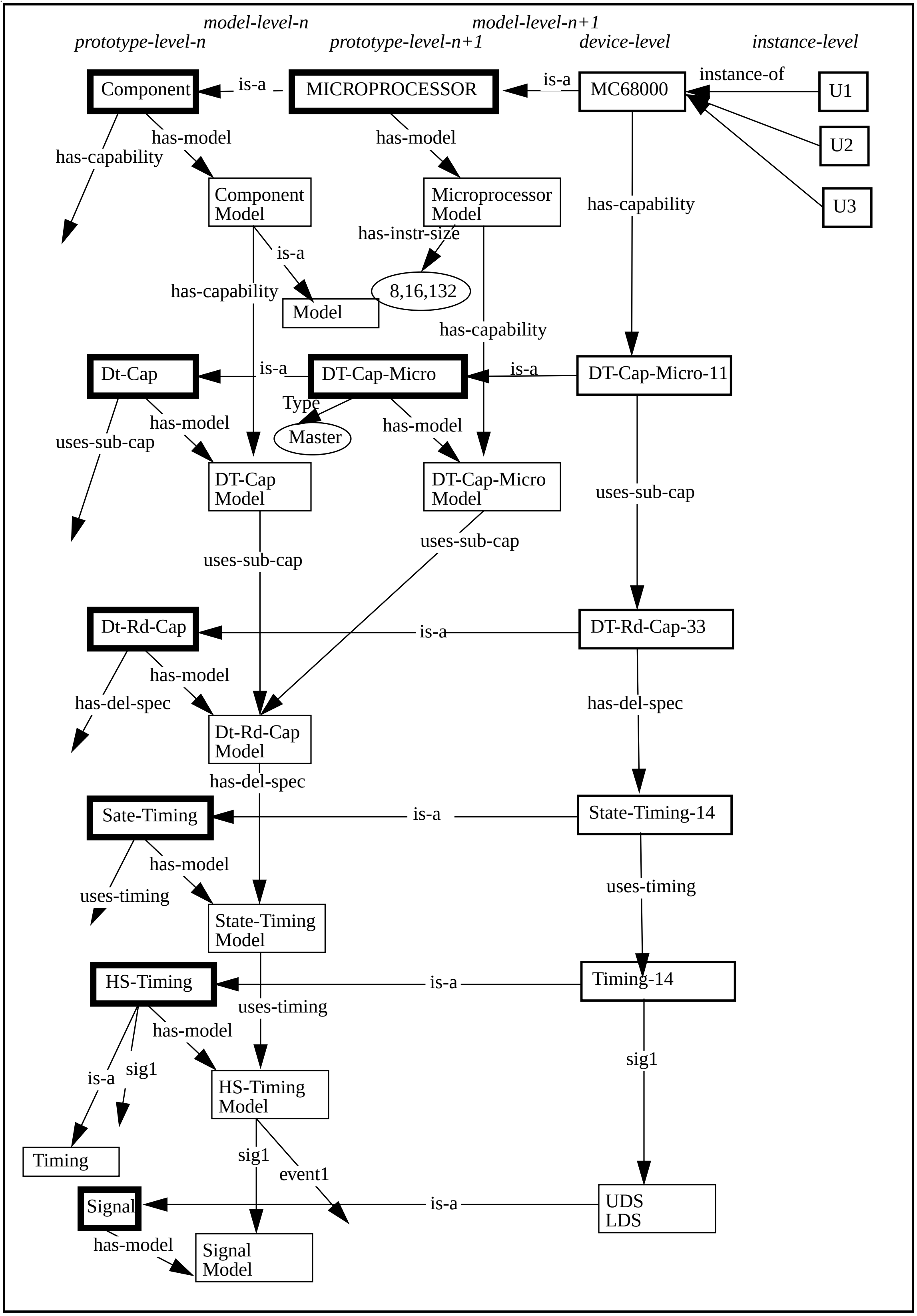


FIGURE G-2. Prototype, Model, Device and Instance of Device frames

PARTIAL COPYRIGHT LICENSE

Title of Dissertation: **Microprocessor System Data Transfer Interface Design: An Expert System Approach Using Signal Timing Behavioral Patterns.**

Author: ______________________________

Benedikt T. Huber

5 November 1998

VITA

Surname: Huber **Given Names**: Benedikt Theodor

Place of Birth: Munich, Germany

Educational Institutions Attended:

University of Victoria	1985 to 1986
University of Victoria	1978 to 1983

Degrees Awarded:

B. Sc.	University of Victoria	1983
M. Sc.	University of Victoria	1986

Honors and Awards:

Graduate teaching award	1991-1995
NSERC post graduate scholarship	1985-1986
NSERC post graduate scholarship	1989-1991
Presidents scholarship	1978,79,81

Publications:

Huber, B. T., K. F. Li, N. J. Dimopoulos, M. Escalante, E. G. Manning,. “Modeling Data Transfer Signals in DAME,” In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, British Columbia, pp. 505-509, May 19-21, 1993.

Huber, B. T., K. F. Li, N. J. Dimopoulos, E. G. Manning,. “Data Transfer Interface Design in DAME,” In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, British Columbia, pp. 510-513, May 19-21, 1993.

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

Dimopoulos, N. J., K. F. Li, E. G. Manning, B. T. Huber, M. Escalante, D. Li, D. Caughey. “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, British Columbia, pp. 388-391, May 9-10, 1991.

Dimopoulos, N. J., B. T. Huber, K. F. Li, D. Caughey, M. Escalante, D. Li, R. Burnett and E. G.Manning. “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.

Huber, B. T., K.F. Li, N.J. Dimopoulos, D. Li, R. Burnett, E.Manning, “Modelling Signal Behavior in DAME,” *Proceedings of the 1990 International Symposium on Circuits and Systems*, New Orleans, La., Vol. 2 pp. 1497-1500, Apr. 29 - May 3, 1990.