

**A New Interface Specification Methodology
and its
Application to Transducer Synthesis**

Gaetano Borriello

**Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley**

Copyright © 1988, Gaetano Borriello.
All rights reserved.

**A New Interface Specification Methodology
and its
Application to Transducer Synthesis**

**by
Gaetano Borriello
B.S. (Polytechnic Institute of New York) 1979
M.S. (Stanford University) 1981**

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

**DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE**

in the

**GRADUATE DIVISION
OF THE
UNIVERSITY OF CALIFORNIA, BERKELEY**

Approved: Randy H. Katz (Chairman)	May 26, 1988
Carlo H. Sequin	May 25, 1988
Alice M. Agogino	May 24, 1988

Filed:	May 26, 1988
--------	--------------

Research supported by the University of California under a Microelectronics Fellowship and the Defense Advanced Research Projects Agency under Contract No. N00039-83-C-0107.

A New Interface Specification Methodology and its Application to Transducer Synthesis

ABSTRACT

In this dissertation, I present a new methodology for the abstract specification of digital circuit interfaces. An interface is the collection of signal wires that cross a circuit boundary and the constraints on the events on those wires. The specification methodology is based on a formalization of the timing diagrams commonly used by digital circuit designers. This mostly graphical method is not only familiar to its intended users but is also concise in its description. An interactive editor, called *Waves*, has been implemented to support this methodology and used to describe a wide range of circuit interfaces ranging from static memories, to microprocessors, to system busses.

Interface specification has a wide range of uses during the design and evaluation of a circuit. *Waves* diagrams and the constraints they capture form the basis for an entire new set of CAD tools that reason about interface design, synthesis, evaluation, and testing. One of these applications, the automatic synthesis of interface transducers, is highlighted in this dissertation.

An interface transducer is the collection of logic circuitry that connects two compatible circuit interfaces. In general, it includes both synchronous and asynchronous components and must satisfy the timing constraints of both interfaces. Interface transducers are required whenever a custom chip is integrated into a computer system or in general, whenever two circuit blocks need to be connected. Their automatic design can greatly reduce the time required to assemble systems or integrate new components into existing systems.

Janus uses a novel approach, based on a small set of templates, to synthesize mixed asynchronous and synchronous control logic. The synthesis algorithm, called *Suture*, first constructs a skeletal circuit and then locally modifies the design to meet interface timing constraints and eliminate internal race conditions. Optimizations of the resulting sequential logic yield transducers that are comparable in both size and performance to those generated by experienced designers. Three practical examples are used to demonstrate this result.

A mio padre

ACKNOWLEDGEMENTS

First and foremost I must thank my advisor, Randy Katz, for taking a chance on me when I first came to Berkeley and helping me get over the rough spots of the last four years. Many thanks also go to the remainder of my dissertation committee Alice Agogino, Alberto Sangiovanni-Vincentelli, Carlo Sequin, and Mark Stefik for their superb guidance and advice.

A special acknowledgement to my friends from my days at the Xerox Palo Alto Research Center: Lynn Conway, Mary Hausladen, Richard Lyon, Alan Paeth, and Mark Stefik. They continuously expanded my horizons and made PARC an incredibly enlightening and exciting place.

The work presented in this dissertation benefitted greatly from many discussions with my friends and colleagues. Among these I must especially thank David Wood, Richard Rudell, David Gedye, and Fred Obermeier for letting me sound them out on many half-baked ideas. To me, my writing still leaves much to be desired. Without the help of David Gedye, Randy Katz, Robert Mayo, and Melissa Westbrook, it may never have become presentable.

My time at Berkeley was made enjoyable and memorable by great friends. They are too numerous for me to include them all, but I must mention Margaret Butler, Gregg Foster, Susan Eggers, David Gedye, Garth Gibson, Mark Hill, James Larus, Robert Mayo, Fred Obermeier, Stuart Sechrest, and David Wood.

My parents, Rosa and Cristoforo, are the best anyone can ever hope to have. They have constantly supported and encouraged me in all my efforts, helping me through my life with their inexhaustable love and affection. I only hope that I may be half as selfless with my own children.

No thanks are enough for my wife, Melissa. Her companionship, love, and support made finishing this work easier and more pleasant than I could have hoped.

And lastly, a special thanks to my son, Cristoforo Samuel, who arrived in time to catch the final moments and put this effort in proper perspective.

CONTENTS

1. Introduction	1
1.1. Circuit Interfaces and their Specification	2
1.1.1. Interface Specification	3
1.1.2. Applications of the Specifications	4
1.2. Interface Transducer Synthesis	5
1.2.1. A Brief Example	5
1.2.2. Summary of Results	8
1.3. Organization of this Dissertation	9

— PART I —

2. Interface Specification	13
2.1. Circuit Interfaces as Collections of Constraints	14
2.1.1 Logical Constraints	14
2.1.2 Timing Constraints	15
2.2. Requirements for Interface Specification	17
2.2.1. Specification of Event Sequences	17
2.2.2. Co-routine Model for Combination	18
2.2.3. Expressibility of HDLs	18
2.3. Related Specification Work	19
2.3.1 Hardware Description Languages	19
2.3.2 State Graphs and Petri Nets	25
2.3.3 Temporal Logic	27
2.4. Formalized Timing Diagrams — A New Approach	31
3. Waves	33
3.1. A Formalized Timing Diagram Editor	34
3.2. The Basic Waves Diagram	35
3.2.1. The Signal Name Window	36
3.2.2. The Trace Window	37
3.2.3. The Time Line Window	38
3.2.4. The Feedback Window	39
3.2.5. The Title Window	39
3.2.6. The Waves Icon	40
3.3. Extensions for Interface Specification	41
3.3.1. Representation of Arbitrary Logic Circuitry	41
3.3.2. Conditional and Looping Event Sequences	44
3.3.3. Combination of Diagrams	47

3.4. Applications of Waves.....	49
3.4.1. Interface Documentation.....	49
3.4.2. Interface Design.....	50
3.4.3. Simulation and Testing	51
3.4.4. Synthesis of Interface Circuitry	53

— PART II —

4. Transducer Synthesis	57
4.1. Interface Transducers.....	58
4.1.1. Interface Operations	58
4.1.2. Specification of Transducer Behavior	59
4.1.3. Automatic Synthesis of Transducers	60
4.2. Related Synthesis Work	62
4.2.1. Synthesis from Algorithmic Specifications.....	63
4.2.2. Synthesis from Input/Output Specifications	68
4.3. Suture — A New Synthesis Method	72
4.3.1. Overview of the Method	73
4.3.2. Examples of its Application.....	77
5. Janus.....	87
5.1. A Synthesis Tool for Interface Transducers.....	88
5.1.1. Specification Using Waves Diagrams	89
5.1.2. Limitations of the Implementation	90
5.2. Generation of the Event Graphs.....	92
5.2.1. Merge Points	93
5.2.2. Intervals of Occurance	93
5.2.3. Interconnection of Operation Graphs.....	95
5.2.4. Timing Constraint Translations	97
5.2.5. Compression of Synchronous Events	98
5.2.6. Splitting of Signals	100
5.2.7. Extraneous Events.....	101
5.3. Synthesis of the Data Path Circuitry	102
5.3.1. Data Transfers Through the Transducer	102
5.3.2. Multiplexed Data Transfer Paths	102
5.4. Synthesis of the Control Circuitry	104
5.4.1. Synthesis of the Skeletal Circuit.....	104
5.4.2. Local Corrections for Constraint Satisfaction	105
5.4.3. Local Corrections for Race Elimination	106
5.5. Extensions for Conditionals and Loops	109
5.6. Logic Optimization	110
5.6.1. Merging Across Operations	110
5.6.2. Sequential Logic Optimizations	111

— CONCLUSION —

6. Conclusions and Contributions	115
6.1. Summary of Contributions	116
6.1.1. Interface Specification	116
6.1.2. Transducer Synthesis	117
6.2. Directions for Future Research	119
6.2.1. Interface Specification	119
6.2.2. Application of Interface Specifications	120
6.2.3. Transducer Synthesis	120
6.2.4. Summary	122
6.3. Closing Comments	123

— APPENDICES —

A. Waves Implementation	127
A.1. Implementation Medium	128
A.2. Decal Windows	130
A.3. Specification Dialog Windows	132
A.4. Constraint Checking	135
A.5. Diagram Editing Support	137
A.6. Diagram Regions	139
A.7. Waves Object Classes	141
A.8. Portability Issues	143
B. Waves Specification Examples	145
B.1. The Intel Multibus	146
B.1.1. Slave Read	146
B.1.2. Slave Write	147
B.1.3. Arbitration	148
B.1.4. Master Read	149
B.1.5. Master Write	150
B.2. The Multibus Design Frame	151
B.2.1. Slave Read	151
B.2.2. Slave Write	152
B.2.3. Master Read	152
B.2.4. Master Write	153
B.3. 2-Phase and 4-Phase Protocols	154
B.3.1. 2-Phase Protocol	154
B.3.2. 4-Phase Protocol	155

B.4. SPUR PCC-SBC Interface.....	156
B.4.1. Clock Signals.....	157
B.4.2. PCC to SBC Request.....	157
B.4.3. SBC to PCC Request.....	159
B.4.4. Cache Handshaking.....	160
B.5. The Texas Instruments NuBus.....	161
B.5.1. Arbitration.....	161
B.5.2. Master Read.....	162
B.5.3. Master Write.....	164
B.6. The Motorola 68000.....	165
B.6.1. Read.....	165
B.6.2. Write.....	166
B.7. Static RAM.....	167
B.7.1. Read.....	167
B.7.2. Write.....	168
C. Janus Implementation.....	169
C.1. Janus Object Classes.....	170
C.2. Validation of Input Specifications.....	172
C.3. Event Graph Browser.....	173
C.4. Representation of the Circuit Library.....	175
C.5. Simulation and Validation.....	177
C.6. Portability Issues.....	178
D. Janus Synthesis Examples.....	179
D.1. The Multibus Design Frame.....	180
D.1.1. Event Graphs.....	180
D.1.2. Janus and Designer Generated Circuits.....	184
D.1.3. Summary and Comparison.....	190
D.2. 2-Phase to 4-Phase Protocol Adapter.....	192
D.2.1. Event Graph.....	193
D.2.2. Janus and Designer Generated Circuits.....	194
D.2.3. Summary and Comparison.....	195
D.3. SPUR PCC-SBC Interface.....	196
D.3.1. Event Graphs.....	196
D.3.2. Janus and Designer Generated Circuits.....	198
D.3.3. Summary and Comparison.....	200
References.....	201
Part I.....	202
Part II.....	204
Appendices.....	206

FIGURES

1.1.	An Example Waves Specification	6
1.2.	An Example Synthesized Transducer	7
2.1.	Excerpts from the Intel Multibus Specification	21
2.2.	SLIDE Multibus Master Read Specification	22
2.3.	BSI/ISPS Multibus Master Read Specification	23
2.4.	ADAM Multibus Master Read Specification	25
2.5.	I-net Multibus Master Read Specification	26
2.6.	CCS Multibus Master Read Specification	27
2.7.	Temporal Logic Multibus Master Read Specification	28
2.8.	Regural Expression Multibus Master Read Specification	29
2.9.	Formalized Timing Diagram Specification	32
3.1.	A Sample Waves Diagram	35
3.2.	A Signal Name Window	36
3.3.	A Trace Window	37
3.4.	A Time Line Window	38
3.5.	A Feedback Window	39
3.6.	A Title Window	39
3.7.	The Waves Icon and Diagram Menu	40
3.8.	A Diagram with a Computed Signal	43
3.9.	The Waves Boolean Expression Menu	44
3.10.	A Diagram with Looping and Conditional Segments	45
3.11.	A Time Line Window with Nested Segments	46
3.12.	An Example of Diagram Combination	48
3.13.	Compatibility of Synchronous Waveforms	51
4.1.	An Interface Transducer	58
4.2.	A Y-Chart for Transducer Synthesis	62
4.3.	BSI/ISPS Specification for an Example Transducer	65
4.4.	Scheduled VT-bodies for the Specification of Figure 4.3	66
4.5.	Synthesized Circuit for the Example of Figure 4.3	67
4.6.	STG Specification for an Example Control Circuit	70
4.7.	Synthesized Circuit for the Example of Figure 4.6	70
4.8.	Circuit Templates Used by the Suture Method	74
4.9.	Examples of Sequential Logic Transformations	76
4.10.	Waves Specification for a Three-Bit Counter	77
4.11.	Event Graph for the Counter of Figure 4.9	78
4.12.	Circuit Synthesized by Suture from Graph of Figure 4.9	79
4.13.	Waves Specification of the Example of Figure 4.6	80

4.14.	Event Graph for the Example of Figure 4.13	80
4.15.	Circuit Synthesized by Suture from Graph of Figure 4.14	81
4.16.	Waves Specification of the Transducer of Figure 4.3	82
4.17.	Event Graph for the Example of Figure 4.16	83
4.18.	Circuit Synthesized by Suture from Graph of Figure 4.17	84
5.1.	A Y-chart for Janus	88
5.2.	The Janus Icon and Menu	89
5.3.	A Janus Interface Transducer Icon	90
5.4.	Example of Intervals of Occurrence	94
5.5.	Interconnection of Event Graphs	96
5.6.	Example of Timing Constraint Translation	98
5.7.	A Specification of Compressible Synchronous Events	99
5.8.	Template Changes for Compressible Events	99
5.9.	Splitting of a Tri-statable Signal	100
5.10.	A Multiplexed Data Transfer Path	103
5.11.	The Implementation of an Output Signal Latch	104
5.12.	Addition of Timing Constraint to Improve Checking	106
A.1.	The Decal Window Object Class Inheritance Lattice	133
A.2.	A Specification Dialog Window	133
A.3.	Examples of Constraint Violations	136
A.4.	Waves Diagram Scrolling	138
A.5.	Waves Diagram Regions	139
A.6.	Waves Object Class Inheritance Lattice	141
B.1.	The Multibus Slave Read Operation	146
B.2.	The Multibus Slave Write Operation	147
B.3.	The Multibus Arbitration Sequence	148
B.4.	The Multibus Master Read Operation	149
B.5.	The Multibus Master Write Operation	150
B.6.	The Multibus Design Frame Slave Read Operation	151
B.7.	The Multibus Design Frame Slave Write Operation	152
B.8.	The Multibus Design Frame Master Read Operation	152
B.9.	The Multibus Design Frame Master Write Operation	153
B.10.	The 2-Phase Protocol	154
B.11.	The 4-Phase Protocol	155
B.12.	The SPUR PCC and SBC Clock Signals	157
B.13.	The SPUR PCC-to-SBC Request Operation	158
B.14.	The SPUR SBC-to-PCC Request Operation	159
B.15.	The SPUR PCC-SBC Cache Handshaking Operation	160
B.16.	The NuBus Arbitration Sequence	162
B.17.	The NuBus Master Read Operation	163

B.18.	The NuBus Master Write Operation	164
B.19.	The Motorola 68000 Read Operation	165
B.20.	The Motorola 68000 Write Operation.....	166
B.21.	The Lattice Logic SR64K4-35 Read Operation	167
B.22.	The Lattice Logic SR64K4-35 Write Operation	168
C.1.	Janus Object Class Inheritance Lattice.....	170
C.2.	Janus Event Graph Browser	174
D.1.	Event Graph for the Slave Read Operation of MDF	180
D.2.	Event Graph for the Slave Write Operation of MDF	181
D.3.	Event Graph for the Master Read Operation of MDF	182
D.4.	Event Graph for the Master Write Operation of MDF	183
D.5.	Manually Designed Circuit for MDF (Part I)	184
D.6.	Manually Designed Circuit for MDF (Part II).....	185
D.7.	Janus Designed Circuit for MDF (Part I)	186
D.8.	Janus Designed Circuit for MDF (Part II).....	187
D.9.	Janus Designed Circuit for MDF (Part III)	188
D.10.	Janus Designed Circuit for MDF (Part IV)	189
D.11.	Waves Diagrams for Double Handshake Operation	192
D.12.	Event Graph for the Double Handshake Operation.....	193
D.13.	Manually Designed Circuit for Protocol Adapter	194
D.14.	Janus Designed Circuit for Protocol Adapter.....	194
D.15.	Event Graph for the PCC-to-SBC Request Operation	196
D.16.	Event Graph for the SBC-to-PCC Request Operation	197
D.17.	Event Graph for the Cache Handshaking Operation.....	197
D.18.	Manually Designed Circuit for SPUR PCC-SBC Interface	198
D.19.	Janus Designed Circuit for SPUR PCC-SBC Interface.....	199

TABLES

1.1.	Comparison of Janus and Experienced Designers	8
2.1.	Classification of Timing Constraints.....	16
5.1.	Time Interval Update for Neighboring Nodes	95
D.1.	Comparison of Circuits for Multibus Design Frame	190
D.2.	Comparison of Circuits for Protocol Adapter	195
D.3.	Comparison of Circuits for SPUR PCC-SBC Interface	200

Introduction

1

Circuit interfaces are an important design abstraction. However, although they are critical to both circuit function and performance, there has been little work by the computer-aided design (CAD) community that directly addresses issues of circuit interface specification and design. An appropriate specification methodology can elevate circuit interfaces to first-class design objects alongside circuit logic and serve as the foundation for an entirely new class of CAD tools for interface design. In this dissertation, I develop an interface specification methodology and demonstrate its utility by automating the logic design of interface transducers, the glue logic that connect two interfaces together. The automatic generation of these circuits required the development of new synthesis algorithms that handle both synchronous and asynchronous circuits and timing constraints on their operation.

This chapter is divided into three sections. In the first section, I will introduce the special nature of circuit interfaces and discuss why a specification method that emphasizes the properties of circuit interfaces is needed. The section makes a case for a specification method that emphasizes the properties of circuit interfaces and describes the use of the specifications in interface CAD tools. The second section outlines one of these applications, the automatic synthesis of interface transducers. It highlights the novel features of the synthesis algorithms developed for this class of circuits and uses a small transducer example to illustrate the form of the initial specification and the resulting logic circuit. A table of results on larger examples shows that the synthesized circuits are comparable in both size and performance to those generated by experienced designers. The third section concludes the chapter with some notes on the organization of the remainder of this dissertation.

A circuit logic block is typically defined by its internal hardware, the logic gates and storage elements that determine the circuit's function. However, this *internal* view is only part of a complete definition. The other, complementary part, is how the circuit is viewed by surrounding circuitry through its *interface*, the signal wires that cross the boundary of the logic block. This *external* view of the circuit is a collection of *constraints* on these signal wires.

No formal or generally accepted definition of a circuit interface exists. In the context of this dissertation, a circuit interface is simply a collection of sets of constraints on the signal wires that cross the periphery of a logic block. Each set corresponds to an *interface operation*, a semantic entity that consists of a sequence of events, the changes in logic level, that represents the exchange of information (i.e., communication) with the circuit's environment. The interface is not concerned with the semantic meaning of the operations but only with the events that must occur and the data that must be transferred. It is also independent of the *interface circuitry*, the internal logic that may use the signals as inputs or generate them as outputs.

The design of circuit interfaces and interface circuitry is as important as the design of internal circuit logic. A circuit interface can be viewed as a contract between a circuit and its environment. If the contract is not met, then the circuit will not be able to communicate with its surroundings and will be useless. If the contract is not met well, then communication might not proceed fast enough to be practical.

The *interface designer* must try to define an *interface contract*, embodied by the interface constraints, that can be met easily and efficiently, with a minimum amount of overhead on the size and complexity of the internal circuitry. Since for a given interface there can be many possible internal implementations that meet the constraints, the *interface circuitry designer* must try to find an implementation of the interface circuitry that is efficient in both the technology used to realize the logic and in the way it complies with the interface contract. Communication across the interface may proceed faster or slower depending on how the constraints are met.

Therefore, interface specification is the description of the constraints on the interface signals. These constraints take many forms that range from the behavioral level — focusing on the sequencing of events — to the details of electrical requirements and proper physical packaging and connections. Traditional hardware description languages are inadequate for interface specification. They emphasize the description of logic circuits and their physical realization in hardware and usually do not provide mechanisms for the specification of abstract constraints.

Circuit interfaces actually have more general applicability than circuit logic. By definition every logic circuit has an interface, however, a circuit interface can exist independently of any logic blocks. Bus structures are a common example of a system component with an interface

but no associated circuitry. A bus specification is solely a collection of constraints and there may not be any associated logic circuitry.

Interface Specification

1.1.1

Interface specifications provide information that is complementary to circuit specification. The interface description of a logic block can be used to generate stimulus vectors for simulators and testers, to verify that two blocks to be connected are compatible, to automatically generate interface circuitry, and to document a design.

Although interface specification is clearly important and many useful tools could be developed to help designers deal with interface issues, there has been little work in the area of interface specification. Most attempts have fallen into one of three categories: extensions to existing hardware description languages (HDLs), the adaptation of Petri nets and other state graphs, and temporal logic specifications.

Each category has its own disadvantages. Because HDLs concentrate on circuit function many interface constraints can become embedded in the logic description. If interface details are not explicitly factored out, it is difficult, if not impossible, to exploit the interface specification for design. Specialized languages have been developed for asynchronous interfaces, an aspect missing from many HDLs, but these languages are usually awkward for synchronous interfaces. Petri nets have also been used to specify asynchronous behavior and sequencing of events but usually ignoring timing constraints. Temporal logic offers a more formal methodology that unfortunately becomes extremely cumbersome when timing constraints are introduced. A problem common to all the categories is the unfamiliarity of designers with the specification language and its idiosyncracies. These approaches are described in more detail in section 2.3.

What is needed is a specification method that covers both synchronous and asynchronous circuits, can handle timing information, is natural and concise for designers to use and, most importantly, stresses the specification of interface constraints over the specification of logic circuitry. In this dissertation, I describe such a method based on timing diagrams.

Timing diagrams are a familiar form of specification for logic designers. The major components of timing diagrams — the shapes of the waveforms and the timing constraints between the changes in logic level — properly emphasize the most important aspects of circuit interfaces. If the diagrams can be formalized to include all the information necessary for interface specifications, then they can provide just the specification method required.

The *Waves* timing diagram editor provides this formalization. It is an interactive editor that directly supports the many types of timing constraints encountered in interface specifications. The editor informs the user of any constraint violations in the drawn waveforms and has application in interface design and documentation as well as specification.

Circuit interface specifications form a foundation upon which tools that reason about interface issues can be built. The classes of tools range from documentation and exploratory design aids to automatic interface design and synthesis tools and are useful through all stages of the design process from initial evaluation through integration and testing.

Waves diagrams are an excellent medium for communicating with interface CAD tools. The diagrams are not only familiar to designers, they also provide dialogue and error reporting capabilities between the user and the tools. Signal events and constraints can be highlighted on the diagrams rather than directing the designer to the object of interest through textual cues in an HDL. The two dimensional nature of the diagrams is also a better match for expressing the constraints between signal waveforms than a linear HDL program.

In the early stages of design, *Waves* diagrams are a spreadsheet-like tool for experimenting with prototypes of the interface specification. An initial sketch of the desired waveforms can be drawn and timing constraints attached to events. Then the designer can vary the positions of some events and see if any timing constraints are violated. The period (or duty cycle) of a clock can be changed to view the effect of different system timing schemes. This is especially helpful when there are interactions with asynchronous signals to consider.

If an interface specification is associated with all system components, then whenever two blocks are to be connected their interfaces can be checked for compatibility. This is an important capability when members of a large design team are simultaneously working on different parts of the design. Slight inconsistencies in circuit connections are a common cause of design bugs.

This leads to a set of tools that can handle incompatibilities and modify or add circuitry to correct the problem. One such tool is an interface transducer synthesizer. An *interface transducer* is the *glue logic* that connects two circuit blocks. Glue logic is common in most systems and especially in those with many components at high levels of integration. Automatic synthesis of interface transducer logic can greatly reduce the design effort in integrating a new custom chip into an existing environment. This enables the *rapid prototyping* of system components, their evaluation in-situ, and their faster introduction into the marketplace.

Finally, tools that deal with simulation and testing issues are made possible. Today, when a chip is tested or a simulation performed the collected output vectors are compared to a sequence of expected vectors. However, this is not the real objective of the test. Rather, we test to determine whether the changes in logic level on the output vectors meet the ordering and timing constraints on the events of the interface, not whether they precisely match one of many possible sequences of acceptable expected outputs. Timing diagrams can be used to directly generate input vectors and then verify that the output vectors do actually meet the constraints.

The application of circuit interface specifications highlighted in this dissertation is the automatic synthesis of interface transducers. Transducers require synthesis methods that differ in two important ways from current methods. First, they need to be able to handle the design of both synchronous and asynchronous components. And second, the synthesized circuits must respect interface timing constraints.

The tool I implemented to perform this task is called *Janus*. Two interface specifications, in the form of collections of *Waves* diagrams, are provided as input to *Janus*. It then generates a specification of the sequential logic that will implement the connection between the two interfaces. Future tools to be integrated with *Janus*, will use *Waves* diagrams to interact with the user during the synthesis process and to compose simulator commands to validate the generated circuitry.

The control logic synthesis algorithm in *Janus* is called *Suture* and it differs from classical synthesis algorithms in several ways. First, in the early states of synthesis, synchronous and asynchronous signals are treated in exactly the same fashion. Second, rather than attempting to generate a correct circuit directly, the *Suture* algorithm constructs a skeletal circuit that may have timing constraint violations and race conditions. Later passes over the circuit correct these problems. Lastly, there is no attempt to generate a dense circuit in the early stages of the algorithm, rather, local transformations are used to reduce the size of the sequential logic once a correct circuit has been generated.

Janus prepares the input to *Suture*. An event graph is derived from the *Waves* diagrams of the interface specifications. *Janus* interconnects the graphs, based on data transfer through the transducer, and calls the *Suture* algorithm as a subroutine. The transducer is designed piecemeal, a complete logic block is separately generated for each interface operation. *Janus* then combines and optimizes the resulting circuitry into a single circuit.

A Brief Example

1.2.1

An example of the use of *Waves* and *Janus* is the problem of connecting a synchronous microprocessor to an asynchronous system bus. Figure 1.1 shows three *Waves* diagrams that are part of the transducer specification. The first diagram describes the read operation as it is seen at the the interface of the microprocessor. The other two diagrams show the details of arbitrating for the system bus — a synchronous process — and the specification for the read operation as seen on the bus, an asynchronous process.

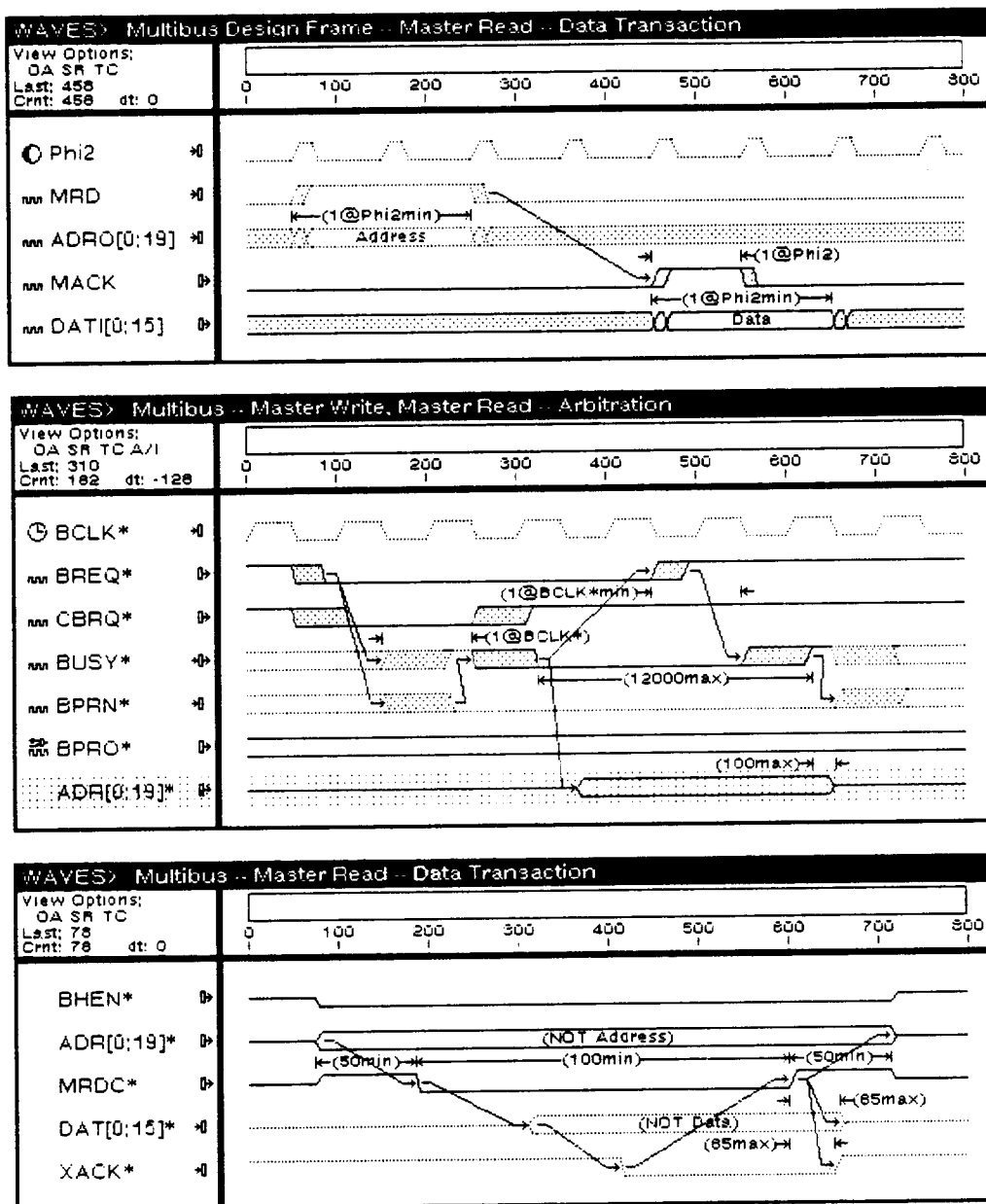


Figure 1.1 Waves diagrams corresponding to the master read operation of the Multibus Design Frame [Borriello85]. The top diagram is the operation as seen by one side of the transducer, a custom synchronous microprocessor. The other two represent the master read operation as seen on the Intel Multibus [Intel82]. One describes arbitration for the bus and the other the details of the data transfer.

These three diagrams are the input to *Janus* and correspond to only one of the many operations performed across this interface. *Janus* requires corresponding diagrams for both interfaces for each operation that the transducer is to support. Not all of the specification is

visible in the diagrams and many constraints are not displayed but entered and modified through the graphical cues on the diagram.

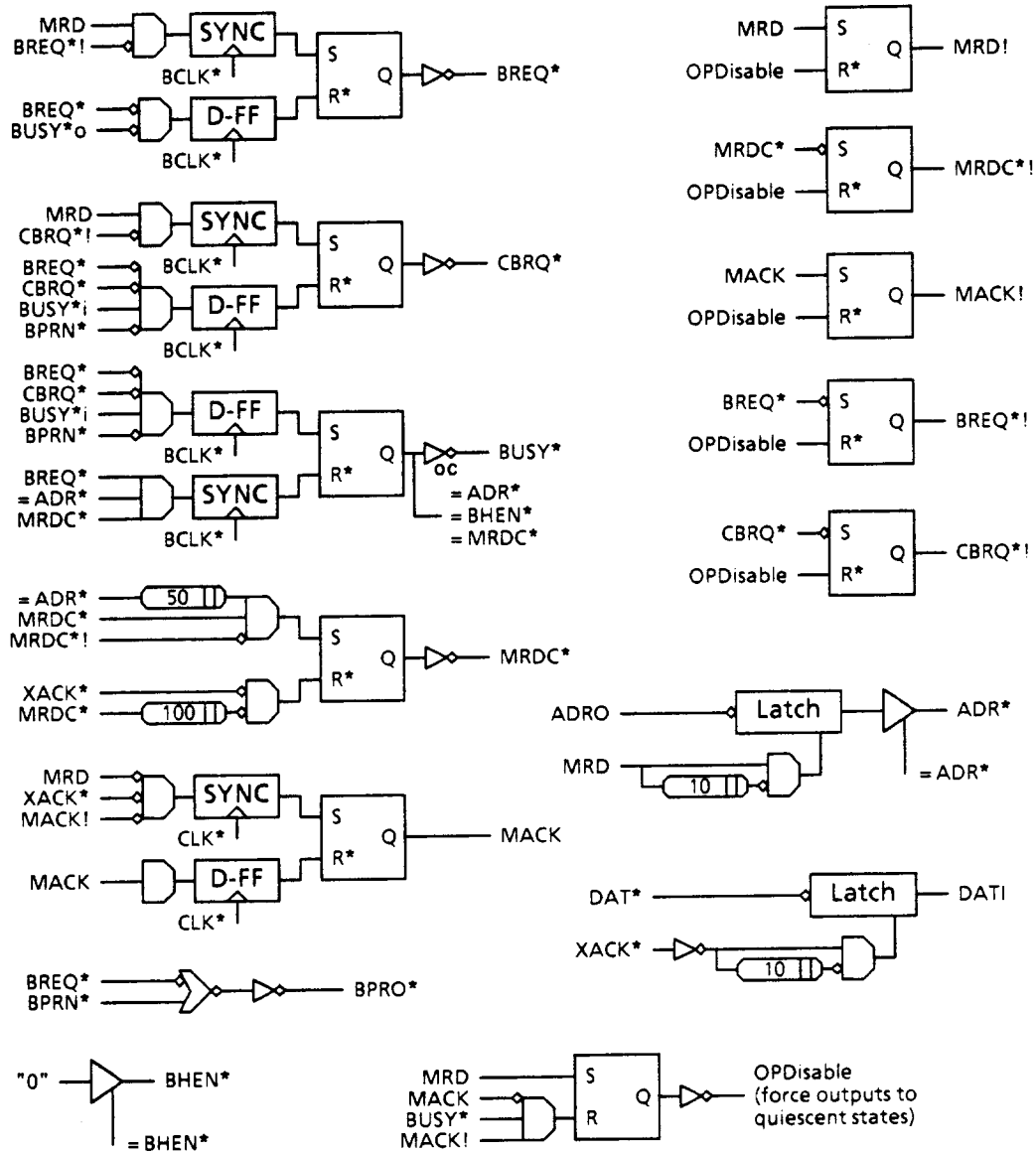


Figure 1.2 Circuitry synthesized by *Janus* from the *Waves* diagram specifications of Figure 1.1. The circuitry is shown before any optimizations. Appendix D provides more details for this and other examples.

The circuitry synthesized by *Janus* is shown in Figure 1.2. As is typical with interface transducers, the circuitry is not particularly large. However, it is complex in its interconnections and in the timing constraints that are enforced by this particular arrangement. The large number of constraints and interactions is what makes interface

transducer design complex and error-prone. *Janus* is meant to relieve the designer of the concerns associated with this myriad of details by using the constraints to synthesize the circuit automatically.

In this example, *Janus* generates circuitry that is only 11% larger than that generated by an experienced human designer. But more importantly, the performance of this interface transducer, obtained by measuring the elapsed time for the transaction, is 9% better than the human-generated design. This can be traced to a simplification made by the designer that greatly reduces the amount of parallelism in the circuit (and the number of interactions). In fact, 40% of the extra circuitry is generated by *Janus* to handle this parallelism.

Summary of Results

1.2.2

Janus emphasizes performance, the most important aspect of a circuit interface. The performance of an interface transducer is determined by its communication bandwidth. A good transducer maximizes the bandwidth by allowing the events to occur as fast as the interface constraints will permit. As can be seen in Table 1.1, the performance of the *Janus*-generated circuitry is at least as good or better as the human-generated circuitry for each of three practical examples. In regard to circuit size, *Janus* is within 20% of the implementations realized by experienced designers. It is important to note that although circuit size is always important, it is less so for interface transducers than other parts of a design. Transducers are not replicated as are many other components, therefore, a slightly larger circuit is usually tolerable. The specifications and resulting circuits for each of the examples of Table 1.1 are described in detail in Appendices B and D.

<u>Example</u>	<u>Performance</u>	<u>Size</u>
Multibus Design Frame	1.09	1.11
2-Phase to 4-Phase Protocol Adapter	1.00	0.61
SPUR PCC-SBC Interface Unit	1.00	0.89

Table 1.1 Comparison of the results of *Janus* and experienced designers for three examples (ratios are *Janus*/Designer). Performance is in terms of maximum throughput (shortest duration for the interface operations). Size is in terms of total number of logic gates. The examples are described in detail in Appendix D.

This dissertation is organized into two principal parts bracketed by this introduction and a summary. Interface specification is the subject of Part I, which consists of the next two chapters. Chapter 2 presents a set of requirements for a circuit interface specification method, describes previous work in this area, outlines a method based on timing diagrams that meets the requirements, and lists the many CAD tools that can be developed based on the substrate of a complete interface specification methodology. Chapter 3 discusses the *Waves* editor and how it formalizes timing diagrams so that a designer can specify circuit interfaces as described in Chapter 2.

Transducer synthesis is the subject of Part II. This part of the dissertation has a similar organization to Part I and is also composed of two chapters. Chapter 4 presents the problems associated with transducer synthesis and explains how previous approaches are inadequate for the task. It then introduces *Suture*, a new synthesis method that can automatically synthesize the control logic of interface transducers. Chapter 5 is a detailed description of *Janus*, the interface transducer synthesizer I have implemented. *Janus* uses the *Suture* method as a subroutine: generating the input to *Suture* from the collection of *Waves* diagrams that specify the transducer and optimizing the sequential and combinational logic output by the synthesis method. The details of the graph algorithms used in *Suture* are also presented in this chapter.

The dissertation concludes with Chapter 6 which describes the contributions of this work and how it builds on previous efforts. The chapter also outlines some avenues for future research into both the application of interface specifications and extensions to the circuit synthesis and optimization techniques developed in *Janus*. A set of appendices provide supplementary information on the implementation of both *Waves* and *Janus*. They include collections of practical examples of interface specification and transducer synthesis that can be used in evaluating this work.

<This page intentionally left blank.>

PART I

<This page intentionally left blank.>

Interface Specification

2

Interface specifications complement circuit specifications. While a circuit specification emphasizes the internal function of a circuit block, an interface specification focuses on the interactions between the circuit and its environment. Circuit logic specifications are collections of Boolean logic equations and memory requirements. Circuit interface specifications are collections of constraints on signal wires in the physical, electrical, logical, and timing domains. Formalized timing diagrams can be used to graphically represent the logical and behavioral constraints of a circuit interface, namely, the timing and sequencing behavior.

This chapter is composed of four sections. In the first section, I describe my taxonomy for the types of constraints that appear in interface specifications. The next section outlines the requirements for a general interface specification methodology. Previous work in this area is surveyed in the third section. The concluding section presents a new approach to interface specification based on *formalized timing diagrams*.

A digital *circuit interface* can be viewed as a group of signal wires with constraints. I classify the constraints into four domains: physical, electrical, logical, and timing. For example, physical constraints include the formfactor of circuit boards, and the positioning and size restrictions of chip packages and signal traces on a circuit board. Logic levels, current requirements, and input/output capacitances are examples of the electrical concerns. In this dissertation, I will concentrate on the logical and timing constraints. The *logical constraints* specify the logic levels along the signal waveforms and the *timing constraints* indicate the proper separation in time of the changes in logic level.

Interface constraints can be collected into sets that correspond to the basic communication operations supported by the interface. An interface operation is an indivisible sequence of events generated by two circuit blocks that constitutes communication between the two circuits. It may be as simple as a two-phase handshake or as complex as a multi-word data-transfer over a shared bus. Different operations may use the same or different signal wires.

Interfaces typically support many operations and each imposes a large number of constraints on the design of the interface circuitry — the circuitry that interacts directly with the signal wires of the interface. These constraints can have many idiosyncratic details. Constraints in one domain are derived from concerns on many different levels of design ranging from physical to behavioral. For example, electrical requirements determine signal rise and fall times that will lead to a timing constraint in the form of a setup time requirement on data to be latched. Considerable design effort is expended in ensuring that all the constraints are satisfied and in achieving good performance across the interface (i.e., that communication proceeds as fast as the constraints permit).

Logical Constraints

2.1.1

Logical constraints determine the shape of a signal waveform by specifying the logic level to be carried on a signal wire over a specified period of time. The boundaries between levels, or *events*, are transitions in logic value or changes in direction, and determine the time interval to which the logical constraints apply. Rise and fall times determine the duration of these transitions. Two orthogonal sets of constraints can be applied to the wire during a time interval: the logic level to be carried (a logic 0, a logic 1, or a high-impedance state) and the direction of the wire (input, output).

Five different level constraints can constrain the logic level carried on a digital signal. These are *logic 0*, *logic 1*, *valid*, *high-impedance (tri-state)*, or *don't care*. A logic 0 or 1 constraint is

self-explanatory; that level must be present on the signal wire during the particular time interval. A valid level means that either a logic 0 or 1 is permitted but the precise value will only be known during each specific use of the interface operation. For example, in the transfer of data it is not known what the logic values of the data will be until the data is actually transferred. A high-impedance or tri-state level means that neither a logic 0 nor logic 1 should be asserted on the signal wire. This constraint is specified when a signal shares a common wire and must defer its use of it to one of the other connected signals. The don't care level signifies that any of the three basic levels (logic 0, logic 1, or tri-state) is acceptable. Although this set of five constraints does not cover all possible combinations of the three basic levels, it does include all the logical constraints used by interface designers.

Logic levels can also be constrained to be a Boolean function of other logic levels. In this case, the logic 0 and logic 1 constraints are never used since the level is determined by the Boolean relationship. However, whether a wire actually carries the level or is in a high-impedance state cannot be expressed by the Boolean function. Valid, tri-state, and don't care constraints still need to be specified for the wire.

Periodic waveforms have a fixed set of level constraints. The signal can consist solely of alternating intervals of logic 0 and logic 1. The position of the events delineating the time intervals is derived from the period and duty-cycle of the waveform.

Specifying that a signal wire is pre-charged or open-collector are examples of electrical, not logical, constraints. An open-collector signal is simply one for which logic 1 and high-impedance logical constraints are equivalent. A pre-charged signal is one that is constrained to be in a high-impedance state during each pre-charge phase. Correct logical constraint specification can be attained with the set of constraints described above. Therefore, whether a signal wire is pre-charged or open-collector is not considered at the logical domain, but rather in the domain of electrical constraints.

Timing Constraints

2.1.2

Logical constraints determine the overall shape of a signal waveform. Timing constraints specify how events are separated in time both for events on the same signal wire and across signal wires. A timing constraint specifies a minimum and maximum time between two events. The events must be separated by no less than the minimum time and no more than the maximum time. In the general case, the minimum and maximum times may be negative as well as positive.

There are more restrictive, but more familiar, forms of timing constraints. These include ordering, simultaneity, and synchronicity constraints. An *ordering constraint* is simply a restriction that one event occurs after another event. The corresponding general timing constraint with a zero minimum time and an infinite maximum time. A *simultaneity constraint* specifies that a set of events occur at the same time, within some tolerance. This corresponds to a set of general timing constraints between each pair of simultaneous events

that have a maximum time equal to the tolerance, and a minimum time that is the negative of the maximum, making it symmetric. A *synchronicity constraint* corresponds to setup and hold times for a synchronous signal. It applies to all events on a signal wire and specifies a timing constraint between the events and the closest edge of the synchronizing signal. In this case, one constraint applies to all events on a signal wire. Table 2.1 is a summary of all these forms of timing constraints and their special characteristics.

<u>Constraint</u>	<u>Minimum</u>	<u>Maximum</u>	<u>Events</u>
general	+/- time	+/- time	one pair
ordering	0	+ infinity	one pair
simultaneity	- infinity	+ tolerance	many
synchronicity	- setup time	+ hold time	many pairs

Table 2.1. The four types of timing constraints and their restrictions. Each type is an abbreviation for a collection of general timing constraints. The ordering constraint is minimum constraint with a value of 0. The simultaneity constraint is a set of maximum constraints between all pairs of events to which it applies. The synchronicity constraint applies to many pairs, constraining each event on a signal to the nearest event on the synchronous periodic signal. This classification is similar to that found in [Granacki86b].

The last type of timing constraint defines the timing relationships of periodic waveforms and is not a variation of the general timing constraint. A periodic waveform is composed of an alternating sequence of logic 0 and logic 1 levels that repeat every *period* with the duration of the logic 1 level defined by the *duty-cycle* of the periodic signal. Therefore, the periodic timing constraint is composed of two values: the period and the duty-cycle. The duty-cycle is often expressed as a fraction of the period.

What distinguishes interface specification languages from functional specification languages is the *emphasis* on constraints. However, there are other aspects of interface behavior that cannot be represented as constraints. To be able to call an interface specification method complete, it must be capable of describing arbitrary circuit behavior. Furthermore, a usable description language requires many of the features common to all computer languages. These include composing a more complex specification from simpler pieces in a hierarchical or non-hierarchical fashion and the ability to specify conditional and looping sequences of events. These are constructs familiar to all programmers, though there are some important differences when these ideas are applied to interfaces. In this section, I will outline the necessary requirements for a complete and general *interface specification language*. The *Waves* editor described in Chapter 3 is an example of a specification methodology that meets these requirements.

Specification of Event Sequences

2.2.1

The ability to express timing constraints distinguishes interface specification languages from traditional hardware description languages. It must be possible to specify a general timing constraint between two events. The prerequisites for this are the ability to specify the shape of the signal waveforms and to uniquely identify events so that they can be connected by one of the constraints outlined above (see Table 2.1).

A specification language must be capable of describing more than just a simple linear sequence of events. It must have the capability of expressing conditional and looping sequences as well. In specifying conditional sequences the differences between interface specification and computer languages is substantial. Each conditional sequence is enabled by a specific event. However, the enabling event may not be just the change in logic levels that defines the event, but also that the event occurred within a specific time interval. This is markedly different from the simple *if-then-else* statement. It includes time as a discriminant as well as values. For example, it should be possible to specify that if an event occurs later than a certain time then a different sequence of events should be followed.

Looping sequences are specified almost exactly as they would be in software computer languages. In fact, it is necessary to also support *while* statements, where an event sequence may be repeated until some terminating event occurs, as well as simple deterministic iterations. The terminating condition for a *while* loop takes the same form as that for conditional sequences. An example of the use of such a loop is in arbitrating for a bus where bus grant lines are repeatedly polled until the bus is granted to the requestor.

Co-routine Model for Combination

2.2.2

The procedure or subroutine is the most commonly used method of composing larger programs from smaller ones. However, this is not a sufficient construct for interface specification. A more appropriate model is that of *communicating sequential processes* or *co-routines*.

Hardware, by its very nature, has a high degree of parallelism. It must be possible to describe communicating processes that proceed through event sequences in parallel. Synchronization points determine when the event sequences must interlock. Events may occur in parallel before or after the synchronization points. This is quite different from sequential procedure calls where a sequence is simply inserted between two events in another sequence.

Expressibility of HDLs

2.2.3

The last requirement for an interface specification language is that it have the expressibility of a general hardware description language. Many interfaces include a description of finite automata that control aspects of the interface or preserve state across interface operations. Typically, this logic is identical in all implementations of the interface circuitry. One example is the bus arbitration or request daisy-chain logic of many busses. Another example is the state information that needs to be preserved across operations in a packet-switched bus to match acknowledge packets with outstanding requests. It should be possible to directly specify these combinational or sequential logic components.

To make this specification possible, three hardware description language features are required in interface specification languages as well: Boolean expressions, latching conditions, and delay elements. Boolean expressions require no explanation. Latching conditions are necessary for interfaces that have state, and together with Boolean expressions, permit the description of arbitrary finite state machines by providing a means of specifying state. Delay elements are necessary for modeling real hardware elements and especially asynchronous components where the delay of components is sometimes critical to the proper function of the circuit. For example, if a signal logic level is a Boolean function of other signals then it should be possible to specify that a change in the logic level of one of the inputs will not be seen at the output signal for a certain period of time. For the purposes of this dissertation, these three features are the minimum required to describe arbitrary digital circuits. Of course, higher-level abstractions based on these primitives, as used in hardware description languages, are extremely useful. For example, an ALU that is part of the interface (e.g., for address computation) should be specified as such and not by the large number of logic equations that would be required.

Interface specification has attracted limited attention from the CAD community. There have been only a handful of attempts at developing interface specification methodologies. These can be classified into three main categories: (1) hardware description languages, (2) state graphs and Petri nets, and (3) temporal logic. The three categories correspond to three specification needs encountered by different communities of researchers. The languages work arose in response to the problems of system specification and hardware synthesis. The state graph approaches derive from work on self-timed circuit design and the specification of communication protocols. Temporal logic specifications were attempted so that formal verification of digital circuits would also include verification of interface constraints.

In this section, I will describe each of the approaches and how they meet some, but not all, of the requirements for an interface specification language presented in the previous section. To make the approaches easier to compare and evaluate, I will specify one of the operations of the Intel Multibus, the master memory read, in each of the methods. This is an asynchronous data transaction and is only one of the many operations supported by the Multibus interface. However, its description raises many of the special issues associated with interface specification. Figure 2.1 contains excerpts from the Multibus specification describing this operation. Timing diagrams and accompanying text are used to outline the sequence of events and describe timing constraints.

Hardware Description Languages

2.3.1

The earliest work in the specification of interface details naturally began in the hardware description language community. This work was motivated by: (1) the complete specification of digital systems including their interfaces, and (2) the synthesis of digital hardware with interface constraints taken into account. A good history of the early work in interface specification languages is given by Parker [Parker85].

2.2.2.5 MULTIBUS COMMANDS. In this section, we will discuss the command lines and how they work in conjunction with the lines explained in the previous sections to accomplish a read or a write operation.

There are four MULTIBUS command lines.

Memory Read Command - MRDC/
I/O Read Command - IORC/
Memory Write Command - MWTC/
I/O Write Command - IOWC/

The command lines, which are driven by three state drivers on the bus master, indicate to the bus slave the action that is being requested.

Read Operations. The two read commands (MRDC/ and IORC/) initiate the same basic type of operation. The only difference being that MRDC/ indicates that a memory address is valid on the MULTIBUS address lines, and IORC/ indicates that there is an I/O port address on the MULTIBUS address lines. This address (memory or I/O port) must be valid on the bus 50ns prior to the read command being generated. When the read command is generated, the slave module (memory or I/O port) puts the data on the MULTIBUS data lines and returns a Transfer Acknowledge (XACK/), indicating that the data has been placed on the bus. When the bus master receives the acknowledge, it strobes in the data and removes the command (MRDC/ or IORC/) from the MULTIBUS interface. The slave address (memory or I/O port) is removed from the bus a minimum of 50ns after the read command is removed. XACK/ must be removed from the MULTIBUS interface within 65nsec after the command is removed, to allow for the next bus cycle. Figure 2-8 shows the timing for the Memory Read or I/O Read command.

3.2.1 READ OPERATIONS (I/O AND MEMORY)

A Read operation transfers data from memory or from I/O to the master that is controlling the bus. For detailed functional descriptions refer to Section 2.2. The lines involved and timing specifications for a Read Operation are as follows (figure 3-3):

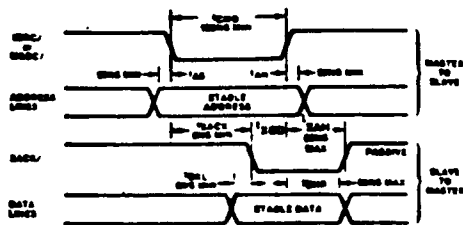


Figure 3-3. Read AC Timing

See Section 5.1 for guidelines and examples. See also the special inhibit operation in Section 3.2.3. For system anomalies with respect to read commands see Section 6.5.

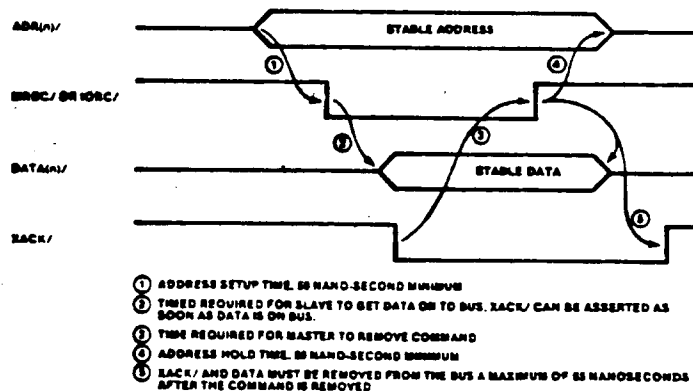


Figure 2-8. Memory or I/O Read Timing

Figure 2.1. Excerpts from the Intel Multibus Specification describing the Multibus master read operation [Intel82]. The excerpts only describe the data transaction part of the operation. The bus arbitration part is not shown for the purpose of brevity. The description includes two timing diagrams and some text to describe the logical and timing constraints. Four signals are used in the operation and a fifth, not shown (BHEN*) to signal a 16-bit wide data transfer. Electrical and physical constraints are described elsewhere in the specification document. Reprinted by permission of Intel Corporation.

Initially, HDL support for interfaces took the form of simple declarative specification of input and output ports to a circuit block, outlining their special electrical and logical characteristics (e.g., output, open collector, active low). This early work was eventually incorporated into the ISPL hardware description language [Barbacci76]. Parallel work at IBM with AHPL, an HDL based on APL, also added some simple timing constructs to represent delay, especially useful for describing asynchronous systems where there is no clock to advance time [Visser76]. However, timing constraints are not represented explicitly, but are embedded in the circuit description. For example, if the circuit is to wait for a transition on an input signal that is to occur within some time interval, a loop construct is used as the specification of this timing constraint. The signal is continuously polled, for a maximum number of iterations derived from the timing constraint, to determine if the transition had occurred. The situation is worse with timing constraints on output signals. The transition is set to occur at a convenient time to meet the constraint but there is no explicit representation of the timing constraint. Therefore, the description includes only one of the many possible correct behaviors for the interface.

The first real example of an interface specification language is SLIDE [Parker81]. SLIDE was originally developed as a programming language that could accurately simulate system interconnect. Since system interconnect (i.e., system busses and backplanes) is primarily a circuit interface, SLIDE directly addresses many of the issues of interface specification.

There are three interesting innovations in SLIDE. The first is the model of interfaces as a collection of communicating sequential processes. Synchronized co-routines are an accurate model of the highly parallel and asynchronous hardware that makes up a computer system and they make it straightforward to implement simulators for system interconnect [Altman80]. The second is the ability to use *signal* and *wait* constructs, similar to those in concurrent programming languages. This allows an explicit representation of timing constraints rather than the polling loop construct discussed above. And lastly, SLIDE emphasizes signal transitions rather than logic levels so that asynchronous systems can be more easily modeled.

Figure 2.2 is a SLIDE description of the Multibus master read operation of Figure 1.1. Here we can see all three of the major features of SLIDE. The interface operation is represented as a separate process that can be started by a *call* from another part of the description or by a *WHEN* clause. A *WHEN* clause specifies a set of conditions that, when met, initiate the

process. The *wait* construct is represented in SLIDE by the *DELAY UNTIL* statement that specifies that the process should remain in its current state until a condition is met. SLIDE also allows for a timeout period on the wait. And lastly, as can be seen in the description, there is equal support for transitions and levels on signal wires.

```

PROCESS masterread;
  BEGIN
    mrdc ← 1; bhen ← 0; adr ← Address;
    bhen.en ← 1; adr.en ← 1; cmd.en ← 1 NEXT
    DELAY 50 NEXT
    mrdc ← \ NEXT
    DELAY UNTIL xack EQL \ NEXT
    Data ← dat NEXT
    mrdc ← / NEXT
    DELAY 50 NEXT
    bhen.en ← 0; adr.en ← 0; cmd.en ← 0 NEXT
  END;

```

Figure 2.2. SLIDE description of the Multibus master read operation. The signals can be assigned levels (1 or 0) and transitions (falling (\) or rising (/)). The Address and Data variables represent bit vectors. Signals with the .en suffix are used as tri-state control for the signal with the same name. The NEXT keyword enforced a sequence on the statements. Statements between two NEXTs can occur in any order.

There are some major difficulties with SLIDE, however. Its emphasis on asynchronous behavior makes it cumbersome to express synchronous behavior. To do so requires that all transitions be described relative to a fine-grain clock, typically with a 1 or 2 nanosecond period. This is necessary for modelling both asynchronous and synchronous behavior in the same program. It is also difficult to express constraints across statements that do not follow each other in the program sequence.

Another approach to interface specification comes from problems in high-level synthesis [Thomas83]. These synthesis systems enable designers to synthesize circuits directly from high-level specifications to actual hardware. However, designers are then faced with the problems of *system integration* (i.e., the satisfaction of interface constraints so two circuit blocks can communicate).

ISPS is a familiar HDL used in high-level synthesis [Barbacci81]. The *Behavioral Synthesis with Interfaces* (BSI) extensions to ISPS add constructs for describing circuit interfaces [Nestor86]. Three features of BSI/ISPS are particularly interesting. The first extends the input and output port semantics to include specifications such as active low and tristatability. The second makes explicit use of ordering statements to enforce the proper order on interface event sequences. Since the synthesis process attempts to optimize circuit size by rescheduling

the order of events in the description, it is necessary to specify explicit ordering constraints to override this rearrangement. This ordering is expressed using the TNEXT() construct as shown in Figure 2.3. Finally, general timing constraints can also be expressed. Rather than using a delay statement as in SLIDE, BSI/ISPS uses labels on statements of the description to identify interesting events. Timing constraints are then declared to exist between these labels and are not restricted to be in the program sequence, as in SLIDE (see Figure 2.2).

```

MasterRead( Address<23:0> )<15:0> : = begin
    OUTPUTP( bhen.l, TS.ENABLE) {L:mr0e};
    OUTPUTP( adr.l, TS.ENABLE) {L:mr1e};
    OUTPUTP( mrdc.l, TS.ENABLE) NEXT
    OUTPUTP( bhen.l, 1 ) {L:mr0};
    OUTPUTP( adr.l, Address) {L:mr1} NEXT
    OUTPUTP( mrdc.l, 1 ) {L:mr2} NEXT TNEXT() NEXT
    WAIT( INPUTP( xack.l )) NEXT
    MasterRead = INPUTP( data.l ) NEXT
    OUTPUTP( mrdc.l, 0 ) {L:mr3} NEXT
    OUTPUTP( adr.l, TS.DISABLE ) {L:mr4} NEXT
    OUTPUTP( bhen.l, TS.DISABLE) {L:mr5} NEXT
    OUTPUTP( mrdc.l, TS.DISABLE) {L:mr6} NEXT
end

mrTenb:      time( mr0e, mr0 ) GEQ 0ns;
mrTena:      time( mr1e, mr1 ) GEQ 0ns;
mrTbs:       time( mr0, mr2 ) GEQ 50ns;
mrTas:       time( mr1, mr2 ) GEQ 50ns;
mrTbh:       time( mr3, mr4 ) GEQ 50ns;
mrTah:       time( mr3, mr5 ) GEQ 50ns;
mrTdis:      time( mr3, mr6 ) GEQ 50ns;
mrTcmd:      time( mr2, mr3 ) GEQ 100ns;

```

Figure 2.3. BSI/ISPS description of the Multibus master read operation (adapted from [Nestor87a]). The procedure takes the address as an argument and returns the data value. Signals with the .l suffix represent lines that are active low, that is, when a logic 1 is assigned to these lines it actually appears a low electrical level. The output statements take two arguments: the name of the signal and the value. The value can be a logic 0, logic 1, or a string to signify that the signal should be driven or high-impedance. The labels used by the timing constraint declarations are within curly brackets ({}).

Two timing constraints of the interface cannot be expressed with BSI/ISPS. These are the 65ns maximum constraints that exist between the event that deasserts the command line (labeled *mr3* in Figure 2.3) and the deassertion of the acknowledge and data lines by the slave

device. To express this constraint it is necessary to use a WAIT statement for the acknowledge input and then add a constraint between that statement and the statement that deasserts the command. However, this is not done in ISPS because the semantics are such that the synthesis procedures would then synthesize circuitry to check that the acknowledge signal was deasserted. This is not the true intent of the specification. It is simply to state that the circuitry can be designed to assume that the signal will be deasserted within 65ns, not that circuitry should be added to actually check for this event. The similar constraint on the data lines cannot even be expressed because the data lines carry unknown logic values and it is not possible to use a WAIT statement because standard digital logic cannot determine when a wire is in a high-impedance state. This is due to the fact that BSI/ISPS was not designed to be an interface specification language. Rather it is intended to be a means of introducing interface constraints into a functional description.

Furthermore, timing constraints are interspersed with functional details. Although the sequence of events on the interface signals and their timing constraints can be separated into their own procedure, as in Figure 2.3, this is not a viable alternative because it leads to inefficiencies in the synthesized circuitry. Unless the procedure is expanded in-line and merged with the procedures that call it, the synthesized interface circuitry will execute sequentially and not in parallel with the circuit internals. The in-line expansion is very difficult because it is not obvious how events should be ordered across procedures. Therefore, timing constraints are typically embedded in the functional description much in the same way as the precursors to SLIDE described above. The separation of interface constraints from circuit function is one of the principal advantages derived from SLIDE's co-routine model that is absent in ISPS.

The *Timing Design System* (TDS) uses a similar approach to constraint specification as BSI/ISPS [Kara86]. The events are simply a list of changes between two logic levels on a signal wire. Constraints are expressed as a minimum and maximum time between two events. Unfortunately, TDS can only express constraints within a fixed sequence of events and lacks the higher-level constructs of HDLs such as conditionals and loops.

A problem common to all these languages is the difficulty of describing event sequences within a linear program. The one-dimensional nature of a programming language is not a good match for the two dimensions of timing constraints. The implicit time axis used to represent the linear sequence of events is the only dimension visually present. The second dimension is used to represent constraints between different signals and is not available in the linear program. This makes the interrelationships between signals difficult to discern. This has led to attempts to make machine-readable interface specification more like the methods employed by designers. One approach has been to describe the interface as well as the circuit in natural language text (see Figure 2.4) [Granacki86a].

Although English is easily readable, it is still difficult to quickly grasp the relationships between the pieces of information described. The use of English only partially alleviates the requirement of learning a new language for specification. It is simply a new and restricted style of a more familiar natural language rather than a familiar computer language.

A 16 bit word of data is read over the dat lines.
A 24 bit address is transferred over the adr lines.
All command lines are active low.
There is an asynchronous four-cycle handshake that takes place across the mrdc and xack lines.
The adr and bhen signals have a 50ns setup time before mrdc is asserted.
The adr and bhen signals have a 50ns hold time after mrdc is deasserted.
The mrdc line must be asserted for a minimum of 100ns.
The data and xack lines will be deasserted within 65ns of mrdc being deasserted.
To read the data the master asserts the mrdc line and then waits for xack to be asserted by the slave.
The data is available while the xack line is asserted.

Figure 2.4. PHRAN-SPAN natural language specification of the Multibus master read operation. To make the natural language system practical the English is restricted to some simple templates of declarative sentences.

My approach is to use the complementary part of interface documentation for formal specification, the timing diagrams. Timing diagrams are ubiquitous in interface specification documents and are familiar to all digital circuit designers. Interestingly, the constraints in TDS are used to derive a timing diagram for the user. In section 2.4, and in more detail in Chapter 3, I will demonstrate how timing diagrams can be used for interface specification, but first, I will describe the other two classes of approaches to interface specification.

State Graphs and Petri Nets

2.3.2

Petri net theory forms the basis of the state graph approaches to interface specification [Petri62]. Petri nets are an abstract graph model for describing event sequencing and have been adapted to describe many sequential processes including interface protocols. Petri nets are bi-partite graphs composed of two types of nodes called *places* and *transitions* [Agerwala79]. Places symbolize computation and transitions are synchronization points for different paths through the graphs. Arcs connect places to transitions and transitions to places. They can fan-out or fan-in arbitrarily.

Petri nets have been adapted for the behavioral description of digital circuits by the self-timed circuits community. These researchers study circuits that function correctly regardless of the relative speed of their components. They are asynchronous circuits designed with only the proper *sequence* of events determining correct operation and not their separation in time. Petri nets are a perfect match for self-timed circuits because they emphasize both event sequencing and parallelism [Misunas73]. One consequence is that the graphs describing these circuits completely ignore timing constraints.

Petri Nets have been used to represent circuit interfaces in the form of I-nets [Molnar85]. In I-nets, transitions are either input or output events. Places correspond to computations that occur between these events. Figure 2.5 is an I-net representation of the Multibus master read operation.

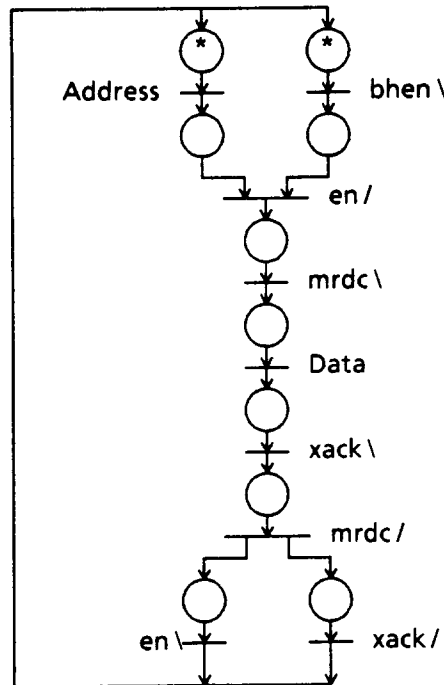


Figure 2.5. I-net Specification of the Multibus master read operation. The place nodes are represented by circles and the transition nodes by horizontal bars. The / and \ notation is used to signify rising or falling transitions on the corresponding signal. The starred (*) nodes represent the initial position of the Petri net tokens.

Timed Petri nets are an extension to the basic Petri net model that assigns a period of time to the nodes in the graph [Goos80]. The two-dimensional nature of the Petri net does make it possible to express delay in more than just the program sequence. These extensions can be applied to I-nets as well. Each place then automatically includes the analogue of a BSI/ISPS minimum timing constraint. But it is not possible to express a maximum timing constraint using this model as was the case in the pre-SLIDE HDL approaches.

Another state-graph approach is based on Milner's calculus of communicating systems (CCS) [Milner80]. The objective of this work is to develop an algebra of operations on circuit interface specifications [Koomen85]. These operations can then be used to derive the specification of a composite block given the two interface specifications for the component blocks. Similarly, the reverse should be possible. Given the desired behavior of a block and one component, it should be possible to derive the interface behavior of the other component.

In this algebra, circuits are only viewed from the outside, through their interface. There is no attempt to model circuit function. Only event sequencing is considered, with timing relationships completely ignored. Interface descriptions are basically state diagrams with arcs corresponding to transitions on interface signals. Figure 2.6 shows a CCS description that is equivalent to that of Figure 2.5.

```

Read  = (adr.out!Address | bhen.out!0) : enable.out!1: Cmd
Cmd   = mrdc.out!0: dat.in?Data: Ack
Ack   = xack.in?0: mrdc.out!1: End
End   = enable.out!0: xack.in?1: Read

```

Figure 2.6. CCS Specification of the Multibus master read operation. State transitions are based on transitions on input signals (.in?) or output signals (.out!). Each transition leads to a new state. In this figure four state are named (Read, Cmd, Ack, and End) and four are not. The "|" symbol is used for parallel combination of sub-graphs, in this case it describes two transitions that can occur in parallel.

Temporal Logic

2.3.3

Temporal logic has also been applied to the high-level specification of digital circuits [Bochmann82]. The thrust of the work is towards the verification of hardware implementations given a high-level behavioral specification written in temporal logic [Dill85]. Temporal logic has proven especially useful in the area of asynchronous and self-timed circuits, where there do not exist as well understood design methodologies as in synchronous design. Asynchronous design is more complex because different sequences of events can occur depending on the relative delays of circuit components. These delays may not be known and vary with time and instance of the circuit. Even just a few components involve a large number of cases that must be considered to insure the circuit operates properly. Continuous signal levels must be taken into account rather than the discrete levels sampled once every clock period as in synchronous design.

A temporal logic specification of a circuit consists of a set of *states* and *statements* that hold for some subset of the states. The logic is built upon logic levels rather than transitions and a state exists for every possible combination of signal logic levels. Directed arcs connect a state to other states that can be reached by a single signal transition. Four different statements, or *formulas*, assert properties of the state graph. If P and Q are Boolean functions of signals then the four operators of temporal logic are as follows: P (no operators), P is true in the current state; $G(P)$, P is true in all following states; $F(P)$, P is true in some future state or states; $P \cup Q$, P is true until Q becomes true. From this set of primitive statements ordered asynchronous sequences of events can be specified (see Figure 2.7).

Temporal logic is a good match for asynchronous, and especially self-timed, logic where only the sequence of events is critical. Since these types of designs are highly error-prone, formal verification methods are essential. Verification is the systematic transformation of one representation of a circuit into another by provably correct steps. The circuit is deemed verified if the two representations can be shown to be identical. Typically the two representations are the circuit logic and the temporal logic specification of its functionality.

```

G( (not mrdc*) implies ( (not mrdc*) U (not xack*) ) )
G( (not xack*) implies ( (not xack*) U mrdc* ) )
G(xack* implies (xack* U (not mrdc*) ) )
G(mrdc* implies (mrdc* U xack*))
G( (and (not mrdc*) (not xack*)) implies F( (and mrdc* xack*) ) )
G( (not mrdc*) implies Address)
G( (and (not mrdc*) (not xack*)) implies Data)

```

Figure 2.7. A temporal logic specification of the Multibus master read operation. Each statement applies to those states where the antecedent of the implication is true. The first four statements express ordering constraints for the four-cycle handshake between *mrdc** and *xack**. The first states that whenever *mrdc** is false it stays false until *xack** becomes false, or in other words, *mrdc** can only rise again after *xack** has gone low. The second states that whenever *xack** is false it stays false until *mrdc** becomes true. The third and fourth statements are similar. The fifth states that if a sequence begins it must eventually end and the last two express the fact that the Address is valid whenever *mrdc** is low and that data is valid whenever *xack** is low.

As was the case with the state graph approaches, timing constraints have traditionally been ignored in temporal logic. To address this deficiency, interval temporal logic (ITL) has been proposed [Moszkowski85]. In ITL, time consists of small indivisible intervals and the formulas of the logic express properties that hold over sets of these intervals. This permits the expression of constraints such as "*A is low in 10 to 20 subintervals*". However, this method has only been applied to small sequential circuits such as flip-flops where interface signals only change level once or twice during the entire operation. Again, the fundamental problem with temporal logic is that it stresses levels and not events. Events cannot be easily identified and therefore it is difficult to differentiate between two similar transitions on the same signal.

Timing could be expressed by drastically increasing the number of states and statements to include all possible combinations of signal levels at every point in time rather than just the possible combinations of logic levels. This has been attempted with the use of regular expressions for describing the sequence of levels on the signals [Kimura87]. A deterministic finite automata, constructed from the regular-expression, is used to verify that a sequence of output vectors from a simulation meets the specification.

A more serious deficiency is that it becomes practically impossible to express constraints that span many events due to the combinatorial explosion in the number of terms of the regular expression. This is the case in the example of Figure 2.8 for the 100ns minimum constraint for the time that *MRDC** is low. Every combination of valid levels would have to be listed for the 20 5ns time periods involved. A similar situation exists for the 65ns max constraints on *DAT** and *XACK**. These constraints are only partially represented in the figure, their full specification would include an order of magnitude more terms (and states in the deterministic finite automata that would verify the sequence).

Each of the approaches to interface specification described in the previous section has three major deficiencies. The first is that each employs a methodology for describing interfaces that is not familiar to the designers that will most use the method. Second, none of them treat synchronous and asynchronous signals uniformly with one stressed to the detriment of the other. Third, in the HDL approaches, timing constraints must be expressed within the confines of a one-dimensional program, making them difficult to express and debug. In the state-graphs and temporal logic, general timing constraints are for the most part ignored or so cumbersome to describe to be practically impossible to express.

A specification methodology is required that addresses these three deficiencies as well as the requirements outlined in section 2.2. I propose the use of formalized timing diagrams as a specification method for circuit interfaces. The nature of timing diagrams directly addresses all the deficiencies of the previous approaches and with some straight-forward extensions can meet all the requirements for an interface specification language.

Timing diagrams are commonly used to describe interfaces both for the purposes of communication among designers and for documentation. They are familiar and natural to designers and have a concise and graphical user interface, unlike the language and temporal logic approaches.

Constraints are collected in timing diagrams with each diagram or group of diagrams correspondings to an interface operation. Events are defined by the shape of the signal waveforms. The logical constraints are drawn directly on the diagrams and determine the precise shape of the waveforms. Timing constraints can be specified between any two events. Constraints common to all the events on a waveform (e.g., synchronicity constraints) can be associated with the signal itself along with other signal properties, such as the direction of the signal (e.g., input, output, or bidirectional).

Timing diagrams present synchronous and asynchronous signals uniformly. Rather than concentrating on one or the other as in the languages described in the previous section, timing diagrams emphasize the position of events on signal wires and the timing constraints that exist between the events. Events can be positioned at any point that satisfies the timing constraints. The way constraints are drawn in the diagrams makes it trivial to determine which of the signals and events they interrelate. The two dimensions of the diagram, the time axis and the signal axis, precisely position each event in time and provide an overall picture of the sequence. This is markedly different from the language approaches where it is not obvious how the constraints relate the signals because the signal axis is completely missing. The time axis is the only one present in the program listing, and it is not even a true time axis that shows the relative duration of the intervals between events.

The contrast between timing diagrams and the other methods is quite obvious when the Multibus example is considered. Figure 2.9 shows the corresponding timing diagram and it should be compared to the descriptions of the previous section and especially Figure 2.1. The timing diagram form is almost identical to the figures from the Multibus specification document. The task of entering the specification should be much easier when one can simply redraw the timing diagrams rather than translating them into a completely different form.

Timing diagrams are in many ways very similar to the event graph approaches described above. Each event is a node in an ordered graph of events corresponding to the interface operation. However, timing diagrams support the expression of general timing constraints between events, unlike the graph approaches. Furthermore, the complexity of the timing diagram correlates well with the complexity of the interface, unlike the temporal logic approaches.

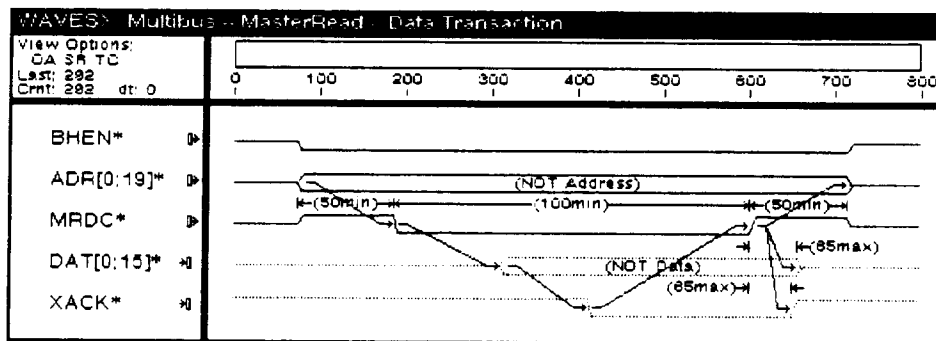


Figure 2.9. Formalized timing diagram specification of the Multibus Master Read operation. This figure should be contrasted with the other specifications in Figures 2.2 through 2.8 and especially with the original specification of Figure 2.1.

In summary, timing diagrams are a much more natural and concise way of describing the interface than a programming language. This is because they focus on the interface's constraints rather than the logic circuitry that operates on its signals, leaving uncommitted the logic that will realize the interface. Their two dimensional nature and uniform treatment of synchronous and asynchronous signals makes it easy to express arbitrary timing constraints. Extensions to support conditional and looping events and diagram composability are described in the next chapter along with the integration of HDL constructs with the timing diagrams so that arbitrary logic can be specified. These extensions make timing diagrams an ideal circuit interface specification methodology that meets all requirements discussed above.

Waves is an interactive editor for the design, specification, and documentation of circuit interfaces. It supports editing operations on signal waveforms collected in timing diagrams. Timing constraints between events on the waveforms can be directly expressed through the editing facilities. The constraints are automatically checked and violations highlighted whenever editing operations cause a change in the position of events. *Waves* diagrams can be used as input to a variety of CAD tools that reason about interface design, synthesis, and testing. *Waves* is implemented in LOOPS, an object-oriented programming extension to the Interlisp-D programming environment, on Xerox 1109 workstations.

This chapter is composed of five sections. The first section is an introduction to formalized timing diagrams as a means of interface specification and the implementation medium chosen for *Waves*. Basic *Waves* diagrams are described in the second section while the third explains how the basic diagram is extended to meet the requirements for an interface specification language outlined in the previous chapter. The chapter concludes with a description of some of the CAD tools that are enabled by the interface specification methodology supported by *Waves*. Appendix A discusses issues in the design of the user interface of *Waves* and the philosophy behind some of the implementation decisions.

Timing diagrams are used by designers to describe the interface behavior of circuit blocks. However, they are an informal method, with many variations in notation and appearance among different communities of designers. This is evident immediately when one leafs through documentation from different semiconductor manufacturers. There have been some attempts to standardize the basic notation for events, causality, and timing constraints [Rony80]. However, these attempts have not addressed the more general problems that need to be solved for general interface specification.

Among these is the problem that timing diagrams do not provide the same power of expression as programming languages. They are good for describing sequences of events but conditional and looping sequences are difficult to express and the constructs used to describe them in manufacturer specification sheets are rarely as general as in programming languages. Also, many interfaces include some logic that is required in every implementation (e.g., the specific logic equations in some bus arbitration schemes). It is usually difficult and cumbersome to describe these Boolean relationships and finite automata in a timing diagram. There have been many ad hoc solutions to these problems, usually involving an attached schematic diagram or textual description, but there has been little proposed standardization of what timing diagrams should look like or extensions to the notation so that arbitrarily complex behavior — that is, any finite automata and its timing constraints — can be described.

In this chapter I describe a set of formalizations and extensions of the timing diagram notation that will make this descriptive method meet the requirements for an interface specification language discussed in the previous chapter. These are embodied in an interactive editor, called *Waves*, that supports interface specification through timing diagrams. What makes *Waves* interesting is how the many of the features of HDLs are incorporated within the framework of the timing diagram. This is accomplished with three formalizations, all of which are necessary for the description of real circuit interfaces. They are the capabilities for expressing arbitrary logic and delay constructs, conditional and looping sequences of events, and the composition of diagrams into larger composite structures.

Waves is implemented in Interlisp-D, a single address space, multi-process, Lisp programming environment running on Xerox 1109 Lisp workstations [Xerox86]. The general procedural interface to window and mouse operations available in Interlisp-D provided the primary impetus for using it to implement *Waves*. *Waves* relies heavily on LOOPS, a set of multi-paradigm programming extensions to Interlisp-D [Bobrow83, Stefik86]. LOOPS adds object-oriented, data-driven, and rule-based programming paradigms to the procedural paradigm already available in Interlisp-D. A more complete discussion of the programming environment and the implementation philosophy behind *Waves* is contained in Appendix A. The appendix also includes a more detailed description of *Waves*' user interface.

A timing diagram in *Waves* is composed of a group of five Interlisp windows (see Figure 3.1). These display a collection of signal wires and waveforms that represent the logic levels on the wires over time. The windows are tied together using the Interlisp *attached window* mechanism. They are moved, resized, and closed as a single entity. The windows have different functions in the editing of diagram information and have the following names: *title* window (the top bar), *feedback* window (on the left below the title window), *time line* window (on the right below the title window), *signal name* window (on the bottom left), and *trace* window (on the bottom right). Through interactions with these windows, the user can use *Waves* to represent basic event sequencing, timing constraints, and signal wire properties.

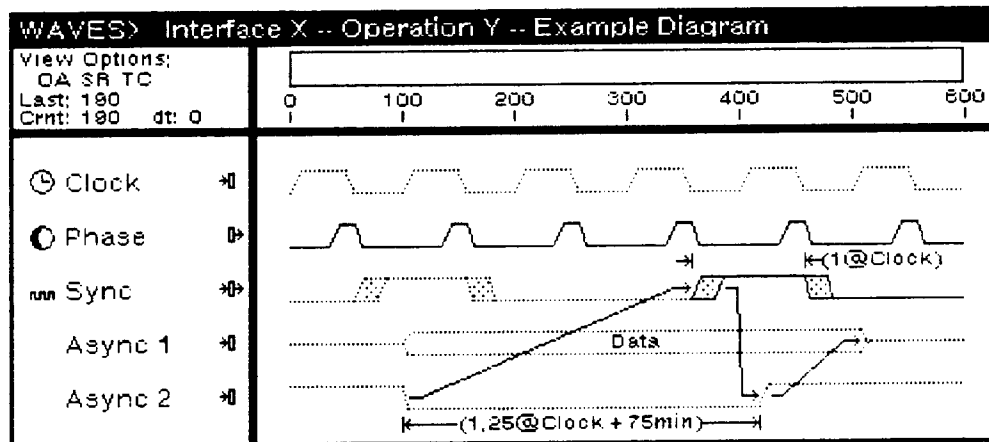


Figure 3.1. An example of a *Waves* timing diagram. It is divided into five windows for: the title bar, feedback information, time line, signal names, and signal traces. The shaded waveforms are inputs and the darker ones are outputs. Events correspond to changes in the logic level of the waveform. Events shown as shaded intervals (don't care) are on synchronous signals and the shading is consistent with the setup and hold time specifications. The signal labelled *Sync* is synchronous to the falling edge of *Phase* and the dilation of the events is determined from its 80ns setup time and 0ns hold time. The logic levels on the signal *Async1* show the representation of the *tri-state* and *valid* logical constraints. Also shown in the diagram are three ordering constraints and two timing constraints. Timing constraints can be expressed as a limited expression consisting of two terms: a number of cycles of a periodic signal and an absolute time amount. A simultaneity constraint is also present between the first events on the two asynchronous signals. Simultaneity constraints are not drawn unless they are violated (see Figure A.3). Not all the specification information is directly visible in the diagram, but all of it is accesible through menus obtained from mouse events on diagram objects.

Waves also includes an extensive repertoire of editing operations. Their implementation and user interfaces are described in Appendix A. In this section, I will describe each of the five windows of the basic diagram in detail.

The Signal Name Window

3.2.1

Besides the name of the signal, the *signal name* window is used to enter all information that applies to an entire signal wire. Examples of this type of information are the directionality of the wire and whether or not the signal is periodic. The direction of a signal is displayed with the small icons to the right of the signal name: input (◁), output (▷), or bidirectional (◁▷). The periodic nature of the signal is displayed on the left side of the name with another set of icons for a clock (⌚), a phase of a clock (⦿), or a synchronous signal (⌘). An asynchronous signal does not have an icon. There is no reference clock to the diagram and there can be any number of signals of each type.

All signal information is attached to the signal names and is entered and modified with the help of a set of pop-menus. The appropriate menu pop-ups depending on the signal or signals selected. For example, for clock signals there is a menu item for changing the period and dutycycle while for non-periodic signals there is one for specifying the setup and hold times if the signal is made synchronous to a clock. Other information, that is not constantly visible, is also attached to the signal names. This includes rise and fall times for edges on the signal and the electrical characteristics of the wire (e.g., open-collector, TTL levels).

⌚ Clock	◁▷
⦿ Phase	▷
⌘ Sync	◁▷
Async 1	◁
Async 2	◁

Figure 3.2. An example of a signal name window. It contains five signal names. The icons on the right are used to indicate the directionality of the signal wire (input, output, or bidirectional). The icons on the left are used to indicate the type of signal (periodic clock, periodic phase of a clock, synchronous, or asynchronous).

As there may be more signals in the diagram than names that can fit, the signal name window is scrollable in the vertical direction. A pop-up scroll bar is visible when the mouse exits the window through its left edge. The user can scroll the window in either direction one signal at a

time or thumb to a specific point. The trace window is scrolled to follow the signal name window.

The Trace Window

3.2.2

The window on the bottom right is the *trace* window. It displays the waveform traces corresponding to each of the signal names in the signal name window. It also displays the various timing constraints that exist between events on the waveforms. Only two types of constraints are displayed: general timing constraints and ordering constraints. Simultaneity constraints are only displayed when they are violated and appear as a line segment connecting two events (see section A.4).

Only the traces of non-periodic signals can be edited. The shape of periodic waveforms is determined by the period, duty-cycle, and phase-offset specified as signal properties in the signal name window. For synchronous and asynchronous signals, events are specified by the appropriate mouse click on the waveform. Events may be inserted anywhere along an asynchronous waveform, while on synchronous waveforms, events can only be placed near the edge of the clock to which the signal is synchronized. Signals can be specified as being synchronous, in the signal name window, and can be permitted to change logic level at both or either of the falling and rising edges of the periodic waveform. Synchronous events are drawn as don't care levels stretching from one hold time after a clock edge to one setup time before the next clock edge. Asynchronous events appear as a single edge between two different logic levels.

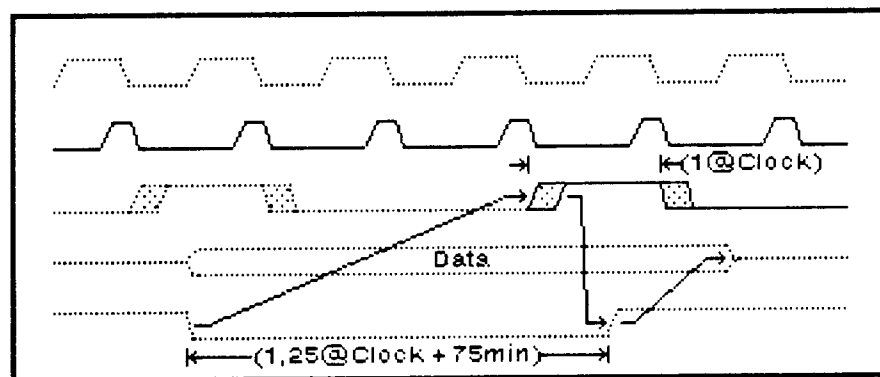


Figure 3.3. An example of a signal trace window. It displays five signal traces or waveforms, three ordering constraints, and two timing constraints. Logic levels can be labeled so that data transferred across the interface can be named.

Logic levels to either side of the event can be changed with the use of a pop-up menu. Levels can also be labeled so that data transferred across the interface can be named. Events may be moved along the waveform and are limited only by the position of adjoining events.

Synchronous events are snapped to the nearest clock edge. Special care must be given synchronous events because of these enforced constraints. When a clock period is changed, any synchronous signals must have their events realigned. To perform this realignment in a semantically correct way, rather than just repositioning the events to the closest new edge (where some may need to overlap), Waves also maintains the same number of cycles between events. This method maintains the relative position of the events relative to the cycles of the clock which is the most common way to think about synchronous signals.

Movements of events and changes in logic levels are always checked for consistency with rise and fall times by the editor and not performed if an inconsistency would arise. Rise and fall times can be specified for the entire signal in the signal name window or specifically for an individual event in the trace window.

Timing constraints can be added directly to the diagram once two events are identified. A pop-up menu is available on each event that permits the user to select one of the three types of event pair constraints: the timing constraint, the ordering arc, or the simultaneity constraint (see Table 2.1). The editor then prompts the user to select the pairing event. This applies to events on periodic waveforms as well.

The timing constraint is added to the diagram after the user specifies time parameters. A general timing constraint has minimum and maximum time parameters. These are expressed as a two-term expression of the form *cycles@clock + time*. The first term is a number of periods of a periodic waveform in the diagram (the @ symbol is read as "cycles of") and the second is an absolute time amount. A similar specification exists for the simultaneity constraint while the ordering constraint does not require time value parameters.

The Time Line Window

3.2.3

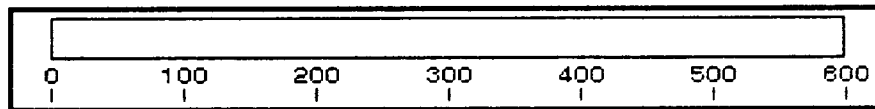


Figure 3.4. An example of a time line window. The horizontal bar represents a single diagram segment. Segments will be discussed further in section 3.3.2. Horizontal scrolling of the diagram is described in section A.5.

The *time line* window is directly above the trace window and displays a time line used to precisely position and align events on the traces. The units of the time line do not correspond to any specific time measure and can be arbitrarily scaled. For the purposes of this dissertation and the example diagram, one time unit will correspond to one nanosecond. The events are positioned by the user in one of the many possible configurations that meet the

timing constraints. The horizontal bar above the time line is used to identify diagram segments and will be discussed further in section 3.3.2.

The Feedback Window

3.2.4

The window directly above the signal name window is the *feedback* window. Its function is to provide information to the user regarding the currently selected view options. These inform the user whether all timing constraints are being displayed (ordering arcs (OA), simultaneity relation (SR), and timing constraints (TC)) as well as other options that are described in section A.5.

The feedback window also displays the current position in time of the cursor as it moves over the trace window. The last position where a mouse button was clicked is displayed as is the difference between that position and the current position of the cursor. This information can be used for precise positioning of events along the time line and measuring distances on the traces.

View Options;
OA SR TC
Last: 28
Crnt: 280 dt: 252

Figure 3.5. An example of a feedback window. It displays the active diagram view options, the position in time of the last mouse click, the mouse's current position, and the time difference between the two points.

The Title Window

3.2.5

WAVES> Interface X -- Operation Y -- Example Diagram

Figure 3.6. An example of a title window for operation Y of interface X. The title of the diagram is *Example Diagram*.

The last window of a Waves diagram is the *title* window. It is a horizontal black bar across the top of the diagram that displays the name and classification of the diagram. A diagram is identified by a three part name: the circuit interface(s) to which it applies, the operation(s) for which it specifies all or part of the event sequence, and the name which uniquely identifies it when more than one diagram is used for the specification of an interface operation. The title

window is also used by the user to obtain the menus of operations that apply to the entire diagram.

The Waves Icon

3.2.6

The *Waves* icon appears on the screen of the Interlisp-D environment. It acts as a portal to all the timing diagrams currently loaded into the virtual memory and also provides access to diagrams stored in files. When the user clicks a mouse button over the icon a menu pops-up giving the user access to all currently loaded timing diagrams.

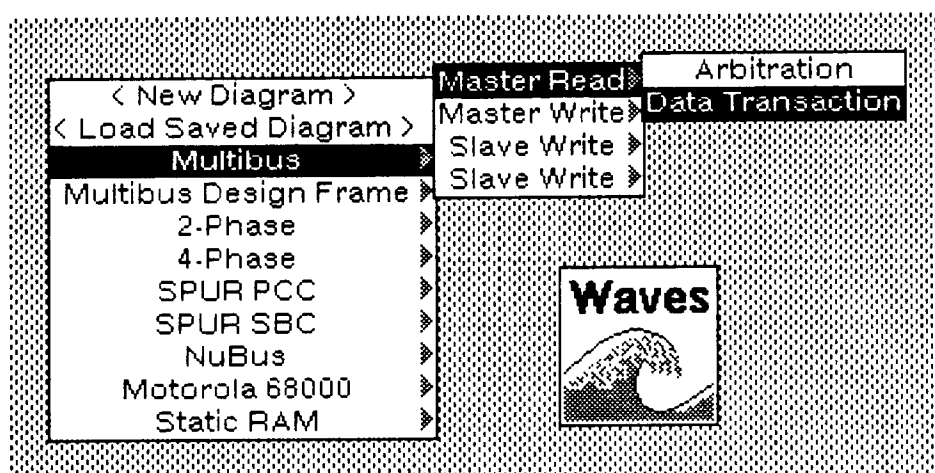


Figure 3.7. The Waves icon and diagram menu. By clicking a mouse button over the icon the user can access all currently loaded Waves diagrams via a three-level menu organized by interface, operation, and title. Diagrams may appear more than once in the menu if they are used to describe more than one operation or interface. In the case above, the Multibus interface and its Master Read operation are selected. There are two diagrams for this operation with the titles Arbitration and Data Transaction (see Figure 1.1).

The menu organizes the timing diagrams into three levels. The top level shows the names of the interfaces for which diagrams have been defined. Two other items are also present: one for opening a new blank diagram and another for loading a diagram that had previously been saved in a file. The second level of the menu shows the names of the operations that have been defined for each interface. The last level simply shows the title of the diagram. These three fields are obtained from the title window of the diagram. A diagram may appear more than once in the menu if it used in the specification of more than one operation (e.g., an arbitration sequence used for both read and write operations) or more than one interface, in the case of similar but not identical interfaces.

The editing capabilities described above permit the specification of simple event sequences and the timing constraints that apply to them. However, this is not all that is required of an interface specification language. In section 2.2, I outlined the minimum requirements for such a specification method. In this section, I will describe how timing diagram notation can be extended to meet these requirements. The basic *Waves* diagrams described in the previous section are inadequate in three areas: representation of arbitrary logic circuitry, specification of conditional and looping event sequences, and combination of event sequences in multiple diagrams.

Representation of Arbitrary Logic Circuitry

3.3.1

Representation of arbitrary automata requires the ability to specify state information and Boolean functions of signals. This is accomplished in *Waves* with a fourth signal direction type, the *internal* signal (\square), and another basic signal type, the *computed* signal (~~\square~~). An internal signal is one that is unobservable at the interface and provides internal interface state. A computed signal is one whose waveform is constrained to be a Boolean function of other waveforms.


Internal versions of all the basic signal types can be defined. A periodic internal signal is a periodic waveform that is internal to the interface and is only used to generate events on output signals. It is not itself an output. Internal synchronous and asynchronous signals can carry arbitrary waveforms and their events can be connected to other events through the usual constraints.

Computed signals are specified by a Boolean function that determines the logic level carried on the wire. However, the Boolean expressions used are more general than standard Boolean algebra. They also include delay and latching expressions, and all three types can be combined into an arbitrarily complex expression. Once the capability for specifying these general Boolean conditions is present, it can also be used to specify when a sequence of events should be started and which of a set of alternative sequences should be followed. This is done by attaching Boolean conditions to specific events. These are used to indicate that the event occurs only if the condition is true.

A delay expression states that a signal is a delayed version of another. It is also used in expressing constraints between events represented implicitly in the diagram and those generated by a computed signal. The delay is specified with the same minimum and maximum time parameters as the general timing constraints.

To express constraints from an event in the diagram to a computed signal, the user specifies the function of the computed signal to include a delay expression. For example, (DELAY (AND A B C) 3@CLK) means that the value of this computed signal is the logical AND of signals A, B, and C delayed by three cycles of CLK. For the reverse, a constraint from an event on a computed signal to an event in the diagram, the user attaches a condition containing a delay expression to the diagram event. The Boolean condition includes the computed signal as a literal. If the condition (DELAY (NOT D) 100) were attached to an event it would be interpreted as a constraint that the event is not to occur unless the signal D was low 100ns earlier. The delay statement can be nested arbitrarily as if it were a Boolean operator, as in (AND (DELAY A 10 50) B), where the signal is the AND of signals A and B, but the effect of signal A on the output is delayed a minimum of 10ns and not more than 50ns.

Latching expressions imply one bit of memory and specify that the level of the computed signal is the output of a latch whose description includes two Boolean functions; one for the input and one for the control signal. A synchronous latching signal can be specified by including the periodic signal in the Boolean expression for the control signal of the latch. (LATCH (OR X (NOT Y)) (NOT (AND ENABLE CLOCK))) specifies that a Boolean function of X and Y is to be latched when the AND of ENABLE and CLOCK is not true. The latching expression can only be used for level-triggered latches, however, it is possible to write a more complex expression for an edge-triggered latch with these primitives. In fact, the LATCH expression is itself a shorthand form for a collection of Boolean expressions that describe each of the gates in a level-sensitive latch.

Computed signals are edited differently than non-computed signals. The only levels to which the user can constrain the waveform of a computed signals are valid, tri-state, and don't care. The exact Boolean value on the wire will be determined by the Boolean expressions. Therefore, computed signals can not be periodic signals unless their Boolean function yields such a result. Asynchronous computed signals carry a signal that is a simple Boolean function of other wires, while a synchronous computed signal () only specifies that the signal will be tri-stated and asserted synchronously. Its value will only be synchronous if the function describing the signal is synchronous. This means that if a latching condition is used then the same periodic signal is used to control the latch and if a Boolean expression is used it includes only signals that are synchronous to the same periodic waveform. The user must be careful that the value of the expression is not used when one of the inputs to the expression may be in a high-impedance state. Waves simply records the Boolean relationship and does not check for correctness.

Interface state and automata are sometimes required to preserve information across interface operations. One example of this is the priority code used to arbitrate for a system bus. An interface may need to sample the value at power-up and then preserve it for later use during data transfer operations. Another example is the addresses of pending operations on a packet-switched bus that need to be saved (i.e., latched) until the acknowledgment packet is received.

Waves supports the specification of interface state through computed internal signals with latching expressions. An arbitrary finite state machine can be specified using internal signals with latching expressions to represent the state bits and computed signals for the combinational logic that determines the outputs and next state. The latch specifies the state memory bit and the Boolean function on the input specifies the next state logic. Moore and Mealy finite state machines can both be described by making the output computed signals functions of just the internal state bits or of both the state bits and inputs. A synchronous or asynchronous machine can be specified by making the corresponding internal signals synchronous or asynchronous.

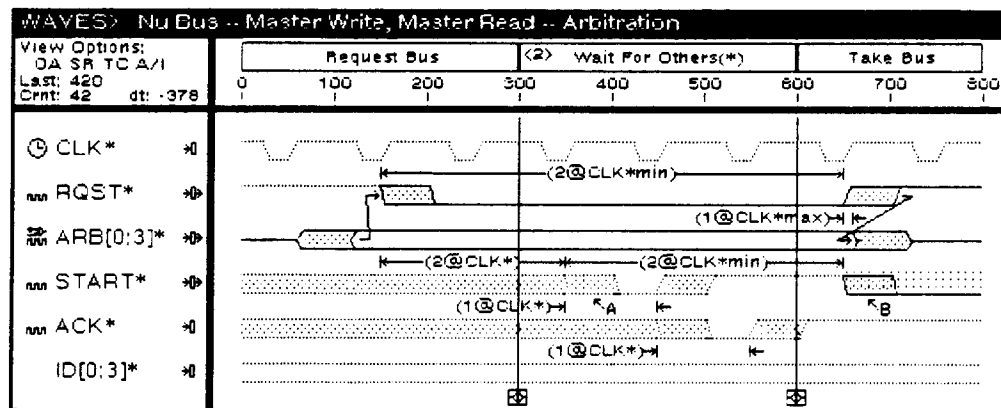


Figure 3.8. An example of a diagram with a computed signal taken from the TI NuBus [TexasInstruments85]. The ARB[0:3]* lines are computed by the following functions taken directly from the specification document:

$$\begin{aligned} \text{ARB0*} &= (\text{NAND}(\text{NOT ID0*})(\text{OR}(\text{NOT ID3*}) \text{ARB3*})(\text{OR}(\text{NOT ID2*}) \text{ARB2*}) \\ &\quad (\text{OR}(\text{NOT ID1*}) \text{ARB1*})), \\ \text{ARB1*} &= (\text{NAND}(\text{NOT ID1*})(\text{OR}(\text{NOT ID3*}) \text{ARB3*}) \\ &\quad (\text{OR}(\text{NOT ID2*}) \text{ARB2*})), \\ \text{ARB2*} &= (\text{NAND}(\text{NOT ID2*})(\text{OR}(\text{NOT ID3*}) \text{ARB3*})), \\ &\quad \text{and ARB3*} = \text{ID3*}. \end{aligned}$$

The ARB[0:3]* lines driven or tri-stated synchronous to a clock edge. However, the logic levels on the wires are not (as can be seen in the equations above). The value on the lines can constantly change while they are being driven. This is used to implement the priority bus arbitration scheme of the Nubus. The labels A and B are explained in the next section.

Ideally, it should be possible to describe arbitrary hardware as part of the interface specifications. A flexible specification language must allow for the integration of existing pieces of hardware into the description. The delay expressions outlined above also help with this requirement. It can be used not only to describe computed signals and the timing constraints on them, but also for specifying a delay corresponding to a propagation delay in already implemented circuitry. This is done by using a delay expression where the minimum

and maximum times are identical (this is specified by a single value as in (DELAY A 10)). This is equivalent to the delay construct common to many HDLs including some of those described in section 2.3.

This collection of escapes to a procedural description language, Boolean, latching, and delay expressions, coupled with internal signals for specifying interface state bits, allow arbitrary circuitry associated with an interface to be attached to a timing diagram. While the mechanisms described are the primitives required for representation of arbitrary circuitry and their constraints, it is obvious that higher level templates could be included to make specification easier for the designer. For example, it should be possible to specify an edge-triggered latch for a signal or that three internal state bits represent a counter without having to include the precise Boolean gates that implement these circuits.

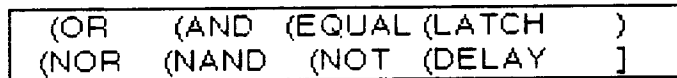


Figure 3.9. The menu used to enter Boolean expressions. It pops-up whenever a Boolean expression is entered or edited. The ten items ((OR, (NOR, (AND, (NAND, (EQUAL, (NOT, (LATCH, (DELAY,), and)) permit the user to specify all supported expressions. Signal names do not need to be typed in by the user, they can be obtained by clicking a mouse button over the name in the signal name window while holding down the shift key.

Conditional and Looping Event Sequences

3.3.2

The second extension is required to describe conditional and looping sequences of events. This is accomplished by breaking a diagram into segments and then applying a regular-expression syntax to the alphabet of diagram segments. A diagram segment is an interval in time of the complete diagram. The diagrams in Figure 3.10 have three diagram segments. The visual representation of this can be seen in the time line window where the single bar of Figure 3.4 is now broken into three segments. Each segment is labeled and the center segment also has a *Kleene star* operator attached to its name. This is the mechanism for describing a looping sequence of events, in this case a *while* loop. *Waves* also supports a specific number of iterations besides the indeterminate number specified by the Kleene star. Conditional sequences are represented by different segments occupying the same interval in time. In Figure 3.10, the two diagrams represent two views on the same diagram and differ only in the last segment. These specify two possible sequences of events for ending the operation.

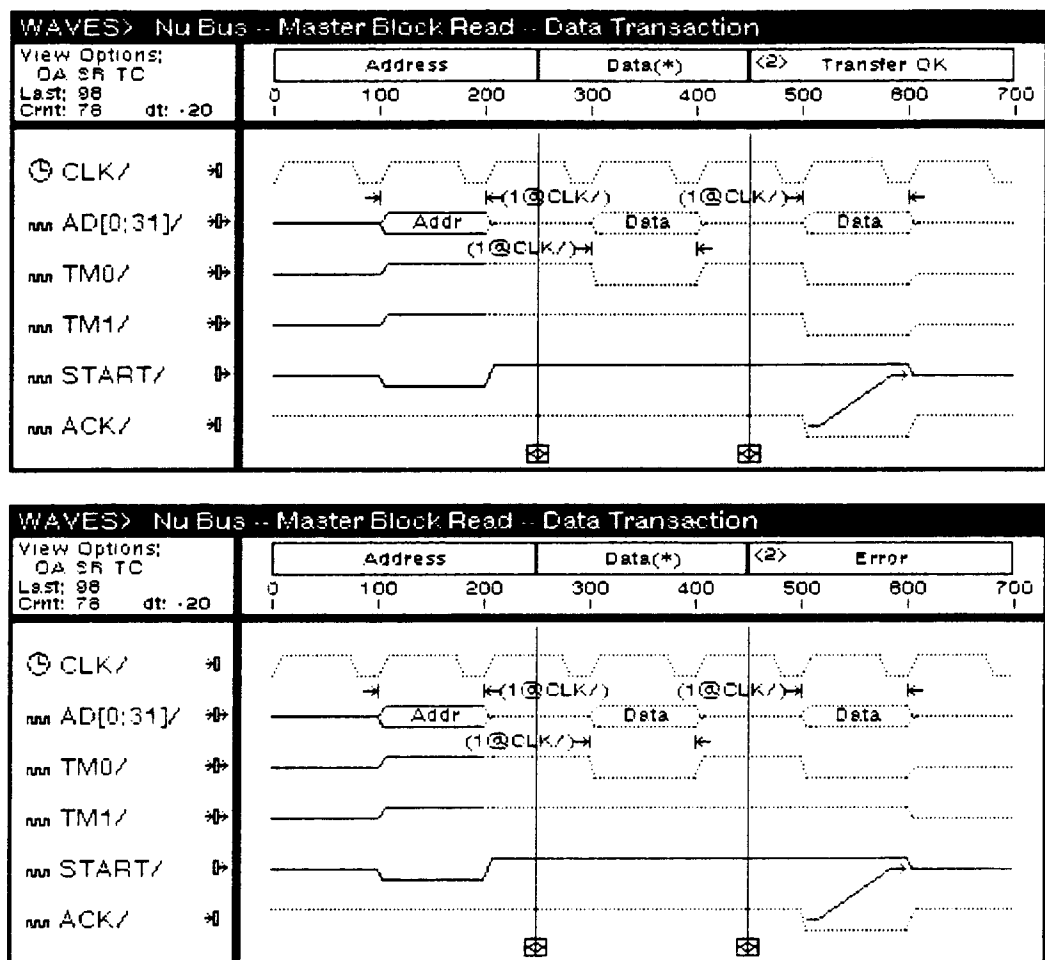


Figure 3.10. Two views of a Waves diagram that demonstrate the use of a regular-expression syntax on diagram segments to represent conditional and looping sequences of events. The center segment of the diagrams has a Kleene-star attached to its name signifying that the sequence of events it contains can occur an arbitrary number of times. The last segment is different in the two views of the diagram and the notation <2> indicates that this is in fact one of two possible alternative segments for its time period. The regular-expression for the diagram would be written as $[\text{Address Data}^* (\text{TransferOK} | \text{Error})]$. The Data segment is enabled by a condition that ACK/ is high on the first TM0/ event in the segment. TransferOK and Error are also enabled by conditions on their first TM0/ events. These are $(\text{AND} (\text{NOT ACK/}) (\text{NOT TM1/}))$ and $(\text{AND} (\text{NOT ACK/}) \text{TM1/})$ respectively. The conditions can be viewed and edited through the use of a pop-up menu obtained by clicking on the events.

Timing constraints can be expressed across segment boundaries. In looping segments, constraints from earlier segments apply to the first iteration of the segment, constraints to later segments apply to the last iteration. If constraints need to be expressed between iterations of the segment itself than two copies of the segment (with identical events and

internal constraints) must be placed in the diagram and identically named. Constraints between events in these two copies apply form one iteration of the segment to the next.

Nested diagram segments are represented by adding a level to the bars of the time line window, as in Figure 3.11. Only the lowest level bars can be used to split and join segments or add levels. The higher level bars are needed only to express looping sequences that include more than one diagram segment. However, they do provide a hierarchical organization for the segments that the user may find useful for purposes of clarity.

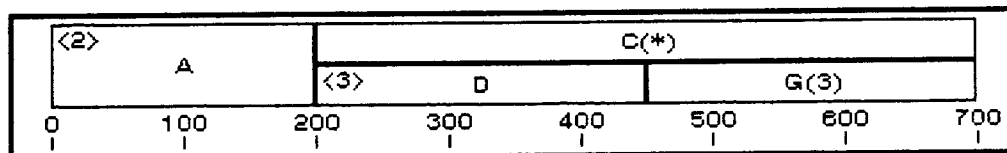


Figure 3.11. The time line window of a *Waves* diagram with nested segments. In this case the regular-expression is $[(A/B)C^*]$. It can also be written as $[(A/B)((D/E/F)G^3)^*]$. The term $(D/E/F)G^3$ is equivalent to C . Only one of each group of alternatives can be viewed at a time. In this case, the view is of the string ADG^3 (or AC^*). Segments B , E , and F are not visible in this view of the diagram. For looping segments, the user can specify a minimum and maximum number of iterations. For example: $(*,5)$ would indicate that the segment may occur 0 to 5 times, $(3,7)$ indicates 3 to 7 iterations, $(2,*)$ means that the segment will occur at least 2 times.

To complete this extension we only need a way to specify how one of two or more alternative sequences is selected. This is done by *segment enablers*. A segment enabler is an event that causes a choice to be made among alternative sequences. The sequence that is selected will occur and those that do not will be bypassed. A segment enabler can be either an event (e.g., a signal going *low*) or a time of occurrence of an event (e.g., a signal going *low within two cycles*). Either the event or the timing constraint can be marked by the user as enabling the segment. The event may also have a general Boolean condition attached to it and this must also be met for the sequence to be selected. For example, in Figure 3.8, the events labelled A and B are segment enablers. The event labeled A enables the *Wait for Others* segment and has a condition associated with it that specifies that it will occur if the $ARB[0:3]^*$ lines are not equal to the $ID[0:3]^*$ lines two cycles after they have been asserted. The event labeled B has the complementary Boolean condition and enables the *Take Bus* segment.

Of course, the enabling conditions must be mutually exclusive or two, possibly conflicting, sequences of events could be enabled. However, they do not need to completely cover all possibilities. *Waves* checks only for conflicts between segment enablers, not for full coverage of all possibilities. Many interface specifications do not specify a sequence of events for every possible condition at a decision point but only for a subset. The other conditions may be known to never occur and therefore the lack of full coverage does not necessarily constitute an error in the specification.

The last extension is needed to support the composability of timing diagrams. It is generally recognized as being useful and more efficient to specify smaller entities and then combine them together into larger ones. The same should be possible for the sequences of events described in timing diagrams. Here the model of concurrent sequential processes is especially appropriate. The user can specify *synchronization points* between diagrams much in the same way as *signal* and *wait* constructs are used in concurrent programming languages. There are four types of *labeled points*: start, end, merge, and order points. Each point has a textual label to distinguish it from other points of the same type.

The start and end points are used to label the first and last events of an interface operation. There can be only one first event for each operation, possibly with a Boolean condition attached, and many last events, the union of which signifies the end of the operation. Multiple last events must be unordered by timing constraints.

Merge points are used to tie together events across different diagrams and to combine two or more event sequences. The sequences can be tied together at more than one point making it possible to *call* a diagram as a *co-routine* from another with the merge points acting as synchronization points. Merge points translate to simultaneity constraints across diagrams.

Merge points can also be used to express timing constraints between events in different diagrams. If two events are on the same signal and are of the same type (e.g., logic 0 to logic 1, tristate to logic 0, etc.), then making them identically labeled merge points causes them to be combined into a single event. In this manner, any timing constraints attached to the events carry over across the two diagrams. In some cases, a constraint may need to be expressed across diagrams without the event actually being part of the sequence in both diagrams. This type of constraint is only meant to apply when the diagrams are both part of the specification of an operation but may also be used separately for other operations. The constraints should only apply when the two are combined. To express these constraints, an event can still be drawn in a diagram and then marked as *inactive*. This specifies that it is not part of the sequence represented by that diagram but rather acts only as a place holder for some inter-diagram timing constraints.

Ordering of events across diagrams can be expressed using ordering points. The difference between ordering and merge points is that an ordering constraint, rather than a simultaneity constraint, is specified between the two events. The event labeled with an ordering point is permitted to occur only after the event with the identically labeled merge point in the other diagram has occurred. Although this could be done using only merge points and ordering constraints, this separate mechanism is required when diagrams from two different interface specifications need to be combined and some dependencies enforced. It should not be necessary to include details of another interface in an interface specification.

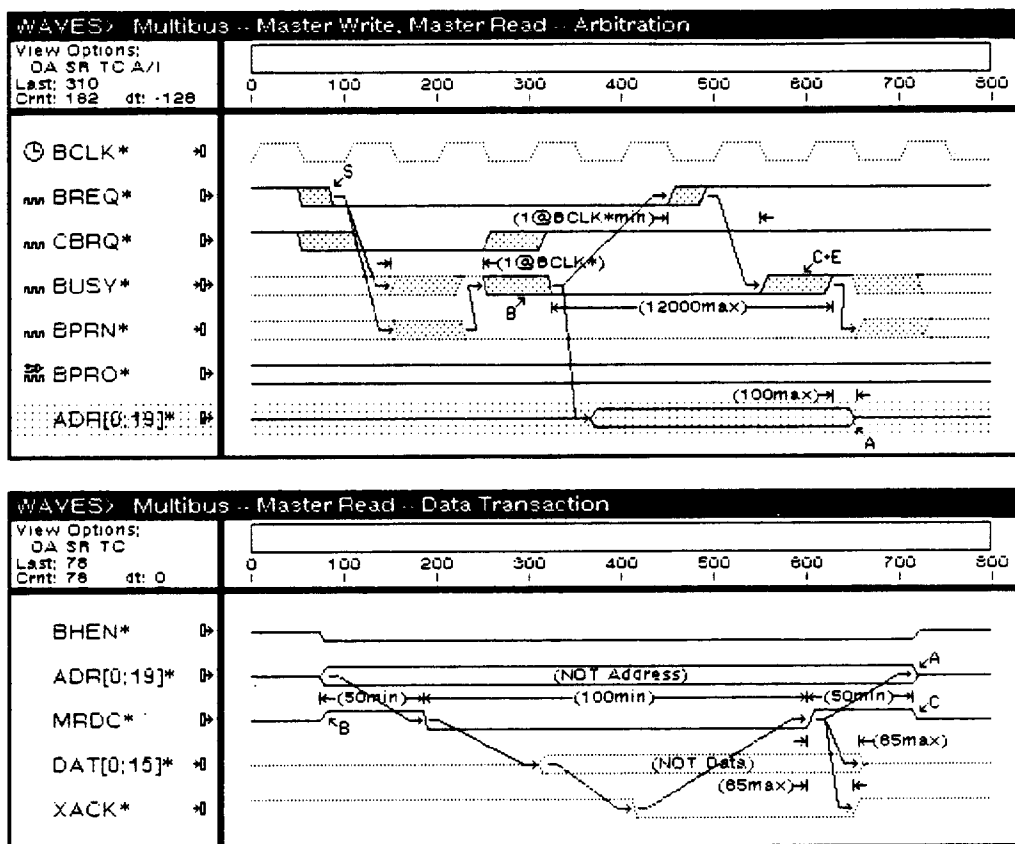


Figure 3.12. Waves diagrams that demonstrate the use of merge points. The two diagrams represent the specification of the Multibus master read operation including bus arbitration. The first diagram specifies the arbitration sequence and the second the details of the data transaction. The letter annotations in the diagrams show where there are labelled points. In the top diagram, there is a start point (S, the first event of BREQ*), and end point (E, the last event on BUSY*), and three merge points that correspond to events in the other diagram. Two sets of merge points are used to merge the assertion of BUSY* and the first event on the command line MRDC* (B) and the deassertion of BUSY* and the last event on MRDC* (C). The other set (A) is used for expressing a 100ns maximum constraint between the time that BUSY* is deasserted to the time the address bus is tri-stated. The signal ADR[0:19]* is shaded to indicate that it is only used as a place holder for the inter-diagram constraint and is not actually part of the diagram. Such a signal (or individual event) is called *inactive*. The second event on that signal is merged with the similar event on the lower diagram (the first event is ignored). The view option to permit the display of the shading of the inactive signal is turned on (see the feedback window annotation A/I).

Waves supports a new and complete interface specification method that serves as a framework for a new class of CAD tools. It can also be a front-end to the other interface specification approaches discussed in section 2.3. The events of the *Waves* diagram can be translated into state graphs or Petri nets quite easily and used to annotate hardware description language code. Each event corresponds to a different state, node in the graphs, or statement in the language. Some researchers using temporal logic and language specifications have suggested the use of timing diagrams as a user interface for logic circuit designers rather than the complex logic notation or one-dimensional language [Granacki86b, Kimura87, Nestor87b]. *Waves* is a specification approach that emphasizes ease of use by applications designers and not just experts in the specification method. This is not the case for the other approaches, especially the formal logic methods.

But *Waves* has other uses besides an interface specification method. Applications range from design and documentation to interface circuit synthesis. *Waves* can serve not only as the front-end to these applications, but it can also generate the different data structures required by the programs. In this section, I will give a short description of how *Waves* specifications can be used by four different types of interface CAD tools. These are only a subset of the tools made possible by virtue of having an interface specification methodology.

Interface Documentation

3.4.1

Waves diagrams can serve as an index to interface documentation and to generate hardcopy specifications in a natural language. The capability of all diagram objects to hold arbitrary text makes this possible. The text is accessible directly through the diagram interface by using the pop-up menus of diagram objects. A text editor window can be used to enter, modify, format, and view the text. Rather than dealing with the index of a specification document, the user can graphically reach the signal, event, or constraint of interest within its proper context.

Documentation guidelines, not unlike those used in many design efforts today, insure that the text has uniform style and formatting. If interface designers can fully annotate a set of diagrams corresponding to all the operations of an interface, then a tool can be developed to automatically generate a hardcopy document describing the interface. Figures and tables can be derived from the *Waves* diagrams and used to elucidate the text description. The text can contain references to objects in the figures and further improve the readability of the description.

Waves is also useful as an aid in the design of circuit interfaces. In the initial stages of design, a system is partitioned into smaller pieces with the complexity of each component typically being less than that of the system taken as a whole. Designers can proceed more rapidly in generating software and hardware implementations for each of the components. However, this divide and conquer approach creates a whole collection of interfaces that did not previously exist and are artifacts of the partitioning process. For this reason, designers must carefully consider the effects of system partitioning. A wrong decision could result in inefficient communication between system components and poor performance.

Waves aids designers in making these decisions by providing a spreadsheet-like way of evaluating changes to the interface. A typical scenario may be as follows. Each of the interfaces in the partitioning is specified using *Waves*. The timing diagrams describe the sequences of events and timing constraints for each interface operation. The constraint checking capabilities of *Waves* are then used to evaluate the effect of changes in the interface. For example, the designer may want to view the effect of changing the clocking scheme for a synchronous component. The clock period can be changed, duty-cycles varied, and signals made synchronous to different clock phases. The effects of these modifications can be seen immediately by observing the violated constraints in the diagram. This is especially useful when two asynchronous components have connecting interfaces. A complementary approach was taken in the Timing Design System [Kara86]. TDS generates a timing diagram, with specific positions assigned to each event so that all the timing constraints are satisfied. This is a useful aid in the synthesis of interface circuitry.

As changes occur during the system partitioning process, the interface specifications can also be used to maintain compatibility among the different components. The misunderstandings or omissions in a verbal or informal description are a common source of design errors in multi-person design teams. Frequently different designers are simultaneously modifying different parts of the design and negotiating the interface details based on the needs of the components being connected. With frequent changes, especially early in the design cycle, it is difficult to keep the many interface details consistent without the help of CAD tools [Katz83]. *Waves* diagrams can be used to check that the interfaces are still compatible after each round of design changes.

To do this, *Waves* diagrams for each operation in each of the two connecting interfaces must be matched. This involves two steps. First, the events in one component's interface must be matched one-to-one with events from the other component's interface. Second, it must be determined if there is a positioning of the events in time that satisfies the constraints of both interfaces.

The matching of events is a restricted graph isomorphism problem. The nodes (or events) are grouped into disjoint sets, with each set corresponding to a different signal and its members partially ordered in time. Interesting problems arise with don't care and synchronous events.

A don't care level on one signal trace must match with any other level. This means that the event that leads to the don't care level may or may not have a corresponding event on the other interface. Synchronous events have a similar problem. If two adjacent synchronous events on the same signal do not have a minimum timing constraint between them, then it may be the case that they overlap (see Figure 3.13). The logic levels of computed signals are evaluated before they are matched. The pairing of events is a classical search problem where tentative decisions about whether two events match are made and then possibly retracted at a later time to try a different matching. Timing constraints can be used to help prune the search to those events that actually occur within a more limited time range. This requires a programming paradigm that supports backtracking (e.g., Prolog). If any logic levels do not match or events are left unpaired then the interfaces are not compatible. The timing diagram user interface permits the program to call the designer's attention to the few events where the matching process failed.

A successful matching of the events does not imply compatible interfaces, however. The timing constraints must also be checked. It must be possible to position the events in time so that the constraints of both interface specifications are satisfied. This requires a general-purpose constraint solver that can process the graph and assign time intervals to each of the events consistent with the constraints. If none of the intervals are empty then the constraints can be met and the interfaces deemed compatible.

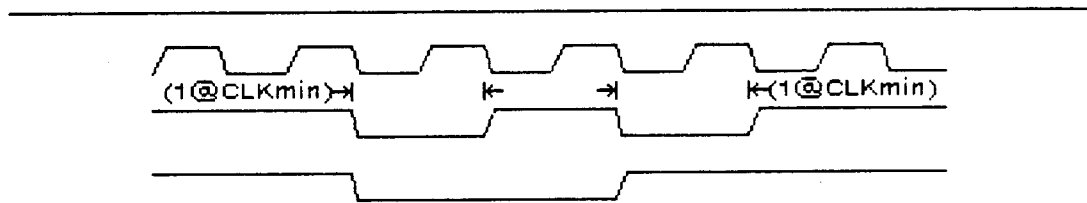


Figure 3.13. The overlap of synchronous events. The middle signal trace has two pulses that are a minimum of one cycle wide. A possible matching signal is shown below it. The two cycles are pushed together overlapping two events and eliminating them from the trace. The compatibility checker must be able to recognize this as a match.

Waves could also be a front-end to interface analysis and critiquing tools similar to those that have been developed for electrical aspects of design [Kelly84]. These tools could call the designer's attention to interface constructs known to be inefficient (e.g., four-cycle handshaking in a synchronous system) and suggest restructuring of the event sequences and constraints.

Simulation and Testing

3.4.3

Once the interfaces have been designed, *Waves* specifications can be used to generate commands for the simulation of the internal circuitry of each component. The diagram

includes all the information required, from the events in the diagram, to generate input signal values for the simulator. The output values, also obtained from the diagram, can be verified as the simulation proceeds. Since time is not a consideration during event driven simulation, the timing constraints need not be considered.

The difficulty in generating the simulator commands comes from the mixing of signal events and periodic events. This is straightforward for synchronous signals whose events are aligned with clock edges but not for asynchronous events or when mixing events synchronous to different clocks. One approach is to simply use the positioning of events in the timing diagram to generate command-time pairs. These can then be sorted in time and used to run the simulation. Difficulties with this approach arise when two or more diagrams with independent time lines are combined. The diagrams may need to be stretched or shrunk in time to properly reflect the interactions of events across diagrams.

Rather than dealing with these issues, the constraint solver outlined above can be used to derive permissible time intervals for all the events in the combined diagrams. The events are each assigned a time of occurrence. The effects of each assignment on the time intervals of future events are propagated and used to make a consistent assignment for the next events. Once this is accomplished the events can again be sorted by time and translated to simulator commands.

The first approach is much simpler to implement. Also, it is easier for the user to understand the results of the simulation because the positioning of the events is reflected exactly in the timing diagram. Responsibility for combining events across diagrams consistently can be left mostly to the user. If the diagrams are already stretched or shrunk to the right size, then the tool need only apply time offsets.

Timing diagrams are widely accepted as an excellent interface to circuit testers. In fact, a waveform editor, similar in many ways to *Waves*, is used as the user interface to Digital Equipment's Knowledge-Based Test Assistant (KBTA) [Arnold85]. KBTA uses a similar algorithm to that of TDS to generate event positions that test the circuit for proper operation within and outside the tolerance specified by the timing constraints.

The issues in generating test vectors are quite different than for simulation commands [DenBeste86]. Testing is not an event driven process but occurs in real-time. Input vectors can be generated from the timing diagrams in the same way as simulation input commands, but output validation is a different matter. As output events may occur at any time within some interval, it is not possible to generate a fixed set of output vectors with which to compare. Rather, the output values must be collected by the appropriate strobes and translated back into waveforms with events identified. These can then be matched with the output waveforms of the timing diagram using the same restricted graph isomorphism algorithm outlined in the previous section. Timing constraints can also be checked in the same way.

This is very different from the way testers work today. Designers specify a full set of input and output vectors for the tester. But these can, by definition, only include one possible set of correct outputs. Many more acceptable output vectors are possible as long as they meet the

timing constraints. The new method outlined above checks that the circuit under test meets the specification, not just one set of possible outputs.

The difference between the specification and one set of vectors is also an issue for the input vectors. It should be possible to use the diagram to generate a whole collection of tests that validate the circuit's behavior relative to the timing constraints. For this to be possible, the input vectors must be generated using the constraint solver and a series of assignments of times to events that exercise the circuit to the limits of all the timing constraints. For example, sets of input vectors could be generated to test the setup and hold time requirements of a synchronous interface. Some testers and simulators are being developed that have these capabilities [Ikos86].

Synthesis of Interface Circuitry

3.4.4

Interface specifications are also useful in the automatic synthesis of circuit blocks. For example, the BSI/ISPS extensions to ISPS enable the synthesis of synchronous circuit blocks while taking interface constraints into account [Nestor86]. *Waves* diagrams can be a front-end for constraint specification in these programs. An ISPS description can be used to generate a timing diagram via a user-controlled simulation. If the events generated by the simulation are automatically labelled with the labels from the language description, then the designer can enter the constraints directly unto the diagram (see section 2.3.1). The connections back to the ISPS code exist through the common labels.

If two interfaces are not compatible, then we can think about developing tools that give the designer hints about how to correct the problem. Ideally, a tool could be developed to modify the two components so that their interfaces are compatible. In the shorter-term, we can look to tools that make the components compatible by generating glue logic to be placed between their two interfaces rather than modifying the internal circuitry.

The synthesis of glue logic is one of the most neglected areas of automatic circuit synthesis. The reason is the difficulty in specifying the interfaces and the complexity of dealing simultaneously with both synchronous and asynchronous signals. *Waves* enables the development of interface circuitry synthesis tools by addressing these two specification issues. A tool for the synthesis of interface transducers, the glue logic that connects two interfaces together, is the subject of the next two chapters.

<This page intentionally left blank.>

PART II

<This page intentionally left blank.>

The glue logic that connects two interfaces is an *interface transducer*. Each interface is described using a collection of *Waves* diagrams. Transducers are predominantly control logic, may include both synchronous and asynchronous components, and must respect many timing relationships on the interfaces to be connected. Traditional synthesis methods are inadequate in the face of these features. I have developed a new synthesis approach based on *event graphs* derived from formalized timing diagram specifications of the circuit interfaces. It can synthesize high-performance transducers comparable in size to designs composed by experienced human designers.

This chapter is composed of three sections. In the first section I define what an interface transducer is and list the features that distinguish this class of circuits. The second section surveys the methods employed in the automatic synthesis of digital circuits and explains why they are inadequate for interface transducers. In the third section, I introduce a new synthesis method, called *Suture*, that addresses these deficiencies.

An *interface transducer* is the glue logic that connects two interfaces. It is important to note that this is more general than connecting two circuit blocks. There may not be a corresponding circuit implementation to one or both of the interfaces. For example, in connecting a chip to a system bus, the chip is a logic circuit block but there is no circuitry corresponding to the system bus side of the transducer.

Figure 4.1 illustrates the concept of the transducer. Two different abstract interfaces are linked together so they can communicate. The communication primitives are events on signal wires that cross the interfaces. Chapters 2 and 3 describe how interface behavior can be specified, including both event sequencing and timing constraints between events, through the use of formalized timing diagrams.

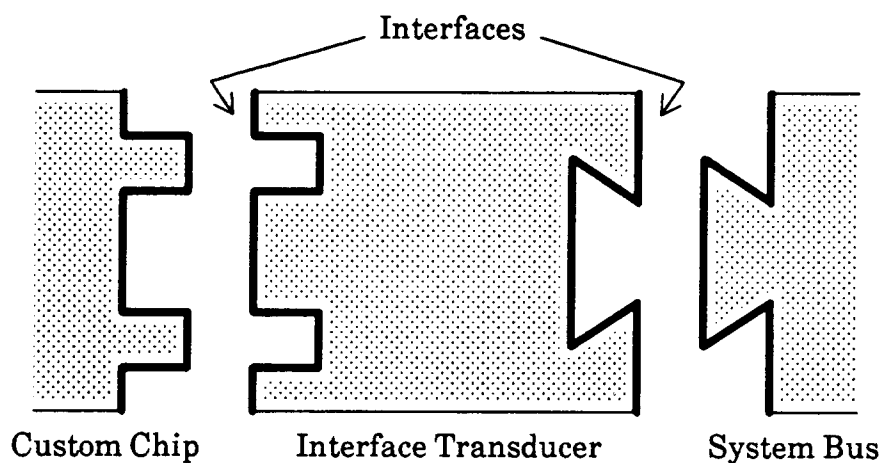


Figure 4.1. An interface transducer connects two interfaces. In this case, the transducer connects a custom chip to a system bus.

Interface Operations

4.1.1

If we view communication across interfaces as the exchange of signal events, then a transducer's function is to *map* a sequence of events on one interface into the *semantically equivalent* sequence on the other interface. The sequences are organized into groups corresponding to *interface operations*. Each operation corresponds to a set of *Waves* diagrams that defines its sequence of events. Semantically equivalent operations are recognized by having the same name (e.g., data read) in both interface descriptions (see section 3.2.6).

Throughout this dissertation interface operations are assumed to be *atomic* entities. This means that a sequence of events corresponding to an interface operation is indivisible (i.e., if it begins, then it will also end before it can begin again). The specification method must permit the description of conditional behavior on the interface to cover all possible ways in which the operation can come to completion (as in *Waves*, see section 3.3.2).

This model of circuit interfaces leads to a two-part specification of transducers. The first consists of the names of the two interfaces that the transducer *connects*. The second is a list of operations, common to both interfaces, that the transducer must *support*. Supporting an operation means that, for the operation, events on one side of the transducer are mapped to the other side. The mapping must respect all timing constraints associated with the event sequences.

For two interfaces to be connected, they must have some operations that are semantically equivalent. This not only means that the corresponding sequences of events have the same meaning but also that if any data is transferred across one interface, it is also transferred across the other. With *Waves*, this means that if labeled data signals exist in a diagram for an operation on one interface then they must be identically labeled for the same operation on the other interface.

Specification of Transducer Behavior

4.1.2

The behavioral specification of an interface transducer consists of all the exchanges of events between the transducer and each interface for every operation it supports. Events must be sensed and generated within specified time intervals corresponding to the timing information in the event sequence specifications. The event sequences are a high-level specification of the interface circuitry. All that is specified is the input/output behavior of the logic; there are no assumptions in the specification of the structural details of the transducer implementation. It is possible to derive many implementations from the same specification.

However, the event sequences are not a complete behavioral specification for the transducer. The sequences on the two sides are completely independent. There are no ordering or timing constraints that can be used to relate them. To specify a complete transducer these event sequences must be interconnected.

Interconnection is accomplished through both explicit and implicit ordering relationships. *Explicit ordering* can be specified by declaring in the transducer specification that one event on one side must occur before another on the other side. This is done using techniques similar to those used to order events on the same side and was discussed in section 3.3.3. *Implicit ordering* can be obtained automatically by observing the data dependencies that exist across the two interfaces. Data that is input on one side of the transducer must be available before it is used as an output on the other side. This fact is used to generate ordering constraints that connect events on the two interfaces.

Another level of behavioral specification for transducers entails the description of the interactions among event sequences of different operations. For example, some interfaces may allow different operations to be in progress concurrently. This aspect is not considered in this dissertation. It will be assumed that the event sequences of different operations do not overlap in time as far as one transducer is concerned.

This assumption is not as serious a limitation as it might seem. Most system busses conform to this model by being a shared resource that can only be used by one system component at a time. This eliminates the possibility of overlapping operations. Even packet-switched busses can be expressed this way by representing the request and acknowledge parts of a data read or write as separate operations. The interface is specified to include the internal state (i.e., addresses) needed to match a request with the correct acknowledge packet. The circuits commonly implemented as custom chips also conform easily to this model. Difficulties arise when finer-grain interfaces between small circuit blocks (e.g., individual gates and switching networks) are considered. These are predominantly combinational logic blocks that continuously operate on their inputs as opposed to responding to and generating sequences of events with well-defined start and end points. Only the larger-grain interfaces encountered in subsystem integration are within the scope of this dissertation.

Automatic Synthesis of Transducers

4.1.3

Interface transducers are an important class of circuits that are required whenever a component is connected to another component, inserted into a system, or moved to a different system. The proper design of transducers is critical to overall system performance. Since they form the communication paths of the system, bandwidth must be kept as high as possible. Ideally, it should be limited only by the internal circuitry of the components being connected. Therefore, a transducer must not only be logically correct, but must perform its functions as quickly as the interface constraints will permit.

To achieve maximum bandwidth, communication may be performed asynchronously. In a large system, this is almost certainly the case, as clock skew among system components makes it virtually impossible to keep the entire system synchronous to a single high-speed clock. Asynchronous designs are also used in smaller systems so that signal events can proceed as quickly as possible rather than being constrained to occur near edges of the system clock.

These considerations contribute to the different nature of transducer circuits when compared to the internal circuits of typical logic blocks. Transducer designs have four distinguishing features: (1) they tend to be control intensive with only modest data paths, (2) they typically include both asynchronous and synchronous circuits, (3) circuitry may be included to delay some signals so that interface constraints are met, and (4) performance is usually a more important consideration in their design than circuit size and complexity.

These features make the design of transducer circuitry difficult for the applications designer (i.e., the expert in the internal logic of a circuit block, not the interface to which it is to be connected). Furthermore, the conflict between performance and satisfaction of the interface constraints makes the design highly error-prone. This is due to the many details that must be considered simultaneously and which may have global implications. For example, responding to an asynchronous signal by synchronizing it and feeding it through a finite state machine may be much slower than a small amount of asynchronous circuitry that operates on the signal directly.

An automatic synthesis method that generates transducer logic from high-level specification of the interfaces can help eliminate these types of design errors. Also, automated design of transducers can substantially reduce the time needed to integrate custom chips into a computer system and thereby make the entire design-test-evaluate-redesign cycle more efficient.

In the next section, I will discuss why previous high-level synthesis methods are inadequate for interface transducers. Only the logic circuit and timing design of transducers will be considered. While for a truly complete synthesizer, one must also consider issues at the electrical, board design, and system partitioning levels. These issues are tangential to much of the discussion of this and the next chapter, and are beyond the scope of this dissertation. The last section of the chapter will present a new synthesis approach that can be used to automate the design of general interface transducers. Chapter 5 will describe the details of an application of this synthesis approach for an interesting sub-class of transducers, namely, those with non-overlapping atomic operations.

There are many ways of describing digital circuits. They can be classified into three main categories or *domains*: behavioral, structural, and physical. Within each domain descriptive methods are distinguished by the *level of abstraction* they emphasize. The Gajski-Kuhn Y-chart can be used to map descriptions along these two dimensions [Gajski83]. The three vectors of the Y-chart correspond to the three domains and the relative position of a description along its axis corresponds to the level of detail in the description (see Figure 4.2).

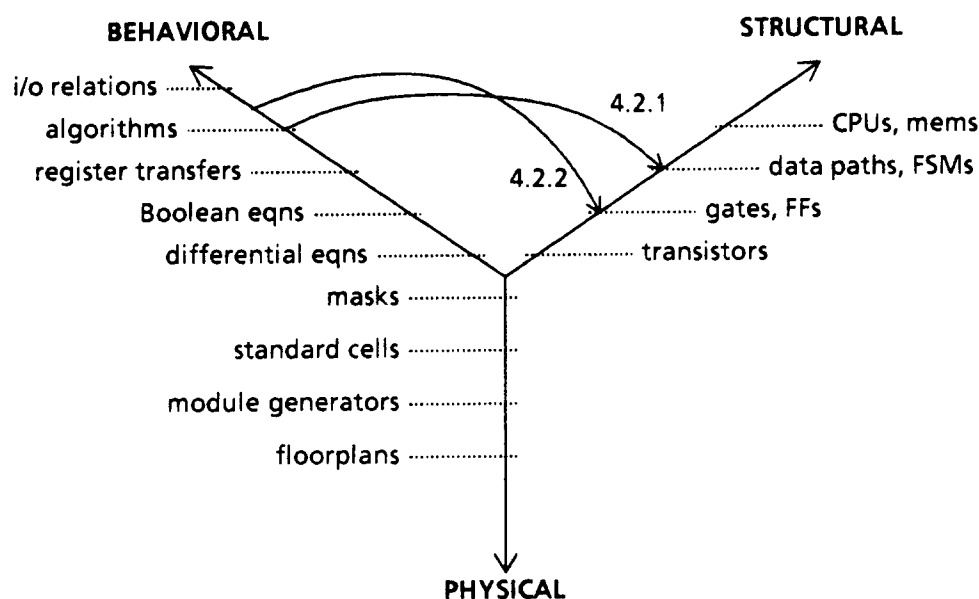


Figure 4.2. The Gajski-Kuhn Y-chart's three axes correspond to three different domains for describing designs: behavioral, structural, and physical. The position of a description on an axis corresponds to the level of abstraction. A description is more detailed the closer it is to the center of the Y (adapted from [Walker85]). The arcs shown on this diagram represent the two approaches to transducer synthesis described in this section.

Synthesis problems can be characterized as transformations from one description into another. In the general case, the two descriptions may reside at different levels of abstraction on different axes. In this chapter and the next, I will be discussing what is usually called *behavioral synthesis*, the automatic transformation of an abstract behavioral description of a circuit into a more detailed structural description.

In this section, I will describe the major features of two behavioral synthesis methods that relate to the task of automatically synthesizing interface transducer logic. They are

distinguished by the levels of abstraction of the behavioral description which they use as input and the structural description they generate as output.

The first class of synthesis methods begins with an algorithmic description of the circuit's function. Traditional HDLs, such as ISPS, support this type of description (see section 2.3.1). This approach emphasizes synchronous designs and generates a structural description containing specifications of data-paths and finite-state-machine controllers [Thomas83].

The second class uses a more abstract behavioral specification that describes the mapping between circuit inputs and outputs. I have placed this level above the algorithmic description because it does not specify the details of the algorithm to be used to perform the mapping. Obviously, this cannot be done for all circuits, but it is an abstraction level appropriate for interface transducers. Graph-based specification methods like those of section 2.3.2 can be used for this type of input description. The structural description generated by this class of synthesis methods is also different. The elements are more primitive logic blocks corresponding to registers (i.e., flip-flops) and combinational logic (i.e., logic gates). This mapping has been used to synthesize small asynchronous designs [Chu86a, Chu86b].

The remainder of this section will describe both of these synthesis methods in detail. The discussion will be focused through an example from the relevant literature of each approach. These will help in the description of the algorithms and in demonstrating their deficiencies. In section 4.3, I will present a new synthesis method that does not suffer from these limitations. The examples will be reviewed in the context of this new approach.

Synthesis from Algorithmic Specifications

4.2.1

Synthesis methods that begin with algorithmic specifications as input have historically focused on synchronous designs. In fact, most of the work has been in the area of processor design. There are four basic steps in the transformation of an HDL description to the structural domain [Thomas83]. First, the HDL specification is compiled into a *control and data-flow graph* (CDFG) [Girczyc85]. The nodes of a CDFG correspond to *operations*, while the arcs correspond to data *values* that are the inputs and outputs of the nodes. Second, the graph is *scheduled*, that is, the execution of each operation is assigned to one or more time periods. This is consistent with the limitation of this approach to synchronous systems. Third, hardware modules are *allocated* to perform all the operations and establish the necessary interconnections. Lastly, the operations and values are *bound* to hardware modules. This is a many-to-one mapping as more than one operation may be executed by the same hardware module (e.g., addition and subtraction performed by the same ALU) and more than one value held in the same register (e.g., if they are used at different times in the schedule) or carried by the same wire (e.g., with the use of a multiplexor to choose which to carry at a specific time).

Trade-offs between speed and area are made during the scheduling and binding steps. If speed is the primary consideration, then more hardware modules can be used to perform more

operations in parallel. The algorithm executes in less time but at the cost of a larger area due to the duplication of hardware modules. If the emphasis is on design size, then the synthesis algorithms must be capable of finding the fastest scheduling given limited hardware resources.

However, in practical designs, the trade-offs between area and speed are more complex. A designer usually requires the ability to express timing constraints on the execution of an algorithm or its constituent parts. Also, when interfacing with other system components, interface circuits may have to meet some minimum or maximum timing constraints. Until recently, these timing considerations were largely ignored by scheduling algorithms [Girczyc85, Nestor86].

Timing constraints change the nature of scheduling algorithms. Rather than optimizing the schedule over the entire set of operations, the algorithm must optimize under limitations imposed by the timing constraints. Previously, only data dependencies, available hardware resources, and sequencing constructs were considered in arriving at a final schedule. With the introduction of timing constraints, the range within which an operation can be placed is limited by timing constraints between it and other operations.

There have been several approaches to this more complex scheduling problem [Girczyc85, Parker86, Paulin87]. However, all of these have concentrated on maximum timing constraints. This is the most common type encountered in processor and data-path intensive designs where a constraint on the execution time of each instruction is the primary concern. In interface circuitry synthesis both minimum and maximum timing constraints must be considered.

ISYN is an extension of the CMU-DA synthesis tools that deals specifically with interface circuitry [Nestor87]. It incorporates a variant of a list scheduling algorithm used in microcode compaction to support minimum and maximum timing constraints [Fisher81]. The algorithm, called CSTEP, uses a priority function that determines the order in which operations are scheduled. Operations are assigned a priority at each time step. Those with minimum timing constraints are assigned a negative priority, delaying their placement until a later step. Those with maximum timing constraints that would be violated if the operation were to be placed in a later step are placed immediately. The remainder are placed according to the rules of data dependency and resource allocation, as before. The completed schedule is then used to derive a finite-state-machine that will control the elements of the design.

A BSI/ISPS specification of an interface transducer is shown in Figure 4.3. The transducer is between a simple synchronous interface and the Intel Multibus. The operation supported by the transducer is the same asynchronous Master Read used as the main example of section 2.3.1. The transducer specification is composed of two procedures. The first, *Adapter*, is the main procedure. It waits for a control signal, *exgo*, to be asserted and then calls the read operation on the other interface, *MasterRead*, using the value of the *exadr* lines as the argument to the procedure and asserting the value returned by the procedure on the *exdato* lines. The operation of the transducer is synchronous to a clock signal that is specified in the port declaration portion of the specification (not shown in Figure 4.3). The only signal

declared to be asynchronous is the *xack* signal on the Multibus side. Also, timing constraints have only been specified for the Multibus side of the transducer.

```

main Adapter := begin
    WAIT( INPUTP( exgo ) ) NEXT
    OUTPUTP( exdato, MasterRead ( INPUTP ( exadr ) ) );
    OUTPUTP( exack, 1) NEXT
    OUTPUTP( exack, 0)
end

MasterRead ( Address<23:0> ) <15:0> := begin
    OUTPUTP( bhen.l, TS.ENABLE ) {L:mr0e};
    OUTPUTP( adr.l, TS.ENABLE ) {L:mr1e};
    OUTPUTP( mrdc.l, TS.ENABLE ) NEXT
    OUTPUTP( bhen.l, 1 ) {L:mr0};
    OUTPUTP( adr.l, Address ) {L:mr1} NEXT
    OUTPUTP( mrdc.l, 1 ) {L:mr2} NEXT TNEXT() NEXT
    WAIT( INPUTP( xack.l ) ) NEXT
    MasterRead = INPUTP( data.l ) NEXT
    OUTPUTP( mrdc.l, 0 ) {L:mr3} NEXT
    OUTPUTP( adr.l, TS.DISABLE ) {L:mr4} NEXT
    OUTPUTP( bhen.l, TS.DISABLE ) {L:mr5} NEXT
    OUTPUTP( mrdc.l, TS.DISABLE ) {L:mr6} NEXT
end

mrTenb:    time( mr0e, mr0 ) GEQ 0ns;
mrTena:    time( mr1e, mr1 ) GEQ 0ns;
mrTbs:     time( mr0, mr2 ) GEQ 50ns;
mrTas:     time( mr1, mr2 ) GEQ 50ns;
mrTbh:     time( mr3, mr4 ) GEQ 50ns;
mrTah:     time( mr3, mr5 ) GEQ 50ns;
mrTdis:    time( mr3, mr6 ) GEQ 50ns;
mrTcmd:    time( mr2, mr3 ) GEQ 100ns;

```

Figure 4.3. Example BSI/ISPS specification of an interface transducer (adapted from [Nestor87]). The two interfaces are a simple synchronous interface and the Intel Multibus. There is only one supported operation, the master data read operation. The synchronous interface asserts a control signal (*exgo*) with an address (*exadr*) and expects data to be returned (*exdato*) and signaled by another control signal (*exack*). There are declared timing constraints only for the Multibus side of the transducer.

A CDFG, or VT-body in the terminology of CMU-DA, is derived for each of the two procedures in the specification. Each VT-body is scheduled separately using the CSTEP algorithm. There is no optimization in the scheduling across VT-body boundaries. The final schedules are shown in Figure 4.4. The dashed horizontal lines separate the operations into different time periods.

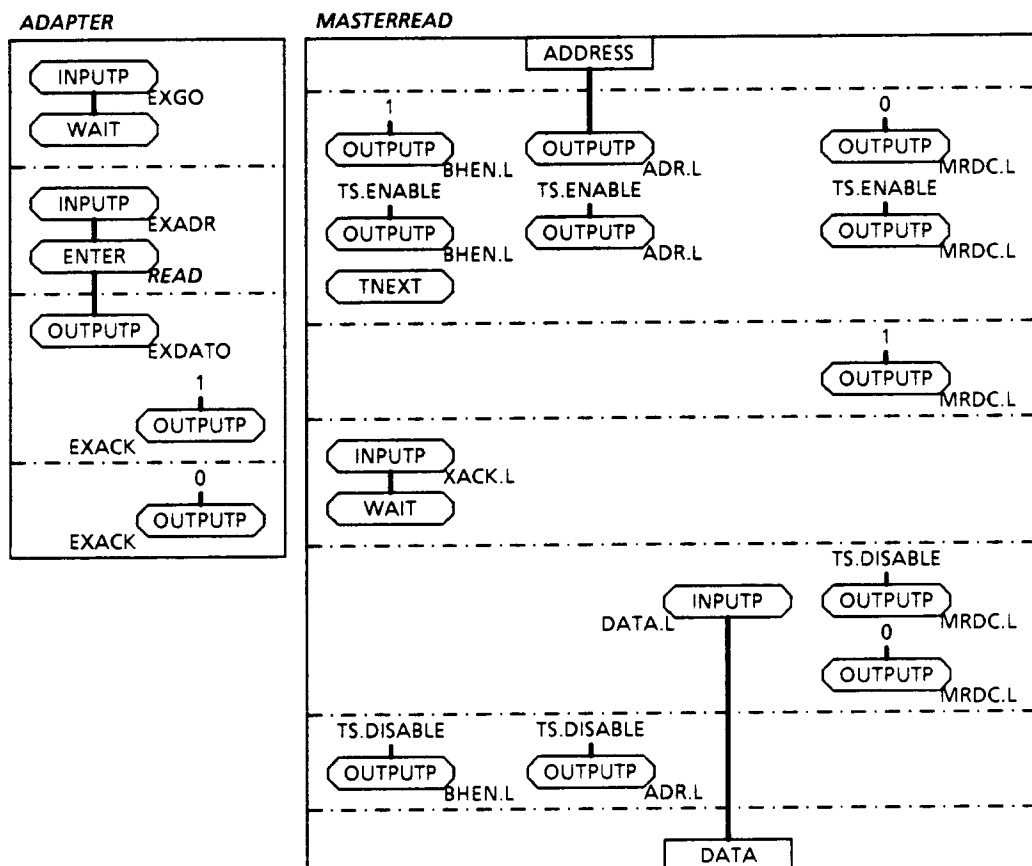


Figure 4.4. Scheduled VT-bodies for the specification of Figure 4.3. There is a one-to-one correspondence between a VT-body and a basic-block in BSI/ISPS. The nodes in the graphs correspond to operations in the HDL description. The solid lines connecting nodes represent data dependencies. Horizontal dashed lines separate the time steps of the schedules. The rectangular nodes at the top and bottom of the VT-body for *MasterRead* are the input argument to the procedure and the return value, respectively. Each VT-body is scheduled separately using data dependency, sequencing (TNEXT), and timing information. No schedule optimization is performed across VT-body boundaries..

For this simple example, the allocation and binding steps are trivial. Each of the input and output ports of the transducer is allocated a buffer, register, or synchronizer depending on its declared type. The control logic for loading the registers and generating the proper events on the output signal lines is derived from the schedule, which can be viewed as a state diagram

description of the finite-state-machine controller. Since all the signals are synchronous to the same clock, it is straightforward to include all the combinational logic and state bits within a single block. A block diagram of the synthesized circuit is shown in Figure 4.5.

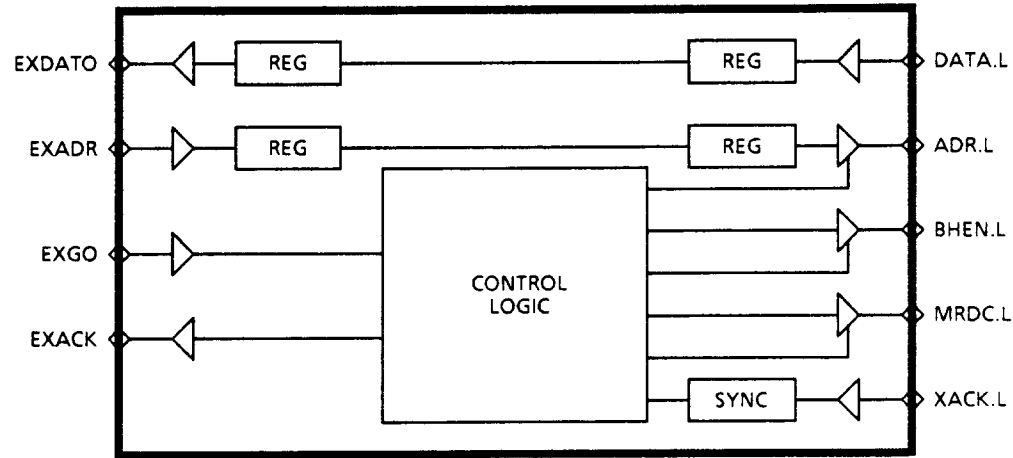


Figure 4.5. Synthesized circuitry for the VT-bodies of Figure 4.4. Input and output ports are allocated buffers, tri-state drivers, synchronizers, and registers (if they carry data). All the control logic is lumped into a single synchronous finite-state-machine.

There are three major shortcomings to this approach. First, and most important, is the lack of optimizations across VT-bodies. The transducer circuitry is not permitted to use both interfaces in parallel. Data must first be collected on one side, then the operation is executed on the other side and the return data collected. Finally, the return data is transferred on the other interface. This artifact of the synthesis procedure cannot be eliminated by in-line expansion of the procedures because it is not clear how statements from each procedure can overlap in time. The sequencing constraints on one side restrict the sequencing of the other side as well.

The second deficiency concerns asynchronous signals. Since the entire circuit (finite-state control and registers) are synchronous, all asynchronous signals are immediately synchronized before being used. In the example above, this means that although the *exadr* lines could be deasserted 50ns after the *mrdc* line is deasserted, they are not deasserted for a full clock cycle (a parameter of the other interface). In larger examples more substantial performance improvements may be possible if some events can be generated solely using asynchronous logic rather than first synchronizing each signal.

The last deficiency concerns the interaction between the specification and synthesis methods. Since every statement in the BSI/ISPS description translates to a VT construct, a complete specification of the interface events cannot always be included. In the example of Figure 4.3, there is no specification that the *xack* line will be deasserted within 65ns after *mrdc* is deasserted (in line 19). If a *wait* statement were to be used to express this then its corresponding VT node will generate an extra state in the control logic to test that this

condition is actually met. Furthermore, since the interface specification is embedded in the circuit description, the specification of two transducers with one interface in common can not share a single description for that interface. Different specifications are required so that all the interactions between the interface and the internal circuitry are tuned for each specific transducer.

Synthesis from Input/Output Specifications

4.2.2

Some of these problems can be alleviated if the synthesis procedure begins with specifications of the input/output event sequences. The internal behavior of the circuit is not specified, instead the specification consists of a partial ordering of the transitions on the input and output signals of the circuit. This eliminates the problem of embedding the interface details within the internal circuit specification. Also, since the description is not subdivided into separate procedures, there are no artificial boundaries that make it difficult to optimize. However, most attempts at this type of specification have emphasized either asynchronous or synchronous behavior, and never included both types.

Variants of Petri nets have been used for input/output event specification (see section 2.3.2). In fact, synthesis methods have been developed to translate each element of a Petri net description (i.e., the places, transitions, forks, and joins) into hardware primitives [Misunas73]. However, these methods tend to generate circuits that are quite complex. The overhead incurred for the ability to directly translate a general graph causes the circuit to be much larger than necessary. For example, a place in a Petri Net might be only a simple change in one output signal but in the direct transformation scheme this corresponds to an entire flip-flop (i.e., state bit). The flip-flop indicates the presence of the Petri net token in its corresponding place. Successor places and transitions are enabled (and predecessors disabled) based on the value of this flip-flop. The logic level of the output signals are derived from the flip-flops assigned to each place.

Another approach is to consider each transition on a signal wire as a transition to a new state and then translate the resulting state diagram into sequential logic [Hollaar82]. This method, when applied to asynchronous designs, suffers from similar problems of unnecessary complexity. The same one-hot state assignment (i.e., one flip-flop per state) is used with the same results on circuit size. Of course, the problem is easier for completely synchronous state diagrams. Since there are fewer race conditions and no hazards in synchronous machines, optimized state assignment and Boolean minimization algorithms can be used to obtain more efficient designs [DeMicheli85, Rudell87].

This type of behavior specification, using Petri nets or state diagrams, has been common in the description and synthesis of self-timed circuits [Molnar85, Chu86a]. Self-timed circuits are speed-independent designs where the relative timing of events is not important but rather only their proper ordering. If some simple restrictions are placed on the topology of the nets then efficient synthesis methods are possible. There are two important properties: *liveness*

and *persistence*. A net is live if all the transitions for a signal lie on a cycle and alternate between rising and falling transitions. This implies that the specified signal transitions are actually implementable and the circuit returns to the same state after some time. A signal is persistent if once a transition is enabled, it cannot be disabled by another event and must eventually occur. Persistence is a property that must hold for systems modeled as Petri nets [Chu87]. These restrictions have posed no limitations on self-timed systems because most follow a well-behaved signalling protocol such as the four-phase two-wire handshake. One module signals it is ready by raising a request line which is acknowledged by an output signal from the other module. Once the acknowledge is detected by the requesting module it lowers its request line to prepare for the next request. The acknowledging module then does the same by lowering the acknowledge line. This protocol is guaranteed to generate a graph that will meet the liveness and persistence requirements.

Once a live and persistent graph is obtained, it is transformed into a state transition graph where there are as many state-bits as input and output signals. These types of graphs have been called encoded interface state graphs (EISG in [Molnar85]) or signal transition graphs (STG in [Chu87]). Arcs in the state graph correspond to transitions (i.e., a change in state) on one of the signals. The state graph is used to build a Karnaugh map for each of the outputs. The entries of the map correspond to the new value of the output variable in its new state. If the state has an arc corresponding to a transition on an output then the signal is *unstable* with respect to that output signal and the entry in the map is the complement of its current value (i.e., a transition in its value). If the state does not have an arc then it is *stable* and the entry in the map is the same value as the signal in the current state (i.e., no transition). If two states in the graph have the same state encoding then a new internal signal is created and transition arcs are added to the graph so that this new signal will have a different value in the previously identically encoded states. Logic equations are then derived from the Karnaugh maps to implement each of the output signals.

An example STG specification, for a FIFO stack control cell, is shown in Figure 4.6. When the cell receives a request for a data element to be shifted from the previous cell it initiates a request to the next cell. This makes room for the incoming data. Once the next cell acknowledges the transfer, the data can be loaded into the cell and the original request acknowledged.

The circuit generated from the STG specification is shown in Figure 4.7. There are three component circuit blocks, one for each unique output: A_i , L , and R_o (D is identical to R_o). The logic equations for R_o and A_i are of the form $Q = DG + QG'$, where Q is the output signal. These can be replaced by a D-type flip-flop with input D and control signal G (or CLK in Figure 4.7). Similar substitutions for other flip-flop types can be made whenever an output signal equation includes the output signal or its complement as one of its literals.

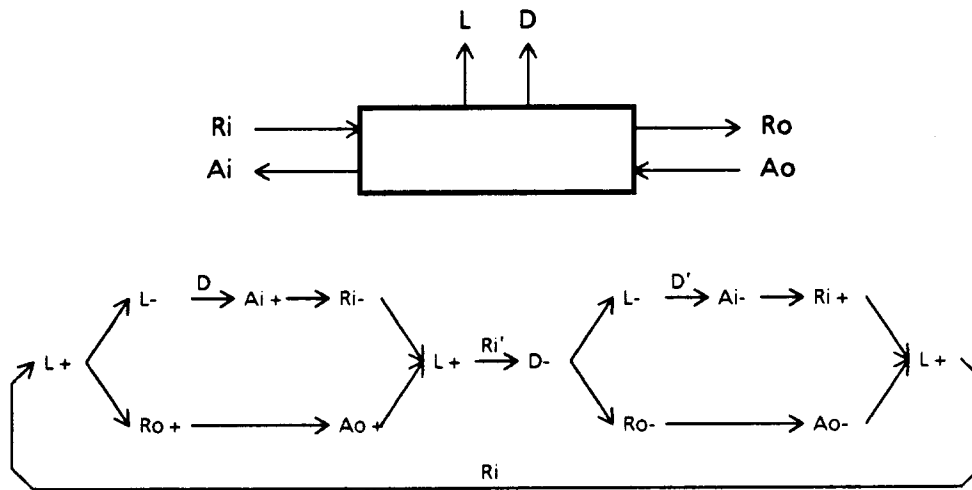


Figure 4.6. Example STG specification of a circuit block (adapted from [Chu86a]). This cell is the control for one word of a FIFO stack. There are two pairs of request/acknowledge lines to communicate with neighboring cells and two other control signals (L and D) to control the latching and storing of data. The symbols + and - appended to the name of a signal signify a rising or falling transition. The symbol ' is used to signify a complemented value.

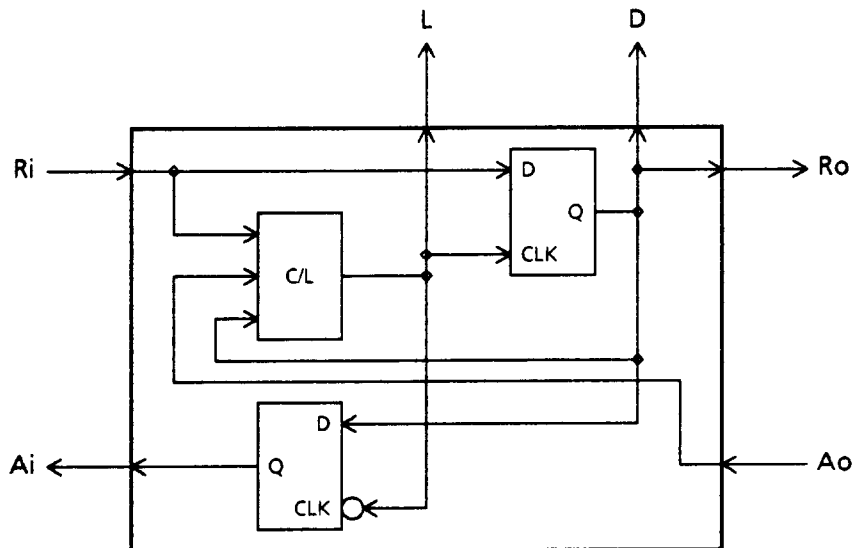


Figure 4.7. Synthesized circuitry for the STG specification of Figure 4.6. The Boolean equation for the combinational logic block is: $L = DA_0R_1' + D'A_0'R_1$. Two D-type level-triggered flip-flops are used to implement the equations for A_i and R_0 .

There are two major shortcomings with this approach. The first is that timing constraints are not considered. They would greatly complicate the process of deriving a state transition graph from the Petri net specification. This limits the method to self-timed systems where this is not a serious limitation because components of these systems typically use the same handshaking protocol to communicate (e.g., four-phase handshake). The problems associated with integrating these systems with off-the-shelf components that do not follow the same protocol and require particular timing relations between events are not addressed.

The second shortcoming concerns the property of persistency. Again, well-behaved protocols inherently meet this requirement. However, many practical interfaces, including most bus interfaces, do not. For example, whether an event or another occurs may signal a different way of completing an interface operation. A synthesis method is required that does not impose this restriction and can handle conditional sequences of events.

The deficiencies of the above approaches make them inadequate for transducer synthesis. To reiterate, transducers require a synthesis method that can (1) handle timing constraints on signal events, (2) synthesize both synchronous and asynchronous components, and (3) focus on generating circuits with as close to the maximum communication bandwidth as possible. None of the methods of the previous section meet all these requirements. In this section, I will briefly outline a new synthesis method, called *Suture*, that does.

The *Suture* method has four novel aspects. The first is that *Suture* does not directly synthesize a complete and correct circuit. Rather, it first constructs a skeletal circuit that contains all the crucial elements to implement the specification of the transducer (i.e., to generate the appropriate transitions on the output signals). However, the circuit may be incorrect. It may violate some timing constraints and contain race conditions. These problems are corrected in later steps by local modifications of the skeletal circuit.

Second, *Suture* supports both synchronous and asynchronous events and is not biased toward one type or the other as are previous methods. Specifications can be fully synchronous, fully asynchronous, or a mixture of the two. Many real systems combine the two types of signals when optimizing communication across an interface. A general synthesis method must support both.

The third novel aspect is that *Suture* synthesizes the transducer from primitive elements rather than using complex components that implement higher-level functions directly. For example, flip-flops, latches, and random logic are used to implement finite-state machines rather than programmable logic arrays (PLAs). The resulting circuit can still be optimized by the standard methods but has a major advantage, namely, that the initial circuit structure is simple enough to easily accommodate the local changes required to satisfy timing constraints. Also, by basing the synthesis method on primitive logic components it is easier to ensure that the method is independent of the implementation technology.

Lastly, *Suture* emphasizes communication bandwidth over circuit size because a transducer must add as little delay as possible to the communication path between two interfaces. Events should occur as quickly as the timing constraints and data dependencies of the interfaces will allow. Furthermore, *Suture* only generates fully static designs. The improved robustness in dealing with both synchronous and asynchronous signals and the added flexibility for low-speed testing more than justify the larger circuit size implied by this choice. Unlike internal elements of circuit blocks, transducer logic is typically not replicated within a design. Since only one instance of this logic is used, a slightly larger design for an interface transducer is usually acceptable while a smaller but slower and less robust design may not be.

The *Suture* method forms the core of *Janus*, a synthesis tool for the logic design of interface transducers. The algorithms in *Janus* and *Suture* will be described in detail in the next

chapter and their implementation in Appendix C. However, the *Suture* method can stand alone and be applied to the synthesis of arbitrary control logic. The remainder of this section is devoted to providing an overview of this approach and is not intended to include a complete description of the algorithms, this is relegated to Chapter 5. The overview is followed by a summary of the results of its application to three examples.

Overview of the Method

4.3.1

The input to the *Suture* method is the specification of an event sequence. The output is a logic specification of a circuit that will have input/output behavior as specified by the input. *Waves* diagrams can be used to generate the input data structures for *Suture*. Diagrams are translated into event graphs that are similar to that of Figure 4.6 (see section 5.2). The nodes of the graph correspond to signal transitions and the arcs to two types of constraints, the ordering and min/max timing constraints. Simultaneity constraints cause their paired nodes to be grouped into super-nodes.

The synthesis method can be divided into four steps: skeletal circuit construction, timing constraint satisfaction, race elimination, and logic optimization. The first step constructs a circuit that may contain errors. These are corrected by local modifications in the next two steps. Finally, the sequential logic is simplified by applying transformation heuristics and the combinational logic optimized using standard techniques.

The first part of building the skeletal circuit is to assign a set-reset-dominant (S-R*) latch to each of the output signals of the circuit. An S-R* latch can be implemented by two cross-coupled NOR gates. Using a latch for each output signal makes it easy to generate rising and falling transitions. A rising transition is generated by asserting the set input and a falling transition by asserting the reset input. Furthermore, a latch for each output eliminates the possibility of state-decoding hazards that may occur with more compact state-encoding schemes.

The S-R* latches of the skeletal circuit are interconnected during a breadth-first traversal of the graph that uses the arcs corresponding to the ordering constraints. Once all the incident arcs to a node have been traversed, the node is *implemented* by choosing a circuit template that will generate the transition represented by the node. Super-nodes permit a single template to generate transitions on all the output signals represented in the super-node. Of course, only output nodes need to be implemented. Input nodes are the responsibility of the circuit's environment.

Suture derives its name from the fact that the synthesis method is based on template selection. Input signals and the inputs and outputs of the S-R* latches corresponding to each output signal are stitched together by the templates. These are chosen on the basis of the synchronous properties of the node to be implemented and those of the tail nodes of all the incident ordering arcs.

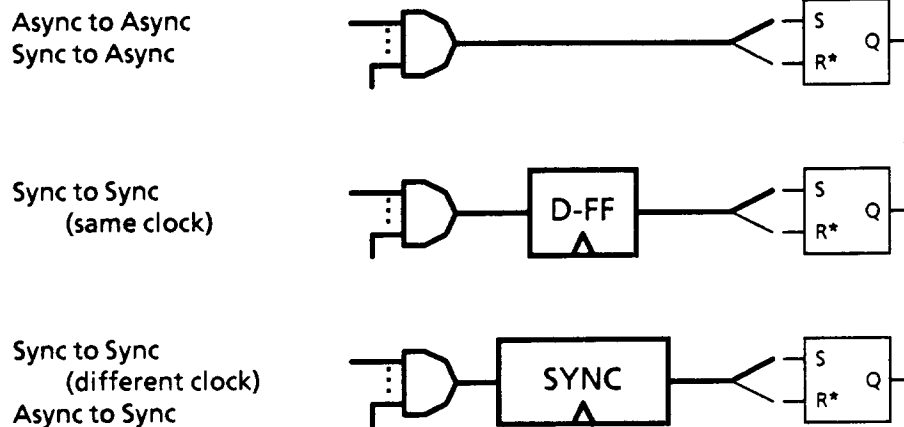


Figure 4.8. Templates used by *Suture* to construct a skeletal circuit. They stitch together input signals and the inputs and outputs of the set-reset latches corresponding to each output signal. There are three basic types corresponding to the synchronous properties of the nodes in the event graph for the circuit. They are connected to the set or reset input of a latch depending on whether the template is used to generate a rising or falling transition on the output signal.

There are only three basic template types (see Figure 4.8): one for generating an asynchronous event (not aligned with a clock edge), one for generating synchronous events from other synchronous events (relative to the same clock signal or phases of the same clock), and one for generating synchronous events from other asynchronous events (synchronous to different clocks or completely asynchronous). The AND gate in each template is used to generate the condition under which the signal transition is to occur. The inputs are collected from the tail nodes of the incident ordering arcs of the node. Whether the signals are complemented or not is determined by the logic level after the transition represented by the tail nodes (i.e., complemented for a falling transition, uncomplemented for a rising transition).

Other inputs to the AND gates are used when conditional behavior is specified. Conditional event sequences are enabled (i.e., by adding an input to the AND gate) when the enabling event of a *Waves* diagram segment occurs. A similar set of enabling conditions is used for looping segments. These are more complex expressions that may include the outputs of a loop iteration counter.

The next synthesis step is to check that the skeletal circuit meets the timing constraint of the event graph. Two types of timing constraints may be violated: minimum or maximum constraints. The circuit can be automatically modified to meet a minimum timing constraint while for maximum timing constraint violations the designer will either to provide a faster circuit library or relax the constraint.

The violation of a minimum constraint means that the circuitry used to generate the sequence of events spanned by the constraint is too fast. Therefore, a delay element must be used to slow

it down. The delay elements may be inverter chains, flip-flops, or delay-lines depending on the available circuit library (the circuit library is discussed in more detail in Appendix C). The delay is added to the corresponding input of the circuit template used to generate the *too early* event (the head of the minimum constraint arc). If the event from which the constraint emanates (the tail of the minimum constraint arc) is not one of the template's inputs then it must be added. This corresponds to an extra ordering constraint. This direct procedure (as opposed to adding delay along an indirect path between the two nodes) guarantees that no maximum timing constraints will be violated in satisfying the minimum constraints.

Adding delay is not common in most synthesis techniques. Typically, operation sequences are scheduled to occur as quickly as possible. However, timing requirements may be imposed on the design when previously designed or off-the-shelf components are involved. This is quite different from the situation where the entire design is under the control of the designer.

Maximum timing constraint violations imply that the circuitry used to generate a transition is too slow. Since Suture synthesizes the fastest possible circuit possible with the available library, there is no automatic modification that can be performed. The designer must be notified of the violated constraint and must either provide a faster circuit library or change the timing and ordering constraints to allow more time for generating the *too late* event. The synthesis procedure must then be repeated for the entire circuit.

The third step in Suture's synthesis procedure is to eliminate any race conditions that may exist as artifacts of the skeletal circuit construction. Corrections for race conditions are the last set of local modifications required before the circuit is fully correct. There are three types of possible race conditions. One is that the set of inputs to the AND gate of a template may not all be true at the required time (i.e., a signal may have already changed before the others reached the logic levels required to trigger the event). Another is that the two events on a signal that is an input to a synchronizing template may be too close together, resulting in neither being noticed on the output of the synchronizer. And lastly, the set or reset conditions of a latch may be true earlier than needed to generate the event (causing it to occur prematurely) or are again true after the event has already occurred (and cause an erroneous reoccurrence).

All three types of races can be detected by a traversal of the graph from each output node that assigns an *interval of occurrence* to each event. The time interval represents the earliest and latest time the event can occur relative to the event of the starting node. Each type of race condition can be detected based on these intervals.

The required correction is similar for the first two race conditions, a S-R* latch is added to *trap and hold* the transitory event and the output of this trapping latch replaces the signal in question as the input to the template. In the first case, the trapping latch records that the event occurred while waiting for the occurrence of the other events with which it is gated. In the second case, the trapping latch extends the duration of the asynchronous event so that it can be synchronized. And lastly, the extra latch holds the reset input of the S-R* latch active until the set input becomes inactive. The third can be corrected by adding inputs (i.e., the state of other signals) to the AND gate of the template that can insure the enabling condition

for the event occurs only at the right time. If no other signals can be used, it may be necessary to use a latch to record that an event occurred and gate the latch's state into the template AND gate. These corrections are described in more detail in section 5.4.3.

Finally, the resulting circuitry is optimized. The templates used by the *Suture* method are meant to provide flexibility in modifying the circuit to correct constraint violations and race conditions. However, the circuit that is finally constructed may be less efficient than necessary. Two types of size reductions are possible: sequential logic transformations and combinational logic optimizations. Sequential transformation is performed first. Stated simply, it is the transformation of the set of latches, flip-flops, synchronizers, and gates that correspond to each output signal into a smaller set. This can be done by exploiting sequential logic equivalence or replacing sequential logic with combinational logic if certain conditions are met (see Figure 4.9). Multi-level combinational logic optimization can be used on the resulting circuit to combine common sub-circuits [Brayton87].

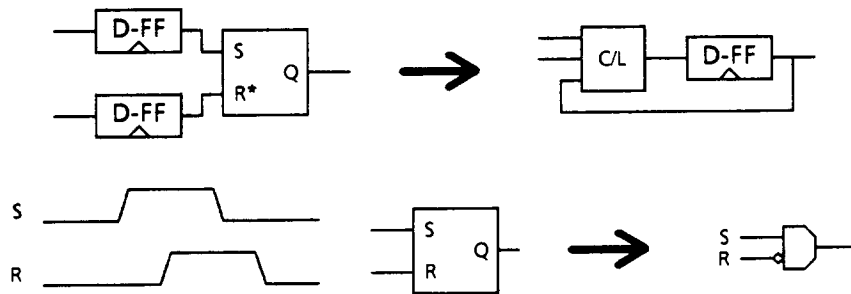


Figure 4.9. Two examples of sequential logic transformations. The first combines two flip-flops and a latch into a single flip-flop and combinational logic. the second replaces a latch with a simple AND gate that has the same output behavior as a reset-dominant latch under the specified input behavior.

Three examples can be used to demonstrate the interesting features of *Suture*. The first is a fully synchronous three-bit counter. The second is a fully asynchronous FIFO stack control cell (from section 4.2.2). The last is a mixed synchronous-asynchronous interface transducer (from section 4.2.1).

The specification for the input/output behavior of a three-bit counter is shown in Figure 4.10. The *Waves* diagram includes the shape of the output waveforms and the timing relations that must be maintained between logic transitions. Furthermore, the first event on Bit0 is annotated to only be generated if the condition (AND (NOT Bit0) (NOT Bit1) (NOT Bit2)) is true. This establishes a looping three-bit counter.

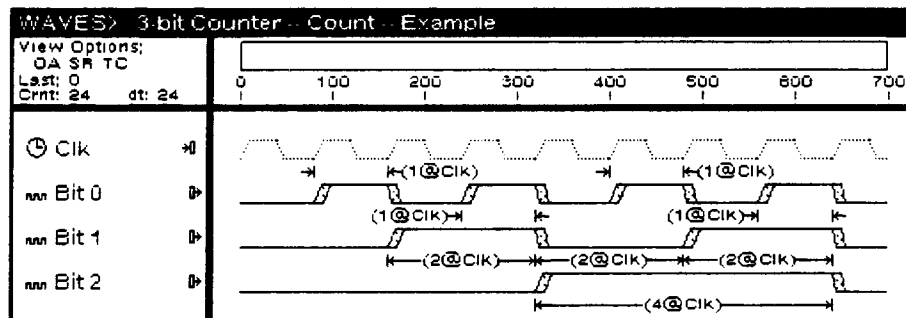


Figure 4.10. *Waves* specification for the input/output behavior of a three-bit counter. Seven timing constraints are used to specify that each state of the counter must last for one cycle of the clock. Simultaneity constraints exist between all events drawn as occurring at the same time. The first event on Bit0 is annotated to only be generated if the condition (AND (NOT Bit0) (NOT Bit1) (NOT Bit2)) is true. The condition is not visible in the diagram but can be accessed by the user through a mouse operation on the event. It is used to establish a looping three-bit counter.

Of course, if it is known in advance that a counter will be required then a great deal of effort can be spared by instructing the algorithm to implement it directly. However, for explanatory purposes a counter is a good example with a familiar logic implementation for comparison.

Suture's first step is to construct an event graph from the *Waves* diagram (see Figure 4.11). The transformation is straightforward. A node is generated for each event in the diagram. Some events are collected into super-nodes by simultaneity constraints. The solid arcs in the graph correspond to ordering constraints (in this case, they are all implicit ordering constraints). The dashed arcs correspond to the timing constraints visible in the diagram. A *start marker* is associated with the first event and an *end marker* with the last group of simultaneous events.

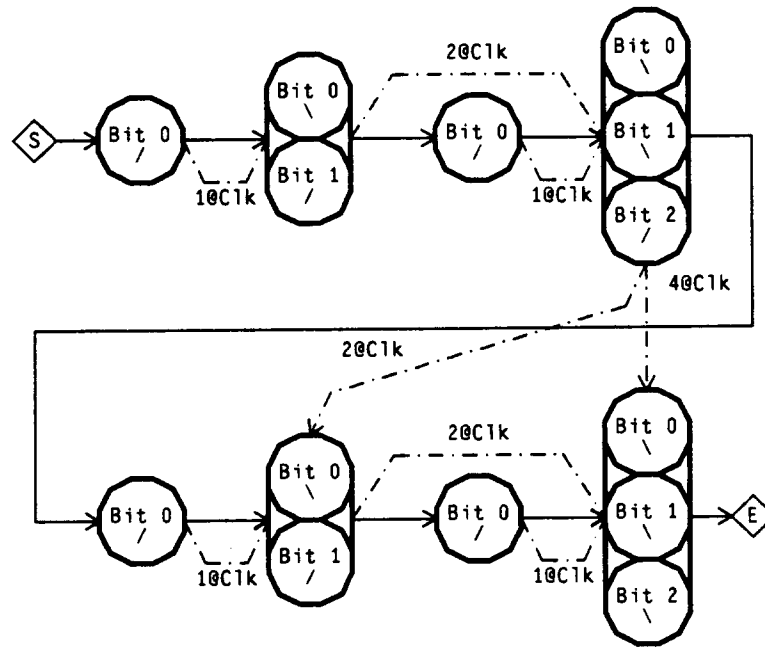


Figure 4.11. Event graph derived from the Waves diagram of Figure 4.10. Nodes in graph correspond to events on the waveforms. Nodes are grouped into super-nodes when simultaneity constraints exist between their respective events. Solid arcs represent ordering constraints. In this case, they are all implicit ordering constraints. No explicit ordering constraints are present in the diagram. Dashed arcs represent the timing constraints visible in the diagram. Start and end markers are associated with the first and last event groups.

The circuit is constructed by traversing the graph and selecting the appropriate templates (see Figure 4.8). In this case, with a fully synchronous specification, all the templates are the same, consisting of a flip-flop and AND gate (see the top half of Figure 4.12).

The inputs to each AND gate correspond to the incident ordering arcs of the node being implemented. Some extra inputs are added to some of the gates to distinguish multiple rising or falling transitions on two of the signals and to eliminate some race conditions. Only the third type of race condition exists in this graph. A set or reset condition that is true earlier or later in the event sequence than it should and causing events to occur in the wrong order. All of the occurrences of this race condition are corrected by adding some extra inputs to the template AND gates.

The circuit is quite large and contains many flip-flops. However, after some sequential logic transformations are applied to the circuitry for each of the output signals, the more familiar and much smaller three-bit counter implementation is obtained (see the bottom half of Figure 4.12). Again, the *Suture* method is intended to provide a framework for the synthesis of circuits with timing constraints. The circuit initially generated provides such a structure.

Any state of the counter can be extended in time by adding delay to the output signal of the appropriate template. Of course, this may limit the extent of optimization that is achievable. In this case, with no timing constraint violations in the skeletal circuit, no delay elements are required and the circuit can be transformed to the optimal implementation.

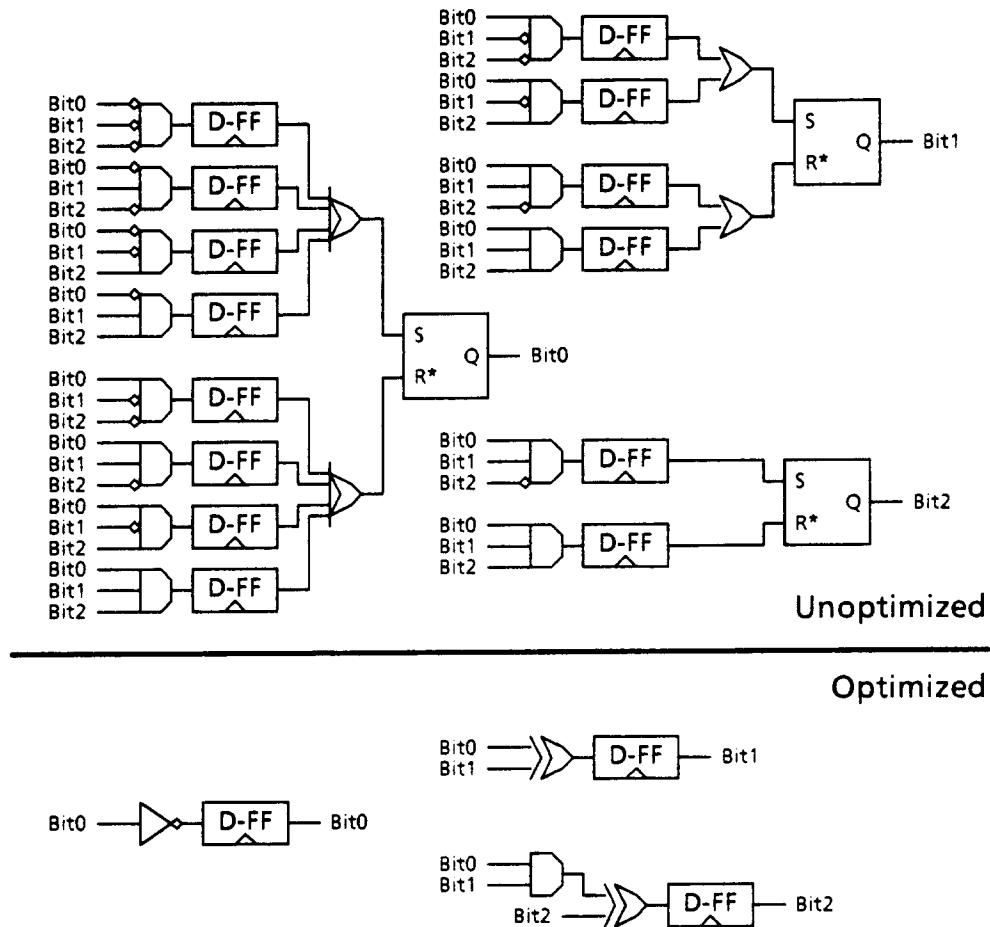


Figure 4.12. Circuit synthesized from the event graph of Figure 4.11. Before any optimizations are performed, the circuit is as shown above the horizontal line. All flip-flop control inputs are connected to the CLK signal. The more familiar three-bit counter implementation is obtained after sequential logic transformations are applied.

These same properties of the method are demonstrated again within the context of a fully asynchronous example. The specification of Figure 4.13 is for the FIFO stack control cell of section 4.2.2. As expected for a self-timed subsystem, only ordering constraints are entered in the specification. Its corresponding event graph is shown in Figure 4.14.

The circuit constructed from the event graph is shown in Figure 4.15. Unlike the previous example, the unoptimized circuit is not especially complex. However, after some simple transformations, similar to those of Figure 4.9. (e.g., an S-R* latch with both inputs gated by

the same signal becomes a D latch with input $(Q + S)R'$, where Q is the latch output) the circuit obtained is identical to that of Figure 4.7 which was derived from the Petri net specification of Figure 4.6.

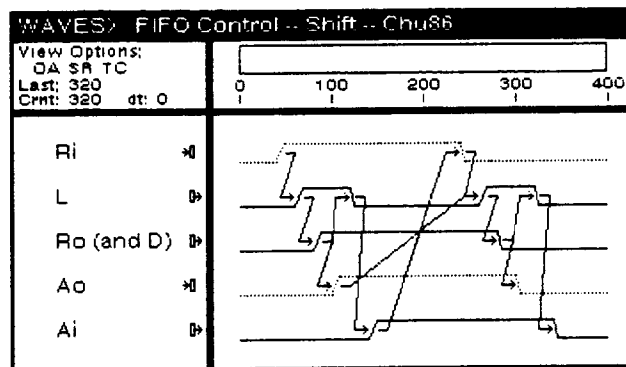


Figure 4.13. Waves specification of a FIFO stack control cell [Chu86a]. There are only ordering constraints in this specification of an asynchronous control cell for a self-timed FIFO stack.

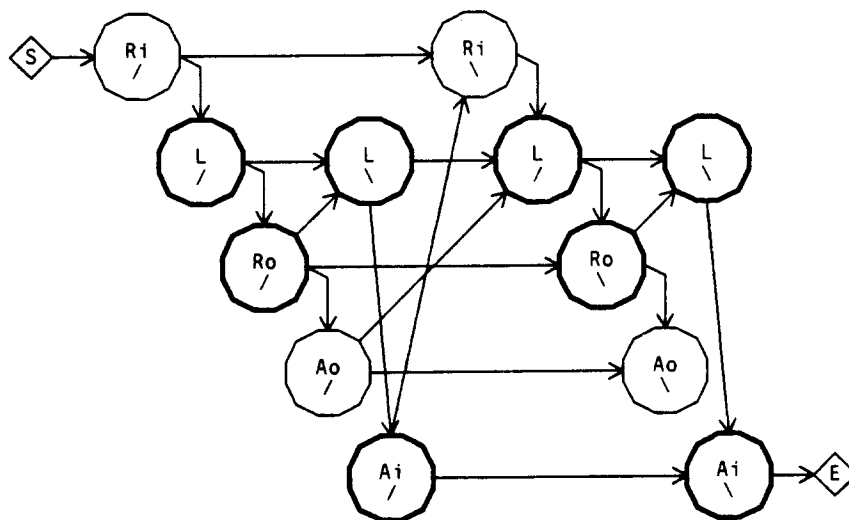


Figure 4.14. Event graph for the specification of Figure 4.13. The graph arcs are all ordering constraints and include no timing constraints. This is typical for a component of an asynchronous self-timed system.

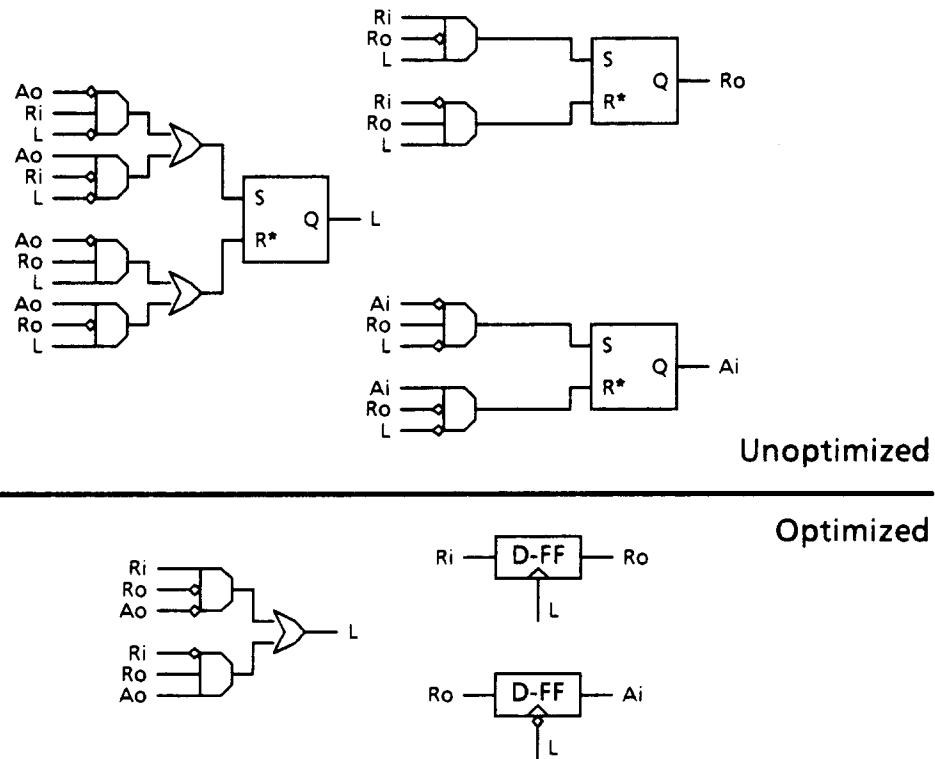


Figure 4.15. Circuit synthesized from the event graph of Figure 4.14. After sequential logic transformations it is identical to that of Figure 4.7.

The example again demonstrates that although *Suture* first constructs a more complex circuit, after optimization it is very similar – in this case, identical – to the circuit constructed by the less general methods of section 4.2.2.

The next and last example is a circuit with both synchronous and asynchronous signals as well as timing constraints that will require attention. The example is taken from section 4.2.2 and is a simple interface transducer between the Intel Multibus and a synchronous interface. Only one operation is supported by the transducer, a data read to a memory that is connected to the bus. The specification for the transducer consists of two *Waves* diagrams, one for each interface of the transducer (see Figure 4.16).

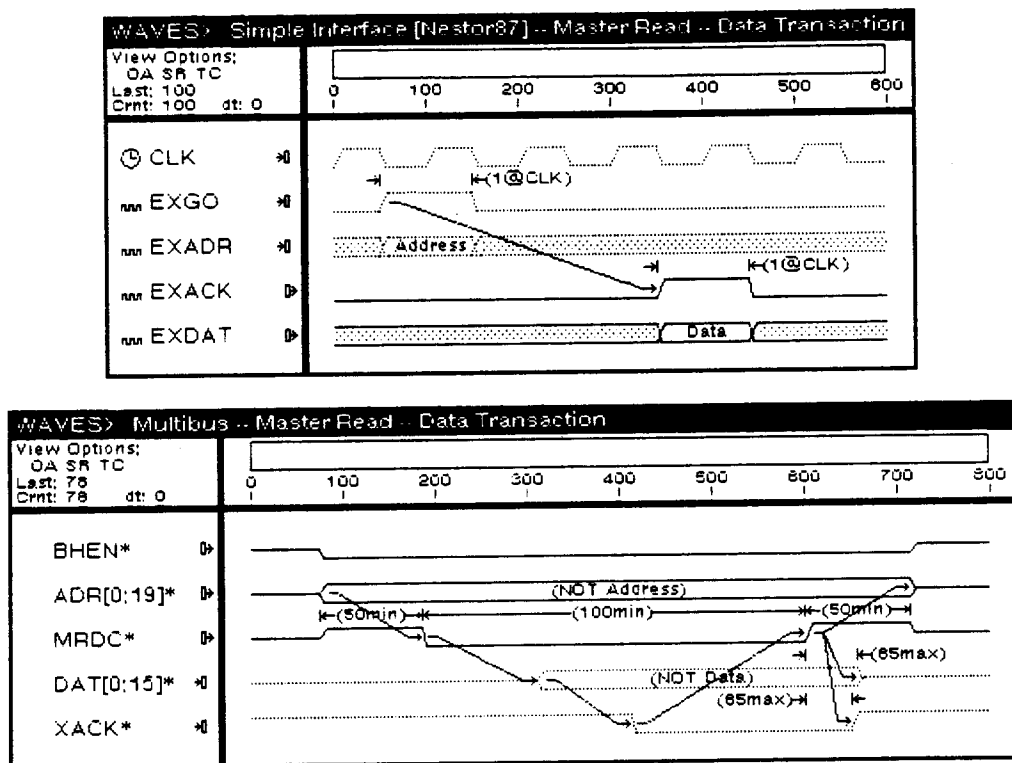


Figure 4.16. Waves specification of the interface transducer of section 4.2.1 [Nestor87]. This should be compared for readability and completeness with the BSI/ISPS specification of Figure 4.3.

The Waves diagrams for this example have characteristics not present in the previous two examples. Some signals are bundles of wires and are used to represent events on the entire group of signals rather than just one at a time. Furthermore, these signals carry *data*, represented in the specification by the labels along the waveforms. And lastly, the waveforms include signals that are tri-stated and must be separated into two: one to carry the logic level and one to enable or tri-state the logic signal.

Data transfer is one of the primary functions of an interface transducer. In fact, by analyzing the data dependencies between the two interfaces the two unconnected event graphs derived from the two diagrams can be interconnected. The combined graph is shown in Figure 4.17. The darker arcs represent those added to the graphs by the data dependencies. Three minimum timing constraint arcs are also added to ensure the proper setup time at latch inputs. Two of the three are parallel to the data dependency arcs and the third ensures that MRDC* will not be raised before the data has been latched.

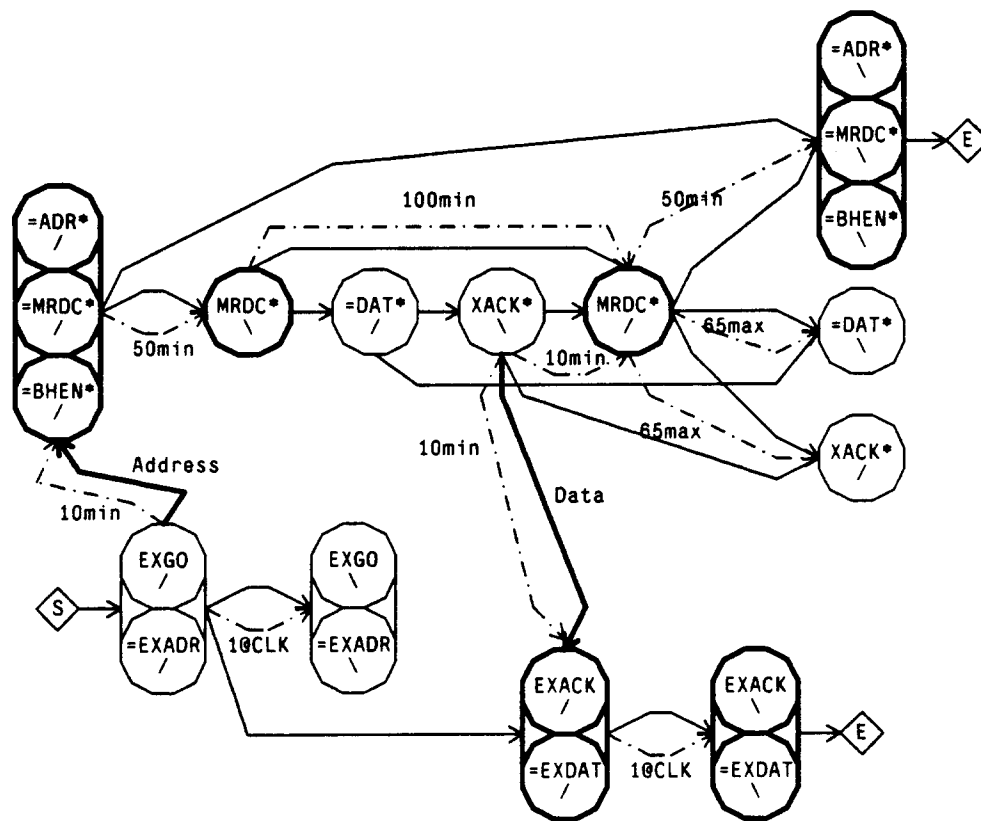


Figure 4.17. Event graph for the specification of Figure 4.16. Two ordering arcs interconnect the graphs of each *Waves* diagram (boldface arcs) based on data dependencies across the interfaces. In this case, Address must be transferred in one direction and Data in the other. Start and end markers are specified by the user. The signals with the = prefix are the tri-state control for the signals with the same name.

The logic design for the transducer is shown in Figure 4.18. The interesting features include two 50ns delay elements required to meet the address setup and hold requirements of the Multibus. They delay the assertion of the command and tri-stating of the Address lines. The circuit also contains latches for the data path and two 10ns delay elements are used to ensure setup time on their inputs. The control signal for the latches is determined by the same method as that used to generate the data dependency arcs for the graph. Note that the third 10ns minimum timing constraint is not represented in the circuit because the logic circuit used to generate the event on MRDC* has an equal or greater delay. Three extra S-R* latches (top right of Figure 4.18) are used to correct race conditions. They record the occurrence of certain events and are used to make sure that events occur in the proper order. Lastly, an extra signal is generated to control the quiescent value of the output signals and reset the race correcting latches. The OPDisable is set and reset by the events associated with the end and start markers.

It is interesting to note that although the circuit of Figure 4.18 is about the same number of gates as that of Figure 4.5 it has a larger communication bandwidth. *Suture's* ability to synthesize asynchronous components where required gives it this advantage. For example, the setup and hold times for the ADR lines on the Multibus are equal to the required 50ns while for the realization of section 4.2.1 they are one clock cycle. This extra time is the overhead incurred by performing the data transaction synchronously.

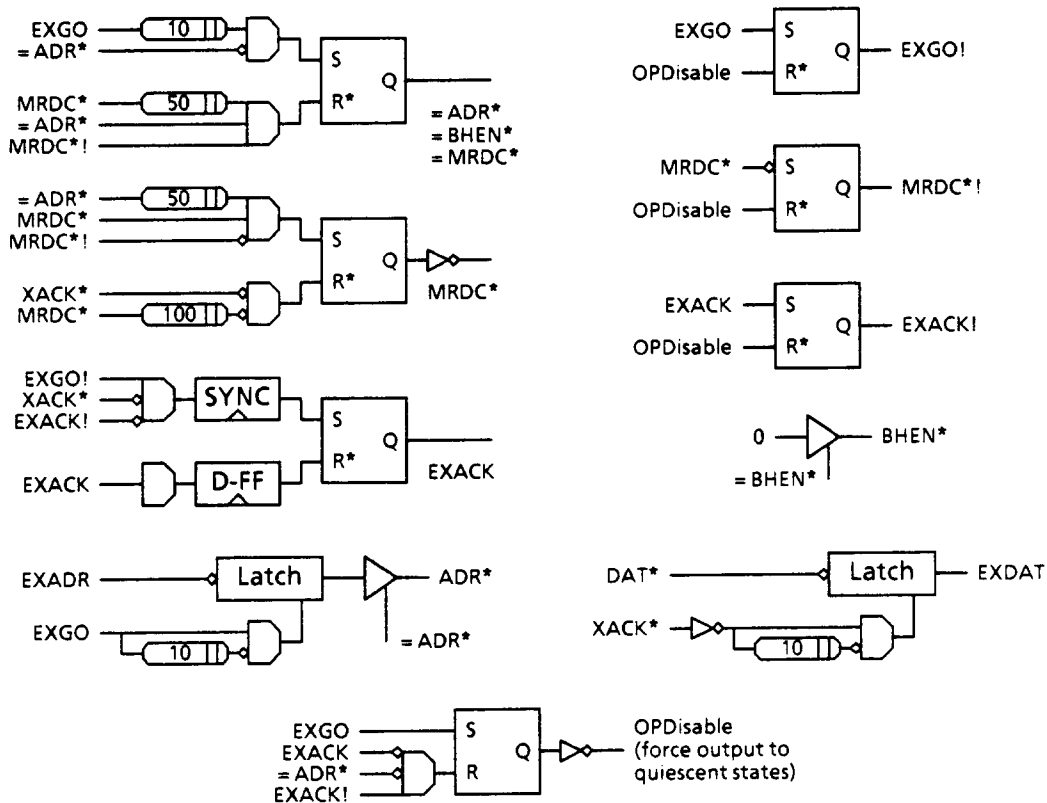


Figure 4.18. Circuit synthesized from the event graph of Figure 4.17. A special signal is generated to keep the wires in their quiescent states (OPDisable) and is a reset input to all the output latches. It is also used to reset trapping latches used to correct for race conditions (three of which are in the top right portion of the figure). Five delay elements (two 10ns, two 50ns, and one 100ns) are added to the circuit to enforce timing constraints corresponding to latch and Multibus setup and hold times and minimum command assertion time. Three 10ns delay elements are required to meet data latch setup times. All the flip-flops in the circuit are clocked by the CLK signal. Only one signal's circuitry (EXACK) can be optimized by sequential logic transformation (see Figure 4.9), its optimized form is not shown. The transformation is possible because a synchronizer behaves as a D-type flip-flop to signals already synchronous to the clock.

These examples have provided an overview of the *Suture* algorithm. In the next chapter, I will review each of the steps in more detail and discuss how the method is used to synthesize interface transducers. Some aspects of interest include: generation of the event graphs from the collections of *Waves* diagrams that describe the transducer, optimization of the graphs by exploiting don't care conditions and the compressability of synchronous events, and generation of the transducer's data path. After *Suture* is applied to the event graphs of each interface operation (generating an independent circuit for each), the resulting circuits are combined into a single transducer prior to the sequential and combinational logic optimizations discussed above. The tool that integrates *Suture* with these pre- and post-processing steps is called *Janus* and is the subject of Chapter 5 (Appendix C provides additional implementation details).

<This page intentionally left blank.>

Janus is a tool for the automatic synthesis of interface transducer logic. It uses *Waves* diagrams as input and generates a logic specification of the transducer. *Janus* begins by building event graphs for each of the operations supported by the transducer. It then uses the *Suture* algorithm of Chapter 4 as the backbone of its synthesis procedure. Extensions include the synthesis of data-path elements and constructs for handling conditionals and loops. *Janus*, like *Waves*, is implemented in Interlisp-D/LOOPS.

This chapter is composed of six sections. The first section defines the class of transducers which *Janus* can synthesize and the type of behavior specification required as input. The second section describes the generation of event graphs from the *Waves* diagram input and their manipulation and annotation prior to synthesis. The third and fourth sections detail the methods used by *Janus* for data path and control synthesis. The fifth section describes the extensions used to handle conditional and looping event specifications. The last section covers the logic optimizations, both sequential and combinational, that are used to reduce the size of the final design.

Janus is the Roman god of beginnings and endings. He is often depicted with two faces looking in opposite directions. Busts of Janus adorned the gates of Roman cities, looking both in and out from the city wall. His name is appropriate for the synthesis tool that is the subject of this chapter, a tool for the automatic logic design of interface transducers.

Janus begins with a specification of the input/output behavior of the transducer's two interfaces and generates the logic specification to be used in realizing the transducer circuitry. A Y-chart showing the transformation performed by *Janus* is shown in Figure 5.1.

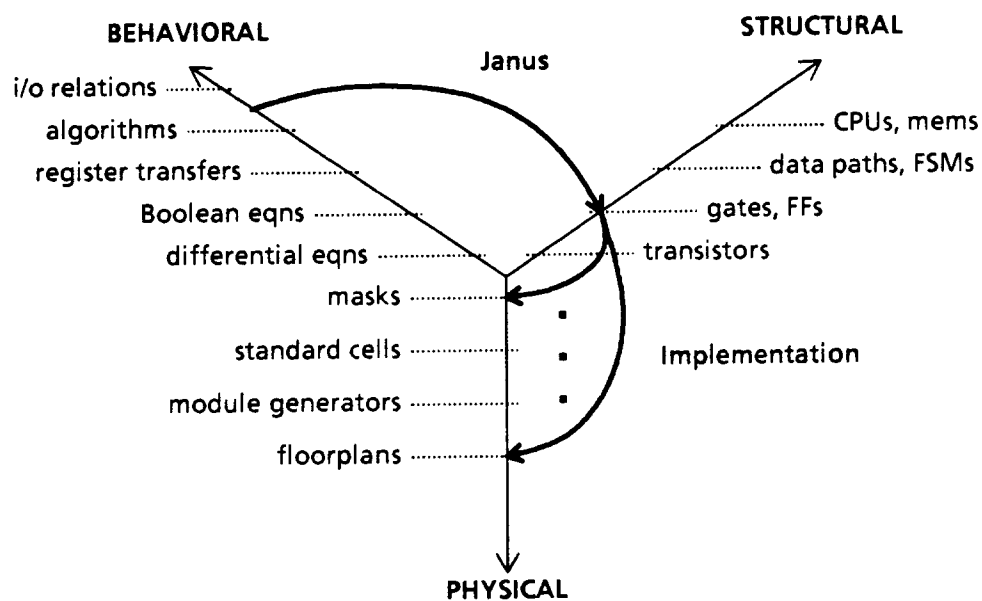


Figure 5.1. A Y-chart for *Janus*. *Janus* transforms a behavioral specification of the transducer (at the level of input/output behavior) into a structural description (at the level of gates and flip-flops). The mapping to a particular implementation technology is left to other tools [Sangiovanni86, Leive81].

The transformation begins with a specification in the behavioral domain. The input/output specifications used by *Janus* are placed above algorithmic specifications because they do not imply that the circuit implements a specific algorithm. The transformation ends in the structural domain with a collection of flip-flops, latches, and logic gates. The mapping of the structural specification to a specific technology, and then to an actual physical realization, is left to implementation tools and is outside the scope of this dissertation [Sangiovanni86, Leive81].

Janus uses the *Suture* algorithm described in the previous chapter as its backbone and extends it in two directions. First, it generates the required event graphs from a collection of *Waves* diagrams that describes the behavior of the interfaces being connected. The graphs are modified so as to consist of only rising and falling events before they are used by the *Suture* algorithm to generate the transducer control circuitry (see section 5.2). Second, *Janus* performs limited data path synthesis (see section 5.3). Transducer data paths consist of the latches and multiplexers required to transfer data through the transducer.

Janus, like *Waves*, is implemented in Interlisp-D/LOOPS [Bobrow83, Stefik86, Xerox86]. Since it uses data structures derived from *Waves* timing diagrams, it is simpler to implement it in the same single address space programming environment. *Janus*' implementation details and a description of its data structures can be found in Appendix C.

Specification Using Waves Diagrams

5.1.1

As discussed in section 4.1.2, an interface transducer specification consists of the two interfaces to be connected and the operations that the transducer must support. The *Janus* icon acts as a catalog of interface transducers currently loaded into virtual memory and also provides access to transducers already created and stored in files (see Figure 5.2).

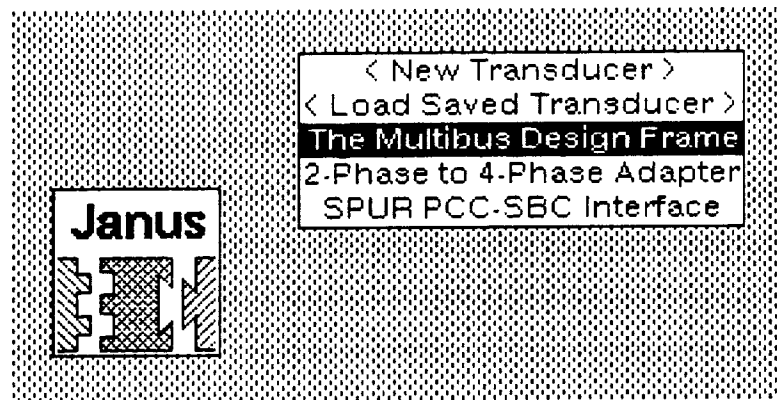


Figure 5.2. The *Janus* icon and its menu, a catalogue of all the transducers loaded into virtual memory. By clicking the mouse button over the icon the user can access all currently loaded transducers. In the case above, three transducers are loaded and the user has selected *The Multibus Design Frame*.

A transducer is symbolized on the screen by an image like that of Figure 5.3. The *interlocking blocks* are used to specify the name of the transducer, the interfaces it connects, and the operations it supports. These elements are visible in the icon except for the list of interface operations. The operations are accessed through the *transducer menu*, available from the center block of the icon.



Figure 5.3. A *Janus* interface transducer icon. The name of this transducer is *The Multibus Design Frame* and it connects the interfaces called *Multibus Design Frame* and *Multibus*. The list of operations that the transducer supports can be obtained from a pop-up menu obtained by clicking the mouse over the center block of the diagram that symbolizes the transducer connecting the two interfaces.

Once the transducer has been specified, the user can invoke *Janus* to synthesize the circuit through the transducer menu. Besides the specification of the operations to be supported and the synthesis of the transducer logic, other menu options permit the user to have only some intermediate synthesis steps performed. For example, the user may request that *Janus* check the specification diagrams and generate a list of errors and warnings if the specification is inconsistent or includes features that the synthesis process cannot handle (see section 5.1.2 and Appendix C). The user can also request that only the translation from *Waves* data structures to *Janus* data structures (i.e., diagrams into event graphs) be performed. This feature is useful in separating a transducer from its specification and thus no longer requiring the *Waves* diagrams to be simultaneously residing in virtual memory.

Limitations of the Implementation

5.1.2

Janus has three significant limitations: (1) it only supports atomic non-overlapping operations, (2) data transfers are of constant width, and (3) it provides only limited error reporting.

The first limitation concerns the basis of *Janus*' model of interfaces, namely, the concept of indivisible operations (see section 4.1). *Janus* is restricted to those interfaces where operations do not overlap and only one operation is active at one time. Most commercial busses, even packet-switched busses, can be described under these restrictions by using internal interface state to hold the information that must extend from one atomic operation to another. By treating each operation separately, *Janus* can synthesize the circuitry for each operation independently and then merge them together. To handle overlapping operations it would be necessary to describe constraints between events of different operations and take these into account during synthesis. Although these constraints can be expressed in *Waves* diagrams, if in a somewhat cumbersome manner (by means of the merge labels described in section 3.3.3), *Janus* does not currently support this capability in its synthesis algorithms.

The second limitation is that data transfers through the transducer must be of the same width. Multiplexers and shift registers may be required to handle different width transfers. These capabilities can be easily incorporated into *Janus* by generating multiplexer control signals

and events as for the case of multiplexed data paths (see section 5.3.2). However, this capability has not yet been implemented.

The last limitation, is that error reporting in *Janus* generates a textual description of the error rather than directly highlighting the offending objects in their *Waves* diagram. This can be remedied when *Waves* and *Janus* are reimplemented on top of a true design data base rather than just an object-oriented environment. Persistent objects, that is, data structures with unique identifiers, can be retrieved from the data base as required rather than being restricted to simultaneously residing in virtual memory.

Removing these limitations does not conflict with the basic approach of *Janus*. They are still present only because they do not have a large impact on the class of transducers that can be synthesized and the effort required is not easily justified for this initial version of *Janus*.

The remainder of this chapter deals with *Janus*' capabilities. It is broken into sections that correspond roughly to the stages in *Janus*' synthesis process beginning with the generation of event graphs from *Waves* diagrams to the optimization of the synthesized sequential logic.

The event graphs used as input to *Janus* are derived directly from *Waves* timing diagrams. Two distinct graphs, one for each side of the transducer, are constructed for every interface operation supported by the transducer. The nodes of the graph have a one-to-one correspondence with events in the diagram (i.e., the transition on signal waveforms). The arcs of the graph are of two types corresponding to the ordering and min/max timing constraints existing between the events.

Janus begins its translation by creating data structures for each of the signals that takes part in the interface operation (i.e., undergoes some logic transitions). These data structures contain properties of the signals such as direction and electrical parameters (e.g., open-collector). Periodic and aperiodic signals are separated — only the circuitry for the aperiodic signals will be synthesized with the *Suture* algorithm. Synchronicity constraints are kept with the aperiodic signal data structure and are not represented in the graph itself.

Data structures are also created for each of the segments of the *Waves* timing diagram. These data structures will be used to construct control logic for the generation of conditional and looping events. Section 5.5 describes the details of the framework for applying the *Suture* algorithm in the presence of these conditional and looping constructs.

Generating the graph structure is straightforward. Each event in the diagram is translated into a node in the graph. Pointers are maintained to the event's signal and the diagram segment during which the event occurs. The remaining three types of timing constraints either add arcs to the graph or group nodes into super-nodes. Again, the fourth type of timing constraint, the synchronicity constraint, is not represented in the graph.

Min/max timing constraints are the simplest to handle. The constraint creates a timing constraint arc from the earlier event to the later event. Ordering arcs are similarly created from the explicit ordering constraints of the diagram. However, ordering arcs must also be created for the implicit ordering constraints among events on the same signal wire. Simultaneity constraints, rather than being translated into arcs, cause their nodes to be grouped into a super-node. The same logic will be used to generate all the events grouped into a super-node and must ensure that they all occur within the maximum time specified by the constraint.

This procedure creates a graph for each of the diagrams associated with a transducer specification. The new data structures derived from the *Waves* diagram include all the information required by the synthesis algorithms. What is lost are the details of the graphical presentation of the events in the form of a timing diagram.

In many cases, more than one diagram is needed for the specification of an interface operation. Diagram combination is achieved through the use of merge labels attached to events in the diagram (see section 3.3.3). This mechanism permits the arbitrary combination of diagrams according to a co-routine model of parallelism and also allows the expression of constraints that cross diagram boundaries.

In translating a collection of diagrams into a single graph, each diagram is first translated separately. Once this is done, all the nodes in the entire collection of diagrams that have attached merge point labels are collected. The nodes (or super-nodes) with identical merge point labels are grouped into super-nodes. After merging, any duplicate nodes within a super-node are destroyed (two nodes are duplicates if they represent identical transitions on the same signal). Nodes with ordering point labels cause an ordering arc to be added to the graph. The arc is directed from the node with the identically labelled merge point to the node with the ordering point label. The restrictions on the ways labeled points can be merged or ordered are described in Appendix C.

Some diagram events are only place holders for constraints (see section 3.3.3). A constraint between events in two diagrams may be expressed by duplicating one of the events in the other diagram, attaching the constraint to it, and then marking it as not being a part of the event sequence depicted in its diagram (i.e., *inactive*) and labeling it with a merge label identical to that of the corresponding event in the other diagram. When the two diagrams are combined, the place holder node is merged with the original in a super-node to which all the constraints are transferred. The place holder nodes and any unmerged inactive nodes can then be removed from the data structure.

After merging, the structure of the graphs may have been drastically altered. The specifier of the interface must be very careful in the placement of these labels. One can imagine that incorrectly placed merge points can easily result in inconsistent event sequences or cyclic graphs. A more probable consequence is that some timing constraints cannot be satisfied after merging two event sequences. Within a diagram, the *Waves* editor's interactive constraint checking capabilities (see section A.4) ensure that a consistent placement of events is possible (i.e., a diagram drawn without constraint violations). However, when different diagrams are merged this property no longer holds.

Intervals of Occurrence

5.2.2

To verify that a graph has a consistent set of timing constraints, it is necessary to obtain an interval of occurrence for each node of the graph. An *interval of occurrence* is a time period during which the event represented by the node may occur relative to a fixed node (i.e., one with a fixed zero-width interval) (see Figure 5.4). When determining whether the constraints

are consistent the algorithm is run with the start node of the operation as the fixed node. If any node has an empty interval then an inconsistent set of constraints exists. The events corresponding to these nodes indicate where corrections are to be made in the specification.

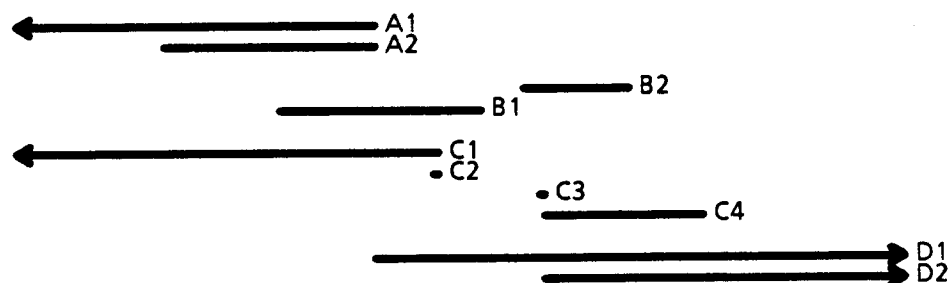


Figure 5.4. An example of intervals of occurrence. Each node is represented by a line segment proportional in size to its interval of occurrence. Nodes with the same letter occur on the same signal wire. In this example, the fixed node is C2. The events represented by nodes A1 and A2 will definitely occur before the event of C2. B2, C3, C4 and D2 will definitely occur later. Only C3 is precisely positioned in time relative to C2 (i.e., it has a zero-width interval of occurrence). Nothing can be said about the order of occurrence of B1 and D1 relative to C2. The C nodes are ordered in time by the fact that they occur on the same signal, however, the interval of occurrence algorithm merely states that C1 will occur before or simultaneously with C2 and that C4 will occur after or simultaneously with C3. A1 and C1 may occur an arbitrary time before C2. D1 and D2 may occur any time after.

The algorithm used for determining intervals of occurrence is similar to those found in compaction and spacing tools for integrated circuit layout [Burns86]. The first step is to detect any cycles that may exist in the graph. *Waves* diagrams guarantee acyclic graphs; cyclic behavior is represented via regular expressions on diagram segments (see section 3.3.2). However, after merging two diagrams this need no longer be the case. The presence of a cycle indicates a malformed merge that is inconsistent with the specification methodology and the specification must be corrected before proceeding further.

Once an acyclic graph is obtained, intervals of occurrence can be computed. The algorithm begins with one node being fixed and assigned a zero width interval of $<0,0>$ corresponding to identical values for the earliest and latest possible times of occurrence. Every other node is assigned the interval $<NIL, NIL>$ corresponding to a lower bound of negative infinity and an upper bound of positive infinity. A queue is used to hold all the nodes whose intervals have been updated. The queue begins with only the starting node and as nodes are added, they are sorted by earliest time of occurrence. The algorithm proceeds by removing a node from the queue and traversing all its fan-in and fan-out arcs. The intervals of the neighboring nodes are updated based on the arc type and its minimum or maximum values, if any (see Table 5.1). If the interval of a neighboring node actually changes then it is added to the queue. This process continues until the queue has been emptied.

It is straightforward to determine the complexity of this algorithm. Each edge in the graph is traversed at least once, yielding a lower bound of $O(E)$ where E is the number of edges. However, when a maximum constraint is encountered, the upper bound of an interval may change. This change may propagate through the entire graph, causing each edge to be traversed again. Therefore, the complexity of the algorithm is $O(E*N_{max})$, where N_{max} is the number of nodes with incident maximum timing constraints. Typically, there are few maximum constraints. When they do occur, they also tend to be local in their effect (i.e., they span nodes that are near each other in time). Therefore, the average running time of the algorithm is closer to $O(E)$ than $O(E*N_{max})$.

	arc from N_0	arc to N_0
ordering	$\langle \text{earliest}_0, \text{NIL} \rangle$	$\langle \text{NIL}, \text{latest}_0 \rangle$
minimum	$\langle \text{earliest}_0 + \text{min}, \text{NIL} \rangle$	$\langle \text{NIL}, \text{latest}_0 - \text{min} \rangle$
maximum	$\langle \text{NIL}, \text{latest}_0 + \text{max} \rangle$	$\langle \text{earliest}_0 - \text{max}, \text{NIL} \rangle$

$$\langle \text{earliest}_i, \text{latest}_i \rangle \leftarrow \langle \max(\text{earliest}_i, \text{earliest}), \min(\text{latest}_i, \text{latest}) \rangle$$

Table 5.1. The time interval update for neighboring nodes N_i is computed from node N_0 by using the table above. $\langle \text{earliest}_0, \text{latest}_0 \rangle$ and $\langle \text{earliest}_i, \text{latest}_i \rangle$ are the intervals for the nodes before an update. A new interval for node N_i is determined by the type and direction of the arcs connecting it to N_0 . Its time interval is updated by taking the maximum of the previous and new earliest value and the minimum of the previous and new latest value.

Intervals of occurrence will be updated many times during the synthesis process. After being used to check timing constraint consistency, this algorithm will be used to interconnect the graphs for the two sides of the transducer (see section 5.2.3), to translate timing constraints (see section 5.2.4), and to determine the possible logic levels of signals at different points in time (see section 5.4.5).

Interconnection of Operation Graphs

5.2.3

The two graphs for each operation supported by the transducer must be interconnected by arcs that span both graphs. The first arc to be added is simply used to start an event sequence on one side when the corresponding sequence has started on the other side of the transducer. This is achieved by adding an arc between the start nodes of the two graphs. The direction of the arc is determined by the direction of the events represented in the start nodes. The arc will

point from an input node to an output node, that is, once the start of an operation is signalled by an input event then the operation on the other side can start with an output event from the transducer. Operations with two input start nodes or two output start nodes are clearly incompatible.

The other ordering arcs added to the graph are due to data dependencies. Data to be transferred through the transducer must first be available as an input before it can be an output of the transducer. The situation is identical for input data that is to be latched into internal transducer state elements. Intervals of occurrence are used to determine the exact placement of these arcs. An event must be identified that indicates when input data is valid. This can be an event that occurs during the time that the data is valid or a precise time before.

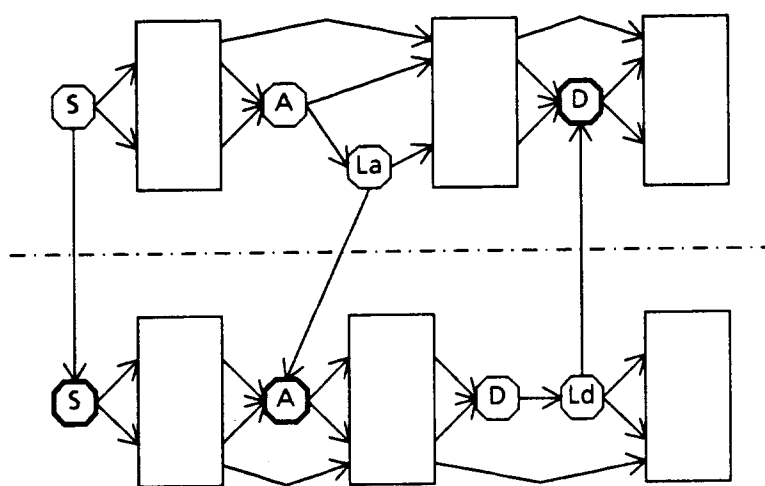


Figure 5.5. The graphs for the two sides of the transducer are interconnected by adding ordering arcs from the input start node of one graph to the output start node of the other and from latching nodes of data inputs on one side to data outputs on the other. In this figure, three ordering arcs cross from one graph to the other, two of them are data dependency arcs (the data are labelled A and D, their latching nodes are La and Ld) and one is a start node arc (start nodes are labelled S). Dark and light outlined nodes represent output and input events, respectively.

The interval of occurrence algorithm is run once for each piece of input data that must be transferred and begins with the node corresponding to where the data first becomes valid as the fixed node with a $\langle 0,0 \rangle$ interval. After the algorithm has completed, the data signalling nodes can be identified. If the node occurs after the data is asserted, then it can be used to latch the data when it becomes valid. This *latching node* will become the tail of an ordering arc whose head points to the output node on the other graph where the data must be output by the transducer. A minimum timing constraints arc whose value is equal to the latch setup time is added in parallel to the ordering arc. If a node that occurs before the data becomes valid can be used, then the minimum timing constraint arc must reflect the extra time that the signal must be delayed before it can be used to latch the data. This prevents the transducer

from attempting to output the data until after it has become valid on its input. A minimum timing constraint may also be added to ensure that the data will be available on the input long enough to be latched. The arc extends from the event used to latch the data to an output event that can delay deassertion of the data. Currently, *Janus* is restricted to latching events that are in the same segment as the data event, the most common case in interface specifications.

Adding these extra arcs to the graph can, as in the case of merge points, cause an inconsistency in the timing constraints. An example of an inconsistency is when an interface requires two pieces of data to be produced by the transducer in a specific amount of time but the other interface supplying the data requires a longer period of time to provide them. At this point in the preparation of the event graph, the interval of occurrence algorithm must be run again to verify that the data transfers across the transducer are consistent.

Problems of this type can sometimes be corrected by adding extra constraints to delay one operation until all its outputs can be determined before they are needed. In the example above, the operation that requires the outputs from the transducer cannot be started until after the other interface has provided both data items. This adjustment is not always possible, however. Some interfaces are simply incompatible and this information must be fed back to the user along with the offending data and constraints. When it is possible to adjust the graph it is done by performing the timing constraint translations outlined in the next section.

Timing Constraint Translations

5.2.4

Suture uses a greedy strategy to generate output events. Events are generated as quickly as possible. If any minimum timing constraints need to be satisfied then a delay is added to the path that generates the event. A problem arises when a configuration of nodes and arcs such as that of Figure 5.6 exists in the graph. The problem is that the circuitry *Suture* designs will generate event B immediately after event A. Then there will be no way to satisfy the constraint between B and C. To rectify this problem, an extra minimum timing constraint arc must be added between nodes A and B whose value is equal to the difference in the lower bounds of their time intervals. In the case of Figure 5.6, this is a 200 minimum constraint.

These cases can be found by determining time intervals for the nodes while ignoring maximum timing constraints. A scan of all the maximum timing constraints will identify those that would alter the lower bound of the time interval of their tail nodes. In Figure 5.6 this is the 100 maximum constraint between nodes B and C. The tail nodes of these constraints (e.g., node B) are the ones for which the *Suture* algorithm is inadequate. A minimum constraint arc is added from the node (node A) that sets the lower bound of the head node of the maximum constraint arc (node C) to the tail node of the arc (node B).

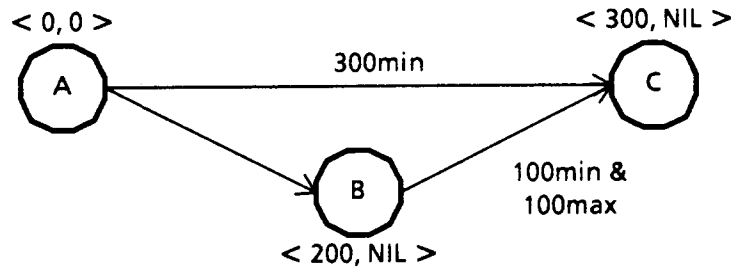


Figure 5.6. A set of nodes that requires a timing constraint correction due to the *Suture* algorithm's greedy synthesis strategy. Node B must have a minimum timing constraint arc of 200 added between it and Node A. Otherwise, *Suture* will generate B as early as possible after A and will not be able to satisfy the constraints on C that require it to be a minimum of 300 after A and exactly 100 after B.

This configuration of nodes and constraints can arise not only from the specification but also from the addition of data dependency arcs to the graphs. For example, there may be an event that must occur a specific amount of time before the transducer outputs some data. These constraints are translated at the same time as specification constraints rather than by complicating the process of interconnecting the graphs.

Compression of Synchronous Events

5.2.5

Synchronous signals are different from asynchronous signals in that their logic level at the time a clock edge occurs is important rather than transitions in logic levels. However, *Suture* constructs circuitry to generate events, not levels. These two ways of looking at things are usually compatible except when there are no timing constraints to separate two consecutive events on a synchronous signal. This is the situation depicted in Figure 5.7. The third event on the top trace is not constrained to occur one cycle later than the second. It could, in fact, occur at the same time as the second event, meeting its implicit ordering constraint. This is shown in the bottom trace of the figure. *Suture* must be able to synthesize circuitry that *compresses* synchronous events in order to generate the most efficient circuitry possible. If not, a cycle is wasted generating an unnecessary event, namely, the second one of the top trace.

Of course, these events can only be compressed if they are not used to directly cause another event. For example, if an ordering constraint emanated from the second event of Figure 5.7, then it could not be compressed. This is because another event would be awaiting its occurrence so that it could fire. To determine which synchronous events can be compressed, the time intervals of occurrence are again pressed into service. Any output event on a synchronous signal with no emanating ordering arcs whose following event is also an output

event and has an identical lower bound for its time interval can be compressed. These events may be single nodes or super-nodes and are marked as compressible for *Suture*.

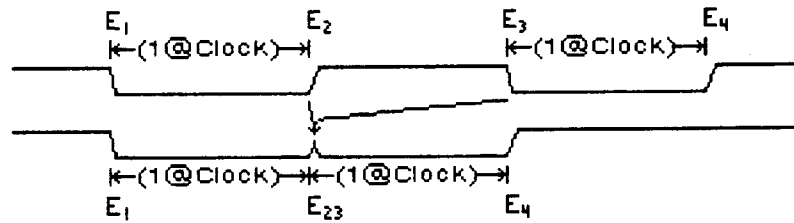


Figure 5.7. A specification of synchronous events that can actually overlap. The top trace shows the original specification, the bottom trace shows a valid implementation that meets the constraints. Note that there is no minimum constraint between the second and third events in the top trace (E2 and E3). The appearance in the lower trace is that two events have been eliminated. The glitch (E23) is for illustrative purposes only. An event graph can be annotated so that *Suture* can generate these synchronous event sequences. It must be capable of doing this, otherwise it would generate circuitry that wastes an extra cycle in generating the two compressible events.

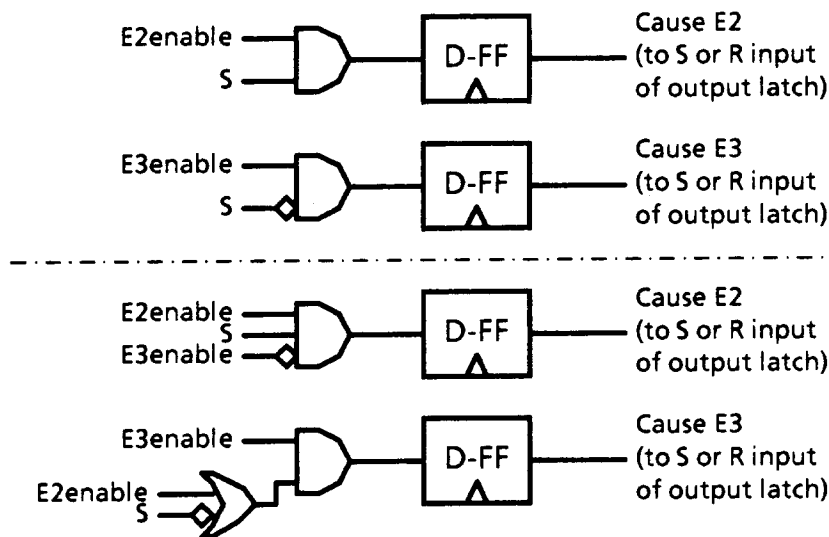


Figure 5.8. Example of the changes in template logic required to compress synchronous events. Above the dashed line is the circuit that would be used for the specification of Figure 5.7 if the events were not to be compressed, below are the modifications that make compression possible. Basically, E2 is generated only if it is not possible to go to E3 directly (i.e., E3 is not enabled) and E3 is generated in the normal manner as well as if E2 is enabled (implemented by the OR gate).

Suture actually generates all the events and compresses them by simply allowing two (or more) to happen at the same clock edge. This must be done so that if an event is delayed, the event that it would have compressed still occurs. This would be the case in Figure 5.7 if the third event were delayed a cycle waiting for an input, the second event would still have to occur to meet the one cycle constraint between the first and second events. If this is not the case (e.g., the third event cannot be delayed by any other events), then the compressable event can be completely eliminated after its constraints are translated (see section 5.2.7).

The changes to the synthesized circuitry are all to the inputs of the template AND gate (see Figure 5.8). For the case of Figure 5.8 without compression, the second event (E_2) is generated by a condition $E_2enable$ (determined from its incident ordering arcs) ANDed with the value of the signal (S) at the time the event is to occur. The third event (E_3) is generated by a similar AND gate with output $E_3enable$ ANDed with the complement of S . With compression, the enabling conditions for E_2 are changed to $E_2enable$ ANDed with S and the complement of $E_3enable$. For E_3 they become $E_3enable$ ANDed with the OR of S complemented and $E_2enable$.

Splitting of Signals

5.2.6

Another set of graph transformations split tri-stated and bidirectional signals. A signal that includes a tri-state level anywhere along its trace, or is viewed both as an input and output at different times, is split into three: one to carry the logic value of the signal when it is viewed as an output, one to carry the logic value of the signal when it is viewed as an input, and a third that determines when the output driver is enabled. A bidirectional signal must have its driver disabled when it is being used as an input. A special case exists for open-collector signals where the driver can be disabled by a logic 1 output value. Therefore, open-collector signals are split into only two separate signal wires (output and input).

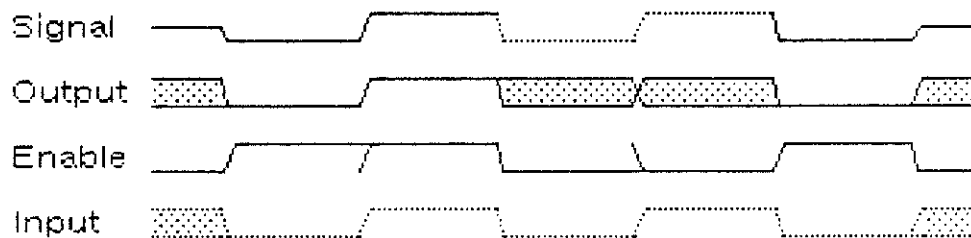


Figure 5.9. A tri-statable or bidirectional signal is split into three: one to carry the *output* value, one to carry the *input* value, and one to *enable* the output. In this example, six extraneous events are generated: three on the output signal, two on the enable signal, and one on the input signal (see section 5.2.7).

An example of how a signal is split is shown in Figure 5.9. Each event is split into three events that are bound together by a simultaneity constraint into a super-node. Some don't care events are created because it is irrelevant what the output logic value of a signal is when it is tri-stated or being viewed as an input. Computed signals are also split, their output signal has no events, however, its logic value is determined by the Boolean and latching expressions in its specification.

Extraneous Events

5.2.7

There are three types of *extraneous* events. The first type is a don't care event introduced by the designer in a *Waves* diagram or generated by the splitting of a signal (see Figure 5.9). The second type is an event that would change the logic level of a signal to the same level that was on the signal prior to the event occurring. This type can only be introduced by signal splitting. The third type occurs only when the first event on a signal would change the logic state of the signal to its quiescent level. This last type is caused by signal splitting or by signals that have different initial transitions in different operations (e.g., if a signal changes from tri-state to logic 0 in one operation and tri-state to logic 1 in another, the logic 0 transition will be extraneous if the signal's quiescent level is set to logic 0).

A *quiescent level* is determined for all the signals of the transducer. This is one of the few steps that requires information from the specifications of all the operations. Since *Janus* imposes the restriction of atomic non-overlapped operations, all signal traces in all operations must start and end with compatible levels. A compatible set of levels is one for which a logic value exists that satisfies all the logical constraints (e.g., logic 0 is acceptable for signals whose traces start or end with valid, don't care, or logic 0).

Extraneous events are detected with a single pass over the event graph after signals have been split and quiescent levels have been determined for each signal. They are marked as extraneous and are ignored by the synthesis steps of the Suture algorithm. No circuitry will be synthesized for these events. They are not removed from the graph because some constraints may propagate through these events.

Data can be transferred through the transducer from one interface to the other or loaded into or read out of internal state elements. Furthermore, the transfers may occur through dedicated wires or multiplexed wires that carry more than one data item. These wires may also be required to carry specific logic levels during the course of an interface operation. The circuit elements required to implement the data paths include input/output pads, latches, multiplexers, and their control logic.

Data Transfers Through the Transducer

5.3.1

All data transfers, whether they use internal state of the transducer or not, are treated in the same way during synthesis. Each input data item must have an associated latching condition. These conditions are derived from the tail nodes of the data dependency arcs added to the graph.

The next step in synthesizing the data path of the transducer is determining whether or not the data in each transfer needs to be latched. This is a trivial process that involves ordering the assertion and deassertion nodes of the data. Again, the time interval of occurrence is used to place the nodes on a time axis. If the two output nodes (assertion and deassertion) always occur between the two input nodes then there is no need for a latch (i.e., the input data is always valid when it is needed for output). Otherwise, a latch is generated and controlled by the latching condition found during the data dependency arc generation step (see section 5.2.3).

Multiplexed Data Transfer Paths

5.3.2

Signals wires that carry more than one data item must be split into separate wires, one for each of the values they may carry. A multiplexer is used to recombine the separate signals onto a single wire. Figure 5.10 shows an example of a signal that carries two data items as well as specific logic levels that require a line of their own. The events on the various signals are generated in exactly the same manner as for tri-statable signals. A pair of events are generated to correspond to data assertion and deassertion.

If a signal only carries data items and does not require specific logic values then there will be no events on the corresponding *Value* signal and no circuitry will be generated for that purpose. If this is the case, one of the control inputs of the multiplexer can be eliminated.

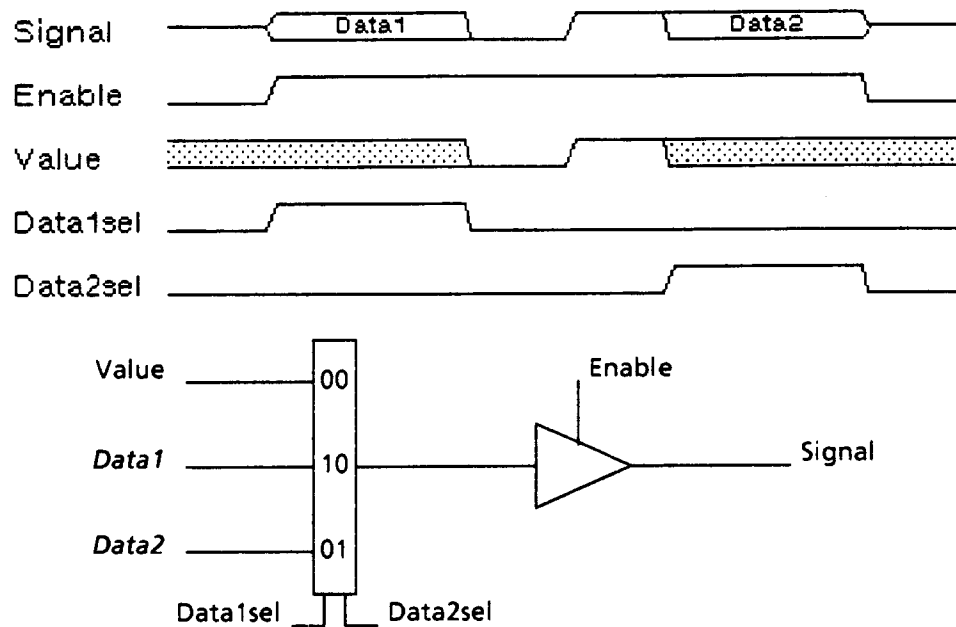


Figure 5.10. A multiplexer is required when a signal wire carries more than one data value during the course of an operation. In this case, *SIGNAL* carries *Data1* and *Data2* as well as asserting specific logic values at other times. To handle this case, a three input multiplexer is required. One input is for specific data values and the other two are for the data to be transferred. An enable signal is also generated so that the signal wire can be tri-stated.

The circuit framework within which *Suture* synthesizes the skeletal circuit consists of one set-reset-dominant (S-R*) latch for each output signal. Signals that have a quiescent level of logic 0 (active high) use the output of the latch directly while those with logic 1 (active low) use an inverter on the latch output. This implies that the templates that generate falling events on active low signals should be tied to the set input while those for rising events to the reset input.

There is another latch in the transducer circuit for each interface operation. It is used to keep the output signal S-R* latches reset (i.e., the outputs in their quiescent state) when the operation is not in progress. The *operation latch* is set by the starting event of the operation (or the event that causes the starting event, if the starting event is an output) and reset by the ending event of the operation. Figure 5.11 shows the implementation for an output signal latch.

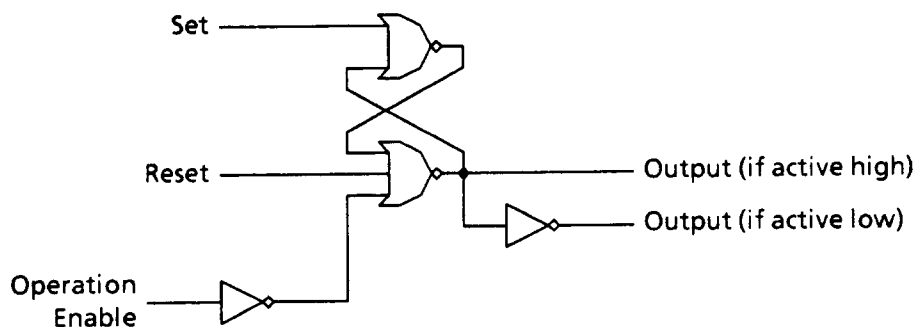


Figure 5.11. The implementation of an output signal latch. An extra reset input is used to force the output to its quiescent state when no operation is in progress. If a signal is used for more than one operation its operation enable signals will be ORed together during the optimization steps.

Synthesis of the Skeletal Circuit

5.4.1

The latches for each output signal and operation set the framework within which *Suture* will synthesize the control circuitry of the transducer. At this point, the event graphs contain only control events and input/output pads have been assigned to all the transducer signals. The algorithm stitches together the output signal latches with templates chosen on the basis of the incident arcs to each output node. This process and the templates are described in section 4.3.1.

Once the templates have been chosen, *Suture* must ensure that all the timing constraints are met and no race conditions occur. Before this can be done, intervals of occurrence for each of the nodes must again be obtained. This time, however, the algorithm can use the expected delay of the actual circuit elements to arrive at more accurate time intervals for the output nodes. It must still use the original constraints for the input nodes.

This information can be used to mark each node with the logic values of the other signals just prior to the occurrence of the events it represents. A step required by the race elimination algorithm (see section 5.4.3). Node marking is a two step process. A depth-first traversal of the graph (using only the ordering arcs) is performed from the first event of each signal. Nodes on a path between any two consecutive nodes of the signal can be marked with a logic value for the signal. Actually, a pointer to the corresponding node on the signal is stored (see section 5.4.3). Nodes that do not fall on such a path cannot be marked for the signal currently being processed.

After all the signals have been processed, the interval of occurrence algorithm is run for each node that has at least one signal whose logic level could not be determined by the depth-first search. The node can be marked if it can be placed at an unambiguous point in time relative to the events on the unmarked signals. If an ambiguity is present then there is no way to mark the node with a definite logic value.

The complexity of the synthesis algorithm is dominated by the number of times the interval of occurrence algorithm must be run. This algorithm is used for many purposes and may be used as many as N times, where N is the number of nodes in the event graph. With an average complexity of $O(E*N_{max})$, this makes the total complexity of the synthesis process $O(E*N*N_{max})$. In the worst case, with E equal to N^2 and N_{max} equal to N the complexity could be $O(N^4)$. However, in a more typical case for interface specifications, E is equal to $2*N$ and N_{max} is a small fraction of the total N . This makes the average expected complexity $O(N^2)$.

Local Corrections for Constraint Satisfaction

5.4.2

Local modifications must be made to the circuit to correct for timing constraints that are not met and to eliminate race conditions. These modifications are local in that they only modify a small part of the circuit, possibly removing some circuitry and adding newly synthesized logic. The overall structure of the skeletal circuit remains intact.

Timing constraints are straightforward. Violations can be identified by using the time intervals computed from the actual circuitry rather than the constraints for each of the output nodes. There are two cases of minimum timing constraint violations each of which is resolved by adding delay elements. The first case occurs when there is also an ordering arc between the two nodes. A delay element is added to the corresponding input of the circuit template used to implement the too early node. The amount of the delay is equal to the error computed from the

interval of occurrence. The second case occurs when there is no parallel ordering arc. In this case, an extra input is added to the template and the delay element is added as before.

Maximum timing constraint violations are quite different. The circuitry used to generate an event was simply too slow. The priority given to performance in the *Suture* algorithm ensures that events are generated as quickly as possible. The timing constraint translations performed during the process of preparing the event graph (see section 5.2.4) guarantee that events do not occur too early when they have a maximum timing constraint to a later event. A specific correction can be made for certain synchronous events, that is, those caused by other events synchronous to the same clock. In this case, there may be enough time to generate the event directly, without going through the D-type flip-flop of the template. However, this is only possible if the setup and hold times of the synchronicity constraint can still be met.

Other types of maximum constraint violations are referred to the designer who will either improve the circuit library by providing faster circuit elements or relax the timing constraint that is violated. However, a maximum timing constraint violation may not always be an error but may be corrected by adding more timing constraints on input events. This more complete specification of time intervals for these events may enable *Suture* to guarantee that the constraint will not be violated. An example of this is shown in Figure 12.

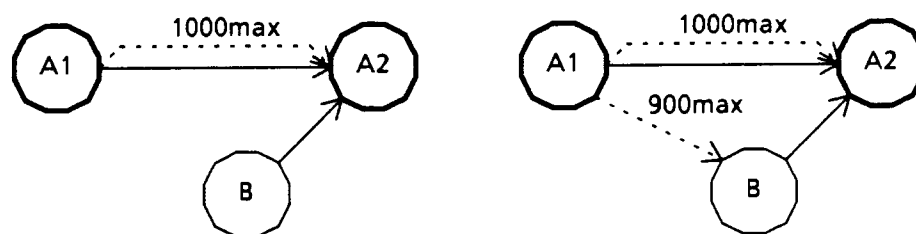


Figure 5.12. The addition of timing constraints can improve checking of maximum timing constraint violations. In the graph on the left, no assumption can be made about when the event of node A2 will occur. Therefore, the 1000max constraint will be flagged as possibly violated (as would be the case if B occurred late). If more information is known about the inputs to the transducer (i.e., the interval of occurrence of input events can be more completely specified), then *Suture* may be able to determine that the constraint will be satisfied. In the graph on the right, as long as it does not take longer than 100 time units to generate A2 after B occurs, the constraint will be met.

Local Corrections for Race Elimination

5.4.3

The elimination of race conditions also relies heavily on the time intervals of occurrence algorithm. Again, there are three different types of race conditions and the corrections for each are described below. All three may require the addition of a set-reset latch to trap an

event. The latches are reset by the same operation latch output signal as the output signal latches. The three types of race conditions are detected and corrected in the order in which they are described below.

The first type of race conditions occurs when a synchronizer misses an input event because another event on the same signal occurs too close to it in time. This race condition can be detected by running the time interval of occurrence algorithm on every event that is synchronized. If the following event on the same signal is not constrained to occur at least one clock cycle later, then a set-reset latch is added to trap the event.

The second type of race condition occurs when an enabling condition for an event never actually occurs. This happens when two events are ANDed together in a template and another event occurs on one of the signals too quickly. This prevents the Boolean condition from ever being satisfied. The first event must be trapped and saved.

The test for this condition entails partially ordering the immediate predecessor nodes (the tail nodes of all the ordering arcs) of the node in question and the immediate successors (the head nodes of the implicit ordering arcs) of these nodes. The partial ordering is achieved by using ordering arcs and intervals of occurrence. If there does not exist a cut through the partial ordering where the condition is true then a potential race is present. As with the previous case, the fix here is to add a latch to the signals whose events are too closely spaced and to use the output of the latch as input to the AND gate of the template rather than the signal itself.

The last type of race occurs when a Boolean condition that causes an event to fire occurs more than once during the course of an operation (i.e., at other nodes). It may occur too early, before the event is supposed to fire, or too late, after the next event on the signal has already occurred and changed the signal to a different logic value. In either case, an erroneous event is generated.

This type of race condition can also be eliminated by adding extra inputs to the AND gates of the templates. The simplest solution is to find another signal (or signals) that has a different logic value for the nodes where the condition is true. These *discriminating* signals are determined by the signal markings on the nodes. A signal or signals must be selected so that the new Boolean condition (i.e., the ANDing of the previous condition with the discriminating signal) separates the nodes into two sets: a *true* set and a *false* set. The true set, must, of course, contain the node under consideration (referred to as the *target* node). However, it may also contain nodes that will occur before another event occurs on the same signal as the target node. This is simply a check that these nodes are marked with the target node in their signal value markings. The correction is simply to add the discriminating signal as an input to the AND gates of the templates of the true nodes, and its complement in the case of the false nodes.

It may not always be possible to find a discriminating signal. One may have to be constructed. This is done by identifying a node that will partition the nodes for the which the original Boolean condition was true into acceptable true and false sets. The event represented by the node is used to set a latch that records its occurrence. The newly created signals, the output of the latch and its complement, are then added as inputs to the AND gates of the templates. In

the worst case, this can result in circuitry that will actually have a latch for every event on each signal.

The races are corrected in this order because the correction of the latter steps may need to be applied to the modified circuits resulting from the previous steps. Also, after each new latched event is created, the nodes of the graph are marked for this new signal. The result of the correction can potentially be used to correct other race conditions.

Simple extensions to *Janus* and *Suture* allow the algorithms to comfortably handle specifications with looping and conditional event sequences. The most important point is that diagram segments representing these constructs are treated as independent diagrams. In fact, many of the interval of occurrence calculations are performed independently on each segment.

In generating event graphs, special care must be exercised while merging when conditional and looping segments are present. Any merge points with another diagram must be placed so the diagrams are combined along a single path of control. That is, the graph that results after the merge must have a single root for all possible conditional executions. An example of a malformed merge is when two merge points are placed in mutually exclusive diagram segments. Checks exist for this and other inconsistent specifications (see Appendix C).

For conditional event sequences, a set of mutually exclusive latches, one for each alternative sequence, is created. When a segment enabling event occurs, a latch is set. Its output is an input to the AND gate of all the templates that generate events within the enabled segment. Setting the latch also disables the latches for the alternative paths. Enabling events that include timing constraints (see section 3.3.2) use the same circuitry except that the input to the latch is delayed by the value of the timing constraint.

Looping event sequences require a counter in addition to a latch. The counter is used to keep track of the current iteration of the loop. It is incremented every time the loop enabling event occurs. Its output is used, as with the latches for conditional sequences, as input to the template AND gates. Later combinational logic optimizations will simplify these logic networks. The latches and counters are reset by the same operation enable signal used for the output signal latches.

The output signals of the counter are also available as internal signals of the transducer. To specify conditional behavior based on a loop iteration, the specification need only refer to the *LoopCnt* state bits. The number of bits is based on the maximum iterations of the loop and is currently limited to 16. Some loops may never require the outputs of their loop counter. For example, in the indefinitely looping segment of Figure 3.8, there is no need for a counter and the logic optimization subroutine can eliminate it from the final circuit since its outputs will be unused.

A condition imposed on diagram segments is that their enabling event be the first event in the segment to occur. This can be verified with time intervals of occurrence. The reason for this requirement is that the synthesis framework used by *Suture*, the latches and counters used to control the firing of events, relies on this property.

The last part of the synthesis procedure is the optimization of the circuitry generated by the *Suture* algorithm. The objective is to achieve high-performance, not compact circuitry. The circuits *Suture* constructs use simple elements so that local corrections as described in sections 5.4.2 and 5.4.3 can be made easily. However, the flip-flops and synchronizers used in the templates can be combined into a smaller circuit. Furthermore, *Janus* generates a completely separate circuit for every interface operation. One would expect that many of these operations are quite similar and could easily share circuitry.

The sequential logic optimizations needed to combine the templates are the subject of this section. Applying them to the circuits generated by *Janus* creates many multi-level logic networks. This logic can be further optimized by using standard combinational logic minimization techniques [Brayton87]. These methods extract common sub-expressions from the Boolean equations to minimize the number of gates in the resulting circuitry.

Merging Across Operations

5.6.1

The different circuits *Janus* generates for each interface operation can be combined by a very simple recursive procedure. Each output signal's S-R* latches are combined into a single latch with multiple set and reset inputs. These inputs are themselves further combined if they are outputs of the same template type. This is often the case for synchronous signals. All the D-type flip-flops are merged into a single flip-flop with an OR gate whose input is the output of the AND gates of the templates being combined. This combination is also performed for mixed flip-flop and synchronizer templates.

Circuit elements can also be merged by looking at the input signals of the transducer. Some signals may be repeatedly delayed along different paths. In this case, the delay elements can be merged and a single delay path used to generate all delayed versions of the signal. A similar situation exists for signals that are synchronized more than once. It is important to make sure that there is only one synchronizer on any one input signal so that inconsistent states do not arise (i.e., the two synchronizer deciding on two different output states for the same input). The circuit is modified to fanout the output of the single synchronizer to where it is required and eliminate the extras.

The last set of optimizations involves the transformation of sequential logic. For example, if a latch has set and reset inputs of the form sX and rX then the latch can be replaced by a gated latch with control signal X and inputs s and r . This transformation was applied in optimizing the circuit of Figure 4.15.

Two other possible optimizations are shown in Figure 4.9. The S-R* latches are, in some special cases, replaced by simpler circuitry. For example, a clocked latch is replaced by a D-type clocked latch with input equal to $(Q + S)R'$, where Q is the output of the latch and S and R are its original inputs. Larger scale optimizations are possible when the inputs to an output signal latch are both output of flip-flops. In this case, the two flip-flops and latch are coalesced into a single flip-flop with a slightly more complex input equation.

Currently, only these three optimizations are implemented since the circuit library only contains these types of latches and flip-flops. After these optimizations are completed, the circuit is output by *Janus* and used as input to the combinational logic optimization and implementation tools that will realize the circuit in a specific implementation technology.

< This page intentionally left blank. >

CONCLUSION

< This page intentionally left blank. >

Conclusions and Contributions 6

Circuit interfaces are an important design abstraction. I have presented an interface specification methodology that can serve as the foundation for a new set of CAD tools that address interface issues. An interactive editor, called *Waves*, that supports this methodology has been implemented. To demonstrate the power of the abstraction, I have developed a new control logic synthesis method, called *Suture*, and applied it to the automatic synthesis of interface transducer logic in a tool called *Janus*. The designs generated by *Janus* are comparable in both size and performance to those generated by experienced designers.

This, the concluding chapter of this dissertation, is divided into two sections. The first section summarizes the contributions of the work presented in this dissertation, and the second section provides some ideas for future work in both interface specification and circuit synthesis.

This dissertation makes contributions to two areas of computer-aided design — specification and synthesis. I have developed a new approach to an important problem in each of these areas and applied it to practical examples. This summary, like the dissertation as a whole, is broken into two sections corresponding to these areas.

Interface Specification

6.1.1

In the area of specification, I have presented a new method for digital circuit interface specification that is based on the familiar timing diagram. As argued in Chapter 2, timing diagrams are an ideal method for interface specification. There are four reasons for this: (1) they are familiar to potential users, (2) they properly emphasize interface constraints rather than the internal circuit's realization, (3) they are a concise method of description, and (4) they can be extended to represent conditional and looping behaviors, combined to form larger specifications, and can describe arbitrary hardware via procedural annotations.

I have implemented an interactive editor to support this methodology. *Waves* not only supports the editing of timing diagrams but also performs interactive constraint checking for the user. An interface is specified by a collection of *Waves* timing diagrams arranged in sets, one set for each operation supported by the interface. Appendix B demonstrates that *Waves* can be used to represent the interface behavior of a variety of interfaces including commercial system busses, microprocessors, memories, and custom applications.

An abstract specification of circuit interfaces enables the development of a new class of CAD tools that deal with interface design and testing issues. The *Waves* editor provides a convenient way to generate data structures to represent these specifications. Their application in the automatic synthesis of interface transducers is the subject of the second part of this dissertation.

In summary, the contributions of this dissertation in the area of interface specification are:

- an abstraction for circuit interfaces that emphasizes interface constraints while remaining independent of the underlying circuit realization, forming a foundation for a new class of CAD tools that deal with interface issues;
- a new, mostly-graphical, interface specification methodology based on a formalization of the timing diagram; and
- an interactive editor, called *Waves*, that supports this specification methodology and can be used to generate interface specification data structures for these new CAD applications.

Transducer Synthesis

6.1.2

One of the applications made possible by the interface abstraction supported by *Waves* is the automatic synthesis of interface transducers. An interface transducer is the glue logic that connects two interfaces together. It may include both synchronous and asynchronous elements and must respect the timing constraints imposed by both interfaces. A transducer is required whenever a new custom chip is to be integrated into a system backplane or whenever two circuit blocks are to be connected.

I have presented a new synthesis method, called *Suture*, to deal with this task. The input to *Suture* is in the form of event graphs, derived from *Waves* diagrams, where nodes correspond to events and arcs to timing constraints. Events may be either synchronous or asynchronous. *Suture* performs the synthesis by traversing the event graph and constructing a skeletal circuit from a set of simple templates. Local modifications to this circuit correct for timing constraint violations and race conditions. I am not aware of any other synthesis method that supports both synchronous and asynchronous signals with timing constraints on their events. When the input to *Suture* is restricted to being purely synchronous or purely asynchronous, with no timing constraints, *Suture* achieves results comparable to the more specialized methods that exist for these classes of circuits.

The *Suture* algorithm forms the core of *Janus*, an automatic synthesis tool for interface transducers. *Janus* interconnects the event graphs of two interfaces based on the data transfers through the transducer and combines and optimizes the circuitry generated by *Suture* for each supported interface operation. The details of three practical transducer examples synthesized by *Janus* are provided in Appendix D. In all three cases, *Janus* yields results comparable in both size and performance to those achieved by experienced designers.

In summary, the contributions of this dissertation in the area of circuit synthesis are:

- a new control logic synthesis method, called *Suture*, that can be applied to specifications that include minimum and maximum timing constraints and both synchronous and asynchronous signals;
- a general synthesis method that yields results comparable to those of more specialized methods;
- a framework for synthesis and optimization based on a process employed by human designers; namely, to generate a skeletal design that is later locally modified to correct for deficiencies; and
- an interface transducer synthesis tool, called *Janus*, that generates the logic design of a transducer from *Waves* interface specifications and generates designs that are comparable in both size and performance to those generated by experienced designers.

The work described in this dissertation has many aspects that should provide fertile ground for future research. The specification methods and synthesis algorithms presented here have a wide applicability but can certainly be extended to cover a larger class of problems. In this section, I will present some ideas for future extensions and new application domains.

Of course, the implementations of *Waves* and *Janus* can stand some polishing and should be ported to a more standard environment (e.g., workstations that run UNIX, Common Lisp, and the X window system). Also, a larger collection of examples needs to be gathered to further validate the ideas that these tools incorporate.

Interface Specification

6.2.1

Although a graphical interface specification method is more amenable to most designers, a textual method is required if interface constraints are to be integrated with hardware descriptions. A viable approach may be an annotation method similar to that of BSI/SPS where the timing information, rather than being declared in the text, is placed in an accompanying timing diagram with cross-pointers between the text and the diagram. This type of facility may provide the best of both worlds, a graphical specification for timing constraints and a program specification for the related hardware components. However, major difficulties must first be resolved in ensuring that the two interconnected pieces of information — interface constraints and circuit functionality — are kept consistent.

A textual representation for interface constraints is also required if an interface view of a cell is to become a first-class design object like the more familiar internal structural views (e.g., layout, transistor, logic, etc.). It would be desirable to include interface information for a circuit block in an interchange language such as EDIF (Electronic Design Interchange Format) or a CAD database such as OCT so that the entire CAD tool suite can manipulate the information.

The graphical specification of cross-operation constraints requires further research so *Janus'* restriction of only atomic non-overlapped interface operations can be removed. A possible solution to this may be a *meta-timing diagram* editor that presents the user an abstraction of timing diagrams that only permits constraints to be expressed between synchronization points and hides the details of the rest of the diagram.

As interface specifications become part of the definition of every functional block, it will become necessary to generate interface specifications automatically. Extracting interface constraints from structural specifications is a difficult task, and there may not be enough information in the structural view for this to always be possible. A partial solution may be to

extract only the obvious constraints and permit the user to further annotate the specification in a verifiable way. Further in this vein, formal methods need to be developed that can generate an interface specification for a composite circuit given the specifications for its components (see section 2.3.2). Temporal logic is certainly promising in this area, however, an improved logic for dealing with timing constraints must be developed. A logic where the size of the specification does not increase exponentially with the number of constraints and is more syntactically appealing to potential users, both CAD tool developers and circuit designers.

Application of Interface Specifications

6.2.2

The new circuit interface abstraction presented in this dissertation makes feasible many CAD applications that were previously difficult or impossible. Some of these have already been discussed in some detail in section 3.4 and include applications from documentation and design to testing and evaluation. It serves to mention some of these again.

Documentation tools that use Waves interface specification as indexes to an English description of the interface are obviously feasible. It is certainly easier for a designer to use diagrams for different interface operations as indices to a large specification document rather than the one-dimensional and less-specific traditional index.

In design and evaluation, there is also a large potential for new tools. To continue the discussion of the previous section, if formal methods are available to combine interface specifications while a system is being assembled, then it should also be possible to check for compatibility between interfaces and suggest ways of improving the match. This can be further extended to the critique of interface designs and improved performance and throughput estimation tools.

Obvious applications exist in the areas of simulation and testing. As described in section 3.4, interesting algorithmic problems exist in generating parameterized input vectors that test the full spectrum of acceptable interface behaviors rather than just one. Similarly, output vectors should be validated against the specification rather than a fixed set of expected outputs.

Transducer Synthesis

6.2.3

The *Suture* algorithm for control logic synthesis can be applied to any event graph to generate a fast static circuit implementation for the input/output behavior represented by the graph. This includes conforming to timing constraints between the events. The most important directions for future research are primarily in the generation of the event graphs for different application domains and the optimizations that can be applied to the resulting circuitry.

Methods must be developed for synthesizing transducers with overlapping operations. Currently, the circuitry for one operation is synthesized independently from that of the other operations and there is no framework for including cross-operation constraints.

Janus can synthesize only transducers with identical width data paths on both interfaces. Techniques for multiplexing data and for serial-parallel conversion need to be investigated. This will require the generation of internal signals to control the multiplexers and shift registers. The actual logic synthesis should fall into the class of specifications *Suture* can handle. More generally, it should be possible to generate circuits that perform two 16-bit operations on one interface for the corresponding 32-bit operation on another, possibly modifying addresses between operations.

Other issues include the ability to specify functions that are not simple mappings of events across the transducer. For example, a reset operation should reinitialize transducer circuitry as well as being mapped to the other interface. Deadlock issues also need to be addressed — independent operations may be pending on both interfaces of a transducer with neither able to complete. It is obvious that a need exists for higher-level abstractions for interface functions rather than the simple atomic operation model employed by *Janus*.

In the area of optimization, don't care information present in the interface specification should be exploited. For example, the event following a don't care event on a signal wire can be placed anywhere between the previous don't care event and its current position. The proper placement of such events, by moving them so as to be simultaneous with other events in the same time range, can result in the sharing of more circuitry by using it to generate more than one event.

Lastly, the synchronous portion of a transducer design can be optimized if its state transition table can be reconstructed. A new state assignment can result in a smaller number of state flip-flops by encoding the state of the synchronous outputs rather than relying on a state-bit for each output as is currently the case. There may be an especially large reduction if flip-flops have to be used to delay some signals in order to meet constraints. The combinational logic between the flip-flops may also be reduced due to the smaller number of states.

Research in the directions outlined above holds promise for further expanding the range of specifiable behaviors and synthesizable circuits. However, some have a greater potential than others. In my view these are:

- the development of formal methods for reasoning about interface constraints together with hardware descriptions,
- extending the graphical methods for representing input/output behavior, hopefully relying on existing paradigms such as timing diagrams, and
- the continued development of synthesis methods that formalize the ad hoc design methods employed by experienced designers.

This dissertation represents a step towards solving the problems of interface specification and achieving the more general goal of high-level specification of digital circuits based on their *input/output* behavior. In the area of synthesis, it makes an important contribution to the synthesis of *practical* circuits, that is, circuits that must conform to external behavioral and timing constraints.

The two implementation efforts, *Waves* and *Janus*, demonstrate that the research contributions outlined above can be applied to practical problems. Furthermore, they represent a substantial improvement over current methods, greatly expanding the range of circuits for which behavior can be specified and a logic design automatically synthesized.

<This page intentionally left blank.>

APPENDICES

<This page intentionally left blank.>

Waves Implementation

A

Waves is an interactive editor for formalized timing diagrams. It is implemented in LOOPS, an object-oriented programming extension to the Interlisp-D programming environment, on Xerox 1109 workstations. Its user interface is based on decal windows, a new abstraction for mouse-sensitive window regions. *Waves* uses the access-oriented programming of LOOPS to implement incremental and interactive constraint checking.

This appendix is composed of eight sections. The first section is an introduction to the implementation medium and discusses the advantages of Interlisp-D and LOOPS for user interface programming. The second and third sections outline the data structures I developed to help with this task. Section 4 explains the implementation of the constraint checking provided by *Waves*. The fifth and sixth sections present the extensive editing facilities provided to the user. The seventh section contains a description of the data structures (i.e., LOOPS objects) used in *Waves* and their interrelations. The appendix concludes with a section on *Waves* portability issues.

Waves is implemented in Interlisp-D, a single address space, multi-process, Lisp programming environment running on Xerox 1109 Lisp workstations [Xerox86]. The general procedural interface to window and mouse operations available in Interlisp-D provided the primary impetus for using it to implement *Waves*. *Waves* relies heavily on LOOPS, a set of multi-paradigm programming extensions to Interlisp-D [Bobrow83, Stefik86]. LOOPS adds object-oriented, access-oriented, and rule-oriented programming paradigms to the procedural paradigm already available in Interlisp-D.

One feature of Interlisp-D windows is that they do not require a corresponding user process to implement their functionality. Any process can use any window and any window can signal any process. This many-to-many mapping is different than most multi-address space environments such as UNIX [Gettys86]. Mouse events in the Interlisp-D environment are handled by a system *MOUSE* process that calls a different procedure for each type of event that occurs (e.g., left button down, scroll bar event, etc.). The name of the procedure is obtained from the appropriate field in that window's data structure. Windows are created with default procedures, but their behavior can be changed by modifying the data structure to hold different procedure names. Once the procedure is obtained it is called with a standard set of arguments (e.g., a pointer to the window data structure) and evaluated within the context of the system *MOUSE* process.

The single address space makes this straightforward model for windows possible. Procedures evaluated under the *MOUSE* process can access the entire address space, therefore, there is no need to associate a user process with a window and accept the context-switching overhead of such schemes. The computation caused by the event ties up the *MOUSE* process and causes no new events to be handled. A problem with this approach is that no new computations can begin until after the event handling is completed. Long computations appear to freeze up the system and other interactions with the user through mouse actions (e.g., pointing at graphic objects) are not possible. Since *Waves* is a highly interactive editor that allows the user to specify most of the arguments for its operations graphically, it was necessary to introduce a lock to the data structure of each diagram. When an event occurs in a *Waves* diagram the first step is to obtain the monitor lock for the diagram. A different operation (e.g., popping up a different menu) or no operation at all may be performed if the lock is not available.

Most computations are much shorter than the few tenths of a second between human-generated mouse events and do not freeze up the system. In these cases, *Waves* allows the *MOUSE* process to be tied up and perform the computation. In cases where other user interaction is required, *Waves* spawns a new *MOUSE* process to handle these interactions. Interlisp-D processes coexist in a single address space, making it fairly inexpensive (i.e., the cost equivalent of a few procedure calls) to spawn the process when required. Whenever one of

the *MOUSE* processes returns to its quiescent state and finds another copy running, it kills itself.

Other useful features of Interlisp-D are the low-level graphics and user-interface routines provided with the system. Lisp functions are available to perform *bitblt*, line drawing and area-filling primitives. All the graphics routines are micro-coded and run much faster than would be expected of the 1109 CPU.

Many user interaction routines are also included as a standard part of the system. These include procedures for prompting the user for type-in in different screen windows and the specification of stable and pop-up hierarchical menus. The menus also take advantage of the single-address space and are capable of performing operations on any data structures present in the environment. Lisp expressions are attached to menu items at the time the menu is defined and are then evaluated with a standard set of dynamic bindings when the menu item is selected by the mouse. This makes it straightforward to pop-up menus after specific mouse events and execute different procedures based on the user selection.

Yet another feature of the Interlisp-D environment is the availability of *LOOPS*. *LOOPS* is a highly integrated set of extensions to Interlisp that provides the programmer with three more programming paradigms besides the procedural paradigm of Interlisp. These are the object-oriented, data-driven, and rule-based programming paradigms. Rule-based programming is the only one of the four paradigms not utilized by *Waves*.

Waves is implemented primarily in the object-oriented style. Procedural methods are used to tie the window and mouse operations to the message passing of *LOOPS* objects. An object abstraction layer for Interlisp windows was provided by colleagues at Xerox PARC to implement this connection [Lanning86]. The mechanism attaches special window and mouse operation functions to an Interlisp window data structure. These functions, rather than performing the operation directly, send a message to the *LOOPS* object that corresponds to the window. A corresponding object method is defined for each operation and performs the same operation previously performed directly by the window function. This extra level of indirection allows *Waves* windows, and other window objects, to be implemented as specializations of these simple windows.

Access-oriented, or data-driven, programming, the third and last paradigm used in *Waves*, is exploited to implement the constraint checking capabilities of the editor (see section A.4) [Bobrow83]. Data-driven programming is the dual of procedural programming. Rather than procedural calls causing data to be read or written, accessing data causes a procedure to be called. Two procedures are attached to the datum; one is called when the data is read and the other when data is written. Trigger and data protection mechanisms can be easily implemented using this mechanism.

The user interface to *Waves* is based upon a modeless interactive editing model. Diagram display objects, including all signal names, events, levels, constraints, and segment specifiers are mouse-sensitive. Each is implemented as a LOOPS object that is specialized to respond in its own way to different mouse buttons. All *Waves* operations are implemented as messages to the appropriate objects running under a copy of the Interlisp-D MOUSE process.

The underlying abstraction for building these object types is based on *decal windows*. I developed decal windows as a clean interface to Interlisp-D windows that contain mouse-sensitive regions. Each decal window object uses an Interlisp-D window with specialized default procedures. The procedures simply generate a message to the decal window object. Each sensitive region within the window is represented by a *decal* object. The mouse event handling methods of the window find the decal over which a mouse event occurred and send that decal object a message indicating the type of event and its position. Each window object has a list of all its decals, some of which may be visible and some invisible, that is, out of the area currently in view in the window.

Decals can be grouped into hierarchical *decal sets*. These behave in the same way as decal windows. Lists of visible and invisible decals are maintained and mouse events handed down to the lowest level decal object that covers the region. If there is no decal covering the position of the mouse event then the window itself handles the event by using a default method.

The hierarchical organization of decals and the differentiation between visible and invisible sub-decals is done for efficiency purposes. Decal sets shorten the search time to find the decal that will field the mouse event. Classifying decals as invisible and holding pointers in a separate list further speeds up user interaction by limiting the search to the complexity of the number of objects being viewed rather than searching the entire (and partly invisible) data structure. The savings in computation are repaid when the window view area changes and the decals must be reclassified. However, the user is more prepared to wait a couple of seconds for an updated view than for a sluggish response to a mouse action.

Each decal is also responsible for its own display and clear operations but not for maintaining a consistent view. Rather than leaving the responsibility for display update with each decal, a decal window maintains two queues: one for regions to be cleared and one for regions to be redrawn. When a mouse event causes drawing changes to occur, rather than updating its display as it moves or changes, each object simply places its display region on one or both queues. After the event has been processed, the window object empties the queues by clearing all the regions on the *clear queue* and sending messages to all the objects that overlap the regions on the *display queue* to redraw themselves. By using this method, the redisplay of the window proceeds faster since it is all done in one operation rather than one object at a time. The biggest advantage, however, is in program modularity. The overhead of maintaining a

consistent view of the diagram can be very high, since there are many display dependencies that could cause objects to be redisplayed many items in an attempt to always keep the drawing consistent. With the decal abstraction, all the programmer needs to decide is whether the object will have to be drawn (if it is newly visible), redrawn (if it has changed in appearance), or just cleared (if it is to become invisible) at the end of the current editing operation and place the appropriate regions on the queues.

There are two variations of the basic decal objects. One is the *extended decal*. An extended decal object has different mouse-sensitive and display regions. For example, a timing constraint is drawn with arrows to point to its two events. However, the sensitive region is only the area occupied by the text label. As an extended decal, it responds only to mouse events within this smaller region. For display operations it uses the larger region that includes the arrows. The methods defined for an extended decal distinguish between these two regions while those for the simple decal assume the same region. The second variation is the *documented decal*. It has the ability to hold arbitrary text, formatted by the Interlisp text editor, TEDIT. This is used in *Waves* for maintaining documentation with the various objects of the timing diagram. These two variations or properties of decals are embodied in *mixin* classes. A mixin class is only used to give subclasses special properties and is never used to instantiate an object directly.

The decal abstraction also forms the basis for a set of specification dialog windows used to collect parameters for *Waves* operations. *Spec windows* are decal windows with a set of decals, each of which is specialized to hold a different argument type. When a mouse event occurs within a decal, it prompts the user for a valid argument. Spec windows are an example of the type of interaction, as outlined in section 3.1.1, that requires an active *MOUSE* process. Since all editing operations are computed within the context of the system *MOUSE* process, whenever a specification window is opened, *MOUSE* process is spawned so that the user can continue to interact with diagram objects (e.g., by pointing at events to be connected by a constraint) and spec window decals. Furthermore, the diagram can be scrolled in both directions with the aid of the scroll bars and the entire contents of the current view can be seen by the user. This would not be possible if the *MOUSE* process were in a busy loop waiting for another argument for the current operation. All mouse operations that could cause a change to the diagram data structures must obtain the monitor lock of the diagram and are effectively locked out while a specification window is open. The operations, such as scrolling, that only change the view on the current diagram do not need to obtain the lock and can still be run.

There are three basic types of spec decals for three different types of arguments: boolean, value, and menu selection. Variations of the *BooleanSpecDecal* are used to display the value as ON or OFF rather than T or NIL. Boolean decals simply toggle their value when a mouse event occurs within their region. Value decals can be constrained to accept only numbers that pass a test contained in a standard method called *ChangeFn*. Specializations of the *ValueSpecDecal* exist for accepting only integer, non-negative integer, or non-negative float arguments. The user types the number in a window that is opened above the specification window. The *MenuChoiceSpecDecal* prompts the user with a pop-up menu of choices from which to select. Rather than a fixed menu specification, this decal requires the specification of a function that can be called to obtain the menu. This is useful when the choices may change in time, as is the case with signal names in a *Waves* diagram, as new signals are defined and named they are available as choices immediately. The object class inheritance lattice for decals and specification windows is shown in Figure A.1.

All the spec windows follow the same message protocol. They are called with a template structure that provides initial values for the arguments and they return a structure with the arguments entered by the user. Default values for the arguments can be set so that common collections of arguments do not have to be reentered each time the window is used. A menu attached to the specification window allows the user to indicate that the arguments are ready to be used, that the operation should be aborted, and set and get the default values for the windows's decals.

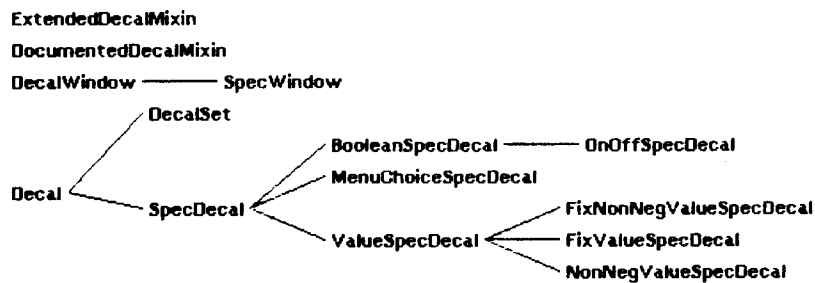


Figure A.1. The object class inheritance lattice for the decal abstraction and specification dialog windows. Classes inherit methods and slots from their *super* classes. The most general classes are on the left and the most specific on the right. All objects whose names include the word *Spec* are used for implementing the specification dialog windows and are specializations of the basic decal objects: *Decal*, *DecalSet* and *DecalWindow*. *ExtendedDecalMixin* and *DocumentedDecalMixin* are mixin classes used to generate extended and documented decals as described in the previous section.

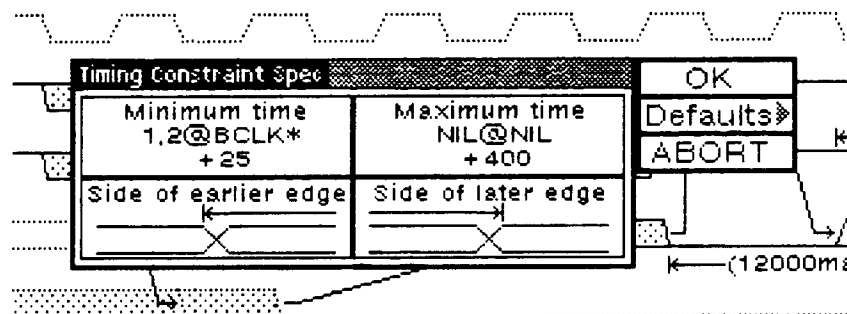


Figure A.2. A specification dialog window for collecting the arguments required for a timing constraint. This window includes four specification decals: two for time arguments and two for Boolean arguments. The minimum time is defined as 1.2 cycles of a period of the periodic signal BCLK* plus 25ns. The maximum time is defined simply as 400ns. The two Boolean values display the side of each edge from which the constraint is to be measured. The attached menu is used by the user to indicate that the arguments are ready to be used (OK), that the operation should be aborted (ABORT), or to set or get default values for the arguments (Defaults).

An example of a spec window is shown in Figure A.2. It is used for collecting the four arguments required for a general timing constraint. It is opened after the two events for the constraint have been selected. The minimum and maximum time for the constraint must be entered as well as the sides of the edges from which the constraint is to be measured. Minimum and maximum times are limited expressions composed of two terms: a number of cycles of a clock and an absolute time. The specification decal for these values is a new specialization of the basic *SpecDecal* that uses a pop-up menu to allow the user to select which

of the three parts of the expression is to be changed. If the periodic signal name is selected then a menu of choices, containing the names all the periodic waveforms in the diagram, is popped-up. The other terms prompt the user for a float or integer value. The two spec decals at the bottom are specializations of the *BooleanSpecDecal* and are used to specify the side of the edge to be used in measuring the time interval between the events. The value is displayed in pictorial form rather than as T or NIL.

Waves uses eight types of *SpecWindows* for collecting arguments for different operations. This abstraction greatly reduces the amount of program code required for user interaction and creates a straightforward and uniform graphical interface for the user. All arguments are visible while they are being specified and the operation can be easily aborted. The choice of the appropriate defaults for an application can also greatly reduce the amount of time required in the interaction.

Waves diagrams do not only support the specification of timing constraints, they also check constraints and highlight any violations. Events are highlighted when their positioning along the time line violates at least one applicable constraint. Constraint checking is performed in parallel with the editing operations by using the data-driven programming paradigm of LOOPS. All events to which a constraint applies have a *trigger*, or *active-value* in LOOPS terminology, assigned to the slot that contains their position in time. The active-value defined for use with the constraints causes a procedure to be called every time the event position is modified.

This mechanism makes constraint checking straightforward to implement. Each time an event moves, the triggered procedure sends a message to each of the attached constraints. If a violation is detected, the constraint places itself on a list of violated constraints associated with that event. When the event object redisplay itself, it checks to see if there are any elements on this list. If so, it highlights itself to indicate a constraint violation. Constraints that are checked remove themselves from the list if they had previously been violated but are now satisfied. In this manner, constraint checking is completely orthogonal to the editing functions. Entire collections of events can be readjusted without any code required to check the constraints. The constraints on the events that are moved will be rechecked automatically. At the end of the editing operation, when the decal window redisplay its contents, any events with violations are displayed with black bars above and below the edge. Violated general timing constraints are displayed with their text label inverted (see Figure A.3). No attempt is made to correct constraint violations. The user may be in the process of some larger editing changes. The final resolution of constraint violations is left to the user.

Synchronicity constraints are handled differently from the ordering, simultaneity, and timing constraints. Synchronous events do not have triggers attached if they only have the synchronicity constraints. These constraints are the only ones directly enforced by *Waves*. When a synchronous event is moved, it is moved to a position consistent with the setup and hold times of the constraints. During interactive editing the events are snapped to the nearest correct position. When an asynchronous signal is changed to be synchronous its events are adjusted to conform to the new constraint. Therefore, synchronous events can never be in a position where they violate the constraint.

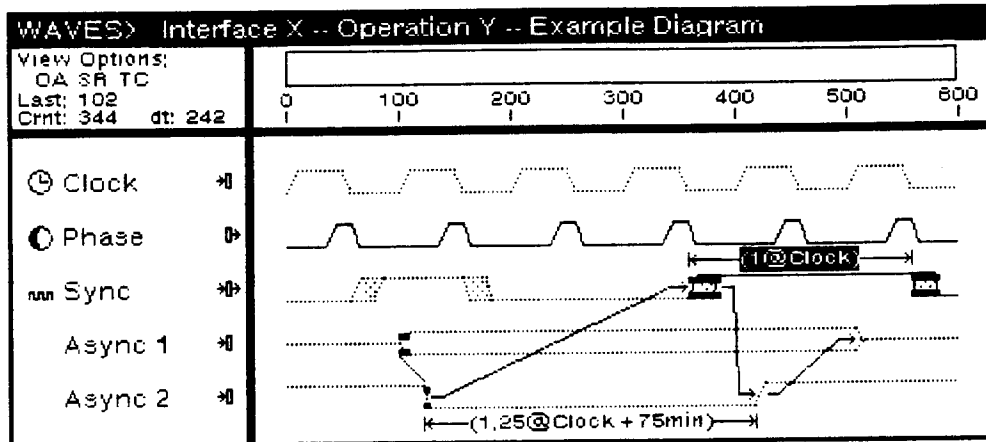


Figure A.3. An example of a *Waves* diagram with two constraint violations: a simultaneity constraint violation between the first events along the two asynchronous signals and a timing constraint violation between two events on the only synchronous signal. Violated simultaneity constraints are drawn as line segments connecting the two events, violated timing constraints are displayed with an inverted label. An event is highlighted if it has at least one constraint that is violated.

The *Waves* editor provides all the support and user interface functions for editing interface specifications. This requires additional functionality beyond the basic operations described in the previous sections. The user must be able to draw diagrams larger than the screen size and be able to locate diagram objects quickly through graphical cues (e.g., pop-up menus). Diagrams can be resized to any aspect ratio and size. In fact, all Interlisp-D default window operations are available. The diagram can be closed to be reopened through interaction with the *Waves* icon. The user can take a snapshot of a diagram view into a new window so that it can be used later, for reference, when another view of the diagram is being edited. Of course, the diagram can also be moved on the screen, buried beneath other windows, and brought back up to the top so that it is fully visible.

Initially, a new *Waves* diagram does not contain any signals. Clocks, clock phases, non-periodic signals, or computed signals are added through the use of a pop-up menu available in the title window. An arbitrary number of signals and clocks can be added. The diagram keeps expanding in the vertical direction and can be scrolled using the scroll-bar of the signal name window. Signals can also be made invisible by deleting them from the diagram. For example, many clock phases may be defined, but the user may not want to clutter the diagram with many similar traces. The phases can be specified and then deleted from the display. To completely eliminate a signal from the diagram it must be expunged. Pop-up menus using the names of the signals as items are used to undelete (i.e., make visible again) and expunge specific signals.

The editing functions on signals include renaming the signal, copying it into another, erasing all events on it, erasing any constraints related to its events, and shifting a signal trace in time. Electrical parameters of a signal can also be edited. These include the technology that determines the logic levels and input/output capacitances and currents of the signal. Rise and fall times of the signal can also be specified and imply the strength of the drivers required for the interface. Signal directionality can also be changed as can any synchronicity constraints on the signal. Extraneous events, that is, those between identical levels and with no associated timing constraints relating them to other events, are usually the artifacts of editing and can be removed from the diagram.

Operations on the other axis of the diagram can be obtained through the title bar, time line, and trace windows. The diagram is scrolled in time using the scroll-bar of the trace window. A pop-up menu on the time line allows the user to expand the time range of the diagram and position the view to start or end at any time point in the range. The diagram can also be rescaled from the default of 2 time units per screen pixel to anywhere from 1 to 32. The time line menus also permit the user to define tick marks to be placed over the trace window to guide the positioning of events and graphically define clock cycles. Any number of tick marks

can be defined, with a period and offset, and they can be made visible or invisible independently.

Pop-up menus available from the title window of the diagram provide a way to bring any constraints in the diagram into view. A menu of all constraints is presented to the user with the items identifying the signal names and times of the constraint's events. The event can be deleted, modified, brought into view, and blinked to identify itself on the screen. Similar menus can be used for all labeled points in the diagram.

Diagram segment bars in the time line window are used to specify and access the parameters of a diagram segment and modify the segment structure of the diagram. These include: highlighting enabling events, specifying the number of iterations for looping segments, editing the name of the segment, splitting or deleting the segment, and creating nested sub-segments.

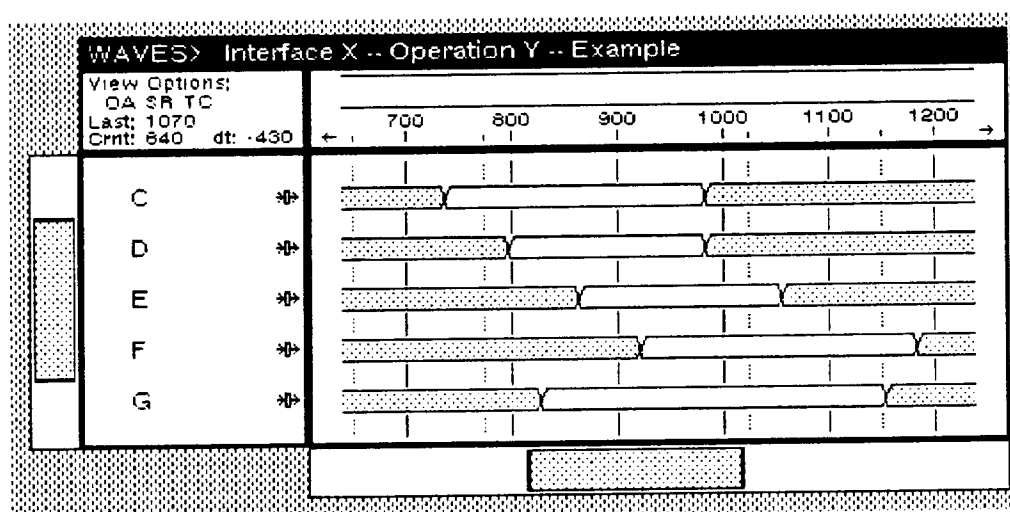


Figure A.4. A Waves diagram with its scroll bars. The shaded region in the scroll bars indicates the portion of the total diagram that is visible in the corresponding dimension. The scroll bars, like all Interlisp-D scroll bars, also support continuous scrolling and thumbing. The arrows at the edges of the time line indicate if the diagram extends out of the window in that direction. Two sets of tick marks are displayed in the trace window. The major set (in dark lines) has a period of 100ns. The minor set (gray lines) has a 125ns period and is offset by 25ns from 0.

Through the feedback window in the top left of the diagram, the user has control over which relations are to be displayed. The various types of constraints can be toggled on and off and menu operations can be restricted to only the currently displayed relations (AR/E, the all relations enable flag, toggles this facility). Furthermore, the A/I (active/inactive) flag, can be used to turn on and off the shading of inactive events (i.e. those that are only place holders for cross-diagram constraints).

For efficient interaction, editing operations must permit actions on entire collections of signals, events, and constraints rather than just one object at a time. Diagram segment bars provide a convenient way to get at the events and constraints within a segment. However, a more general way to specify a subset of the diagram objects is required.

In *Waves* it is possible to indicate any set of events by specifying ranges along both of the two axes of the diagram. By holding the shift-key, the user can select a set of signals along the vertical axis. A signal is selected by clicking a mouse button over its name in the signal name window. The selected signals are all displayed inverted until a signal is selected while the shift-key is up. The horizontal range is specified with the aid of *time interval markers*. These are two vertical lines that appear in the trace window at specific time positions. The user can interactively (or through time line window menus) place the markers at any time point. One, both, or neither of the markers needs to be positioned.

Whenever operations available through the title or signal name window are performed, they are applied to all events within these ranges. For example, to erase all events on two signals within a time interval, the user first selects both signals using the shift-select method and positions the time interval markers appropriately and then invokes the operation through the usual method (see Figure A.5).

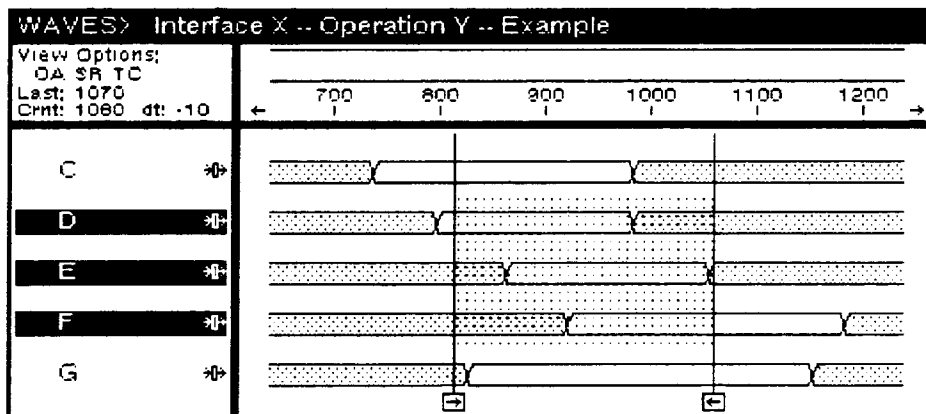
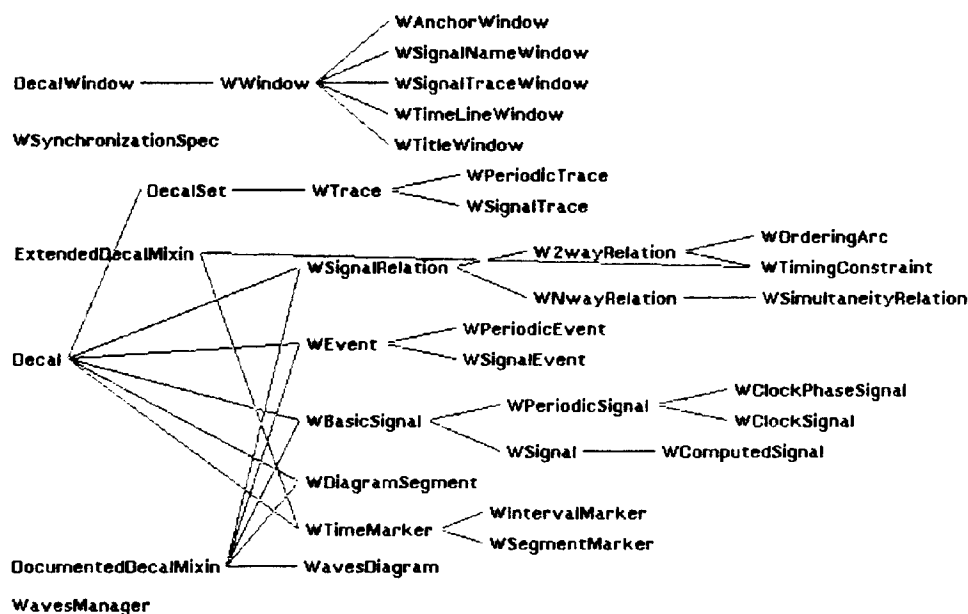


Figure A.5. A *Waves* diagram showing the use of interval markers and multiple signal selection. Four events are selected: the second in signal D, both events in signal E, and the first event of signal F. The shading does not appear in the diagram and is meant only to show the region being specified. The signals need not be adjacent. Operations can be selected that use the selected events as arguments. For example, the four events can be shifted along the time axis.

Another important region defining the structure of *Waves* diagrams is the diagram segment. The diagram segment behaves exactly as a diagram. Events that exist in one segment cannot be moved outside that segment. In fact, there is a strong interaction between the diagram segment and synchronous signals. Whenever synchronous events are moved they must still fit within the boundaries of the segment and also be aligned with the edges of their periodic signal. The events can move either interactively or through an editing operation selected from a menu. If they are moved interactively, then they always snap to a position within the segment. If they are moved by a more complex editing operation, such as a change in the period of their periodic signal, then they must be realigned to completely new points. These points may not be within the boundaries of the segment. For this reason, whenever such drastic changes occur, *Waves* adjusts the boundaries of all diagram segments so that each will still contain the same events as before the operation (unless they are deleted). A copy of all the events in a segment can also be generated. This occurs when an alternative segment is created. All the events in the current segment are duplicated in the new segment so that the user can begin editing from a template that may correspond closely to the final result. Most alternative segments differ only in a few events and enablers.

Segment boundaries can also be adjusted interactively through the use of the *segment markers* in the trace window. A segment marker appears at any common boundary between currently viewed segments. It behaves much in the same way as a time interval marker. A button event on the marker permits the user to move the boundary in either direction as long as no events would be placed in a different segment.



The second group includes specializations of the *DecalWindow* class. These are six classes corresponding to each of the five diagram windows described in section 3.2 and another class, *WWindow*, that holds the methods and slots common to all five window types. The windows have extra slots for their special characteristics. For example, the title window has slots for the interface, operation, and name of the diagram. The time line window holds the parameters

of the time line and the sets of tick marks. Of course, as specializations of *DecalWindow*, the sub-classes of *WWindow* hold pointers to all their decals.

Members of the third group are specializations of *DecalSet*. The classes in this group are the data structures that represent signal traces. There are two types corresponding to the two principal classes of signals: periodic (*WPeriodicTrace*) and non-periodic (*WSignalTrace*). They hold pointers to all the events along the signal traces. There are two varieties because it would be highly inefficient to create an event object for all the edges on a periodic waveform. Rather than doing this, the *WPeriodicTrace* creates the event objects on demand only when the event is needed to handle a mouse event (i.e., the user has clicked the mouse over the event to obtain a pop-up menu of operations). Periodic signal traces also handle all the display operations for the trace rather than relegating it to individual events as the non-periodic trace does. The common methods to both trace classes are held in the *WTrace* class.

The last group includes all the decals that exist in the signal name, trace, and time line windows. The three types of timing constraints between event pairs are specializations of the *WSignalRelation* class (*WOrderingArc*, *WSimultaneityRelation*, *WTimingConstraint*) and are further distinguished by whether they are relations that exist between two or any number of events (*W2wayRelation*, *WNwayRelation*). The synchronicity constraint (*WSynchronizationSpec*) is not displayed as a distinct object and is not a subclass of the decal because it expresses constraints on all the events on a signal. Events are separated into periodic and non-periodic events (*WPeriodicEvent* and *WSignalEvent*) with a super-class to hold common methods (*WEvent*) in the same way as traces. Time markers (*WTimeMarker*) are differentiated by the two classes of time interval markers (*WIntervalMarker*), of which a maximum of two exist, and segment boundary markers (*WSegmentMarker*). *WTimingConstraint* and *WTimeMarker* are also subclasses of the *ExtendedDecalMixin* because their display image is larger than the area to which they are mouse sensitive.

Signals are broken down into four classes corresponding to the basic signal types: clock (*WClockSignal*), clock phases (*WClockPhaseSignal*), synchronous or asynchronous signals (*WSignal*), and computed signals (*WComputedSignal*). The two periodic types share a common super-class (*WPeriodicSignal*) and common methods and slots of all signal types are defined in *WBasicSignal*. Signals are the decal objects visible in the signal name window and also hold pointers to the corresponding signal trace in the trace window of the diagram. *WDiagramSegment* is the only class that defines decals used in the time line window for segment bars. They also hold pointers to events that occur within their time range. As is evident from Figure A.6, all decal specializations, and the diagram itself, can have attached text by virtue of having *DocumentedDecalMixin* as a super-class.

The implementation of *Waves* can be broken into two independent parts: one concerned with input/output operations and the other with data structure manipulations. The input/output operations constitute all interactions with the user through the screen, mouse, and keyboard of the workstation. The data structure routines manage the creation of new objects and maintain consistent pointers between them. This division was consciously enforced in the implementation in the hope that it will make the task of porting *Waves* to other hardware and software environments straightforward.

Only the input/output routines require special attention. These are always the most environment dependent functions of any interactive application. The Interlisp-D/LOOPS code used in managing the data structure uses no special features of the Interlisp-D and LOOPS languages and can be easily translated into any other object-oriented language. In fact, it should be trivial to translate the code into the Common Lisp Object Standard (CLOS), that is becoming one of the most common languages in this class [Bobrow88].

The only LOOPS feature that cannot be directly translated is the active-value mechanism used in checking the constraints in *Waves* diagrams. However, it can be easily translated into a queuing mechanism that has the same functionality (i.e., whenever the position of an event is changed, pointers to the constraints on the event be placed on a queue). At the end of the interactive operation, all the constraints on the queue can be checked and highlighted if a constraint violation occurred. Changes to the code will be minimal as there are only three procedures where event positions are modified. All of the code for *Waves* uses static scoping even though Interlisp-D supports dynamic scoping. Similarly, the flexible argument passing of Interlisp-D was not exploited. These policies increase the modularity of the programs and permit other Lisps to exploit their optimizing compilers (e.g., Common Lisp).

The layer of objects in the decal abstraction provides a separation between the input/output functions and the rest of the program (see section A.2). When *Waves* is ported to an environment with different input/output primitives, this layer must be rebuilt to provide the same functionality in the new environment. The graphics functions required are well-defined and common to most modern workstation environments. The same can be said for the facilities for handling mouse and keyboard events [Gettys86]. Although the rebuilding of this new object layer could require considerable effort it is a well-defined and modularized problem.

The only other point that demands attention is that in Interlisp-D, *Waves* does not require a separate process but rather its functions are evaluated under the Interlisp-D *MOUSE* process (see section A.1). In other environments with more multi-processing protection and multiple address spaces, a separate process may be required. It should be a trivial task to set up a separate process for each *Waves* diagram within its own address space and forward all relevant user-interactions (e.g., mouse movement in the diagram windows) to that process.

To exist in a general-purpose CAD environment, *Waves* will eventually need to be rewritten to use a CAD database such as OCT [OCT87]. This is a trivial task given the object-oriented nature of both *Waves* and OCT. There could be a one-to-one mapping between *Waves* objects and OCT objects corresponding to an interface view of a cell. Using the editing capabilities provided for OCT is a different matter. OCT's VEM editor is ideal for applications where operations can all be written to follow the supported editing paradigm, namely, argument selection followed by operation invocation. *Waves* is not such an application, it relies on access to lower level mouse operations to provide a modeless interactive waveform editing capability. If VEM were reorganized to have hooks at these lower levels then *Waves* could be rewritten as a VEM application. This would be the ideal situation for *Waves*. It would be a timing diagram editing tool integrated with a CAD database, making its data structures available to other interface tools (see section 3.4), and running through an editor written for the X window system, enabling it to run locally on a user's workstation or remotely on a compute server.

Waves Specification Examples B

The utility of *Waves* is best demonstrated via a collection of sample interface specifications. This appendix contains *Waves* specifications for seven representative interfaces ranging from system busses to microprocessors to memories. For each interface the specification is divided into parts corresponding to the interface's operations. These examples are by no means complete. Only a subset of the interface operations available with each interface are specified. However, they do constitute a representative set of operations — similar to the subsets designers typically consider. The references provide a more complete description of the functionality of the interfaces.

System busses support two types of devices: masters and slaves. Masters arbitrate for control of the bus and then initiate data transactions to slave devices. Slaves merely service requests from masters. Each slave is allocated a region of the address space and only responds to requests with addresses within that range.

The Intel Multibus is a popular system bus for 16-bit computer systems [Intel82]. Arbitration for the bus is performed synchronously to a bus clock while data transactions may proceed asynchronously. There is a 20-bit address and a 16-bit data field for each transaction, both of which are negative logic. A separate signal (BHEN*) is used to distinguish between 8-bit and 16-bit transfers (in the specifications below it is always set for 16-bit transfers).

Five operations for the Multibus are specified in this section. There are two slave operations (read and write) and two master operations (read and write). The two master operations also depend on an arbitration sequence which is included in a separate section. Signal directions are as they would be from the perspective of a circuit being connected to the bus. Other operations supported by the Multibus include interrupts from slaves to masters and interrupt acknowledgements from masters to slaves. They are not described here for reasons of space and because they are not fundamentally different from the operations described below.

Slave Read

B.1.1

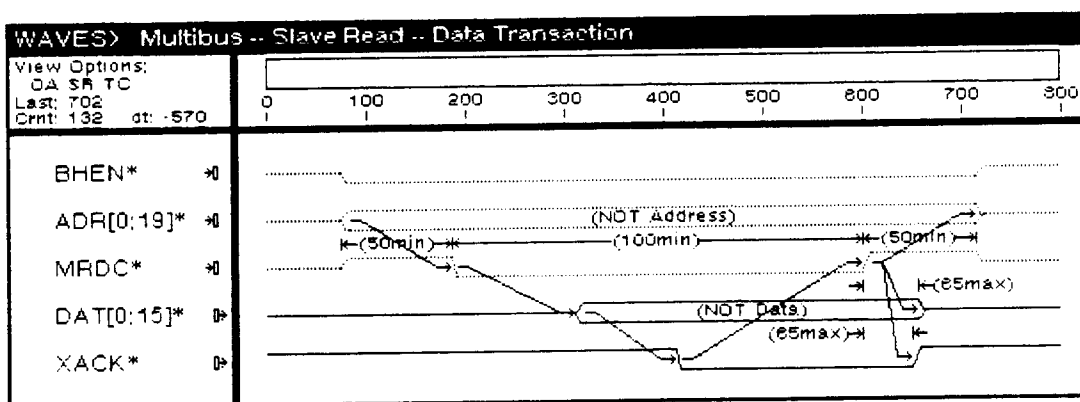


Figure B.1. The Multibus Slave Read operation.

The slave read operation is straightforward (see Figure B.1). A four-phase protocol exists between the MRDC* and XACK* lines. The address is valid while the command is asserted and the data is valid while the acknowledge is asserted. There are some address setup and

hold time constraints as well as minimum durations for asserting the command. Note that there is no setup time for the data before XACK* is asserted. This means that XACK* will need to be delayed before it can be used to latch the data (due to the setup time requirements of latch circuits). The operation begins when the address lines become valid. A Boolean condition on that event specifies a range of address values for which the slave will actually respond (i.e., begin and carry-out the operation).

Slave Write

B.1.2

The slave write operation is similar to that for slave read (see Figure B.2). Address and data are now treated in the same manner with identical constraints. Like the slave read operation, a Boolean condition on the first event is again used to limit the address range for which a device will carry out the operation.

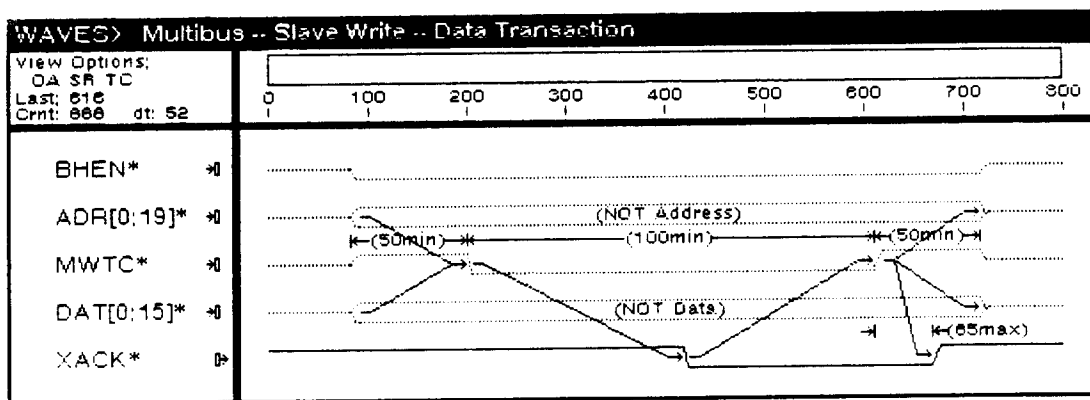


Figure B.2. The Multibus Slave Write operation.

Interrupt operations are simple variants of these two slave operations. A computed signal is used to latch the interrupt line and record that the slave issued the interrupt. It must then be possible to read and write this bit from the master that is interrupted. These are simply modified slave read and write operations. The address range condition on the first event now corresponds to the place in the slave's address space where the interrupt register resides (i.e., in this case the interrupt bit and possibly some other internal interface state). The data field on the read includes the value of the interrupt bit on one of its lines. On a write, the interrupt bit is cleared. See section 3.3.1 for a more complete discussion of computed signals and cross-operation state.

Arbitration for the Multibus, unlike the data transactions, is performed synchronously to the bus clock (BCLK*) (see Figure B.3). The master wanting to gain control of the bus first issues a request via BREQ* (for daisy-chain arbitration) and CBRQ* (for parallel arbitration) and waits for the BUSY* line to go inactive (other masters relinquish the bus) and its priority line (BPRN*) to go active (permitting the master to take the bus). The BPRO* line is used to implement a daisy-chain priority scheme and is represented by a computed signal whose Boolean function is $(OR (NOT BREQ*) BPRN^*)$.

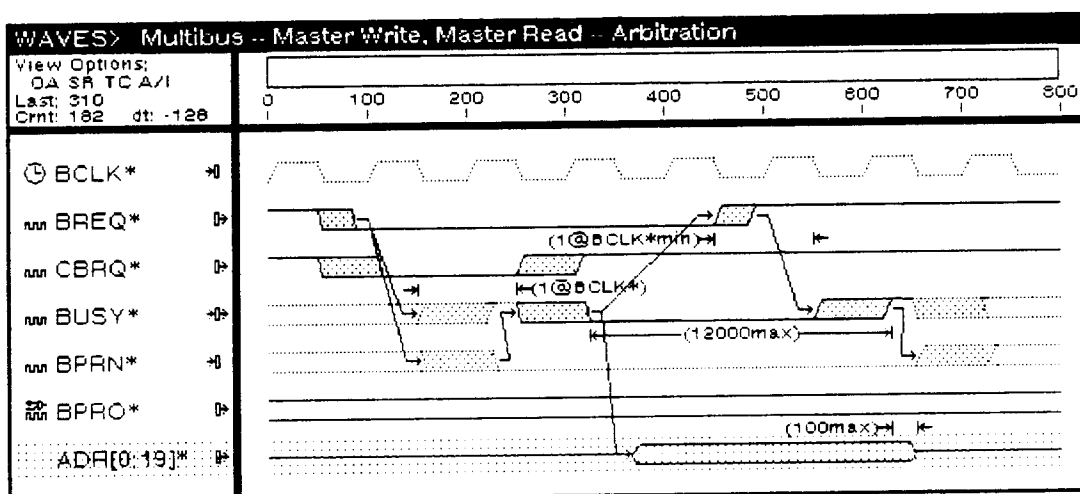


Figure B.3. The Multibus arbitration sequence.

The edges of synchronous signals are drawn stretched out to indicate the uncertainty of exactly where the edge will fall. The constraints on the edge guarantee that it will respect the setup and hold time requirement of the signal. That is why the edge is drawn as two edges with a don't care condition in between. The spacing between the two edges is from one hold time to the next setup time.

The shaded signal at the bottom of the diagram is not part of the sequence of events required for arbitration but is used as a place holder for a cross-diagram constraint (see section 3.3.3 for a discussion of cross-diagram constraints).

The master read and write operations are reflections of the slave operations (i.e., inputs and outputs are reversed). The events and constraints are identical (see Figure B.4 and B.1). The only change is that there is no longer a condition on the address lines because masters can issue requests to any address.

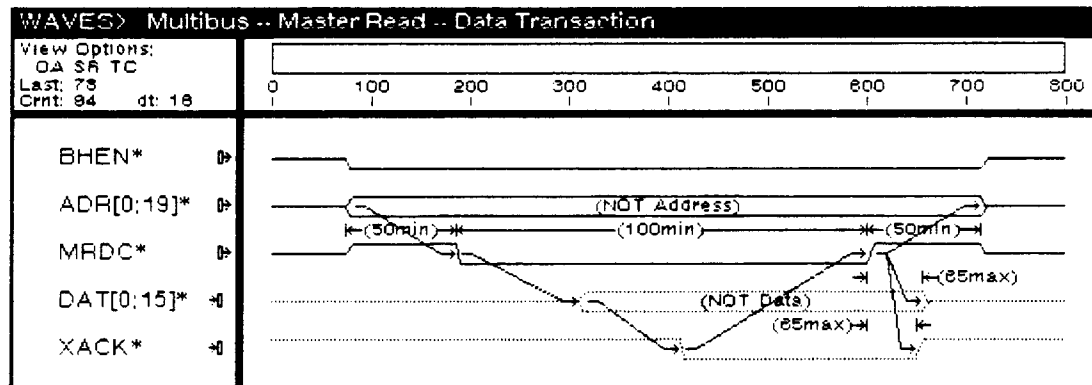


Figure B.4. The Multibus Master Read operation (combined with the diagram of Figure B.3).

This operation relies on the master first having control of the bus. Therefore, the specification for this operation consists of two diagrams, the one above and the arbitration diagram of Figure B.3. They are linked through the use of three sets of merge labels. One set links the first output event on the BUSY* signal with the first event of the read operation (the simultaneous assertion of BHEN*, MRDC*, and the address lines). The second set is used to synchronize the completion of the transaction (deassertion of BHEN*, MRDC*, and address) with releasing the bus (the last output event on BUSY*). The last set is used to transfer a timing constraint across diagrams, namely, the maximum timing constraint between the deassertion of the address and the BUSY* signal.

The master write is a reflection of the slave write and is combined with the arbitration diagram in the same way as the master read operation (see Figure B.5 and B.3).

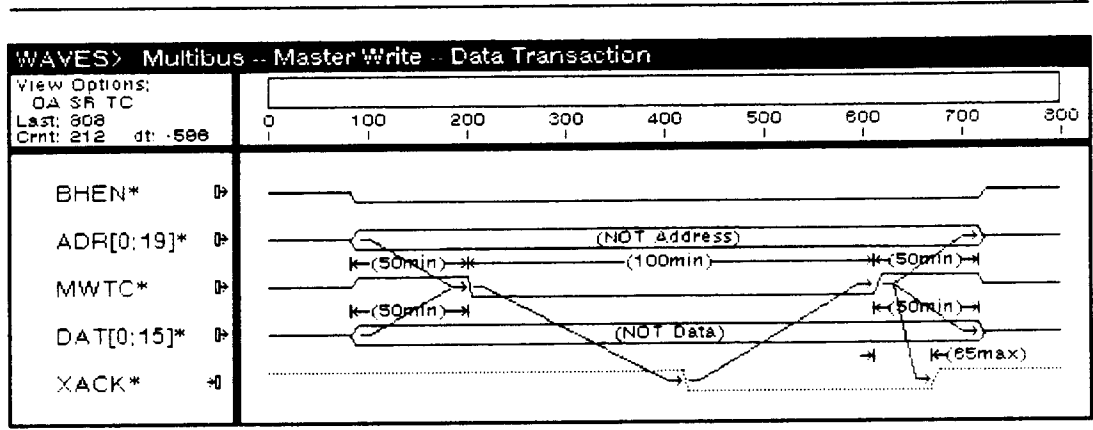


Figure B.5. The Multibus Master Write operation (combined with the diagram of Figure B.3).

The Multibus Design Frame (MDF) is a collection of circuitry that implements many of the functions required when connecting a custom chip to the Intel Multibus. It presents a simpler, more consistent interface to an internal circuit than would a direct connection to the Multibus [Borriello85]. While the Multibus has an asynchronous transaction protocol and a synchronous arbitration protocol and many timing constraints, the MDF presents a uniformly synchronous interface to its internal circuit and four basic operations: slave read and write and master read and write. Details of arbitration are handled by the interface. In Appendix D, *Janus* is used to synthesize the Multibus Design Frame logic from the specifications of this section and those of section B.1.

Slave Read

B.2.1

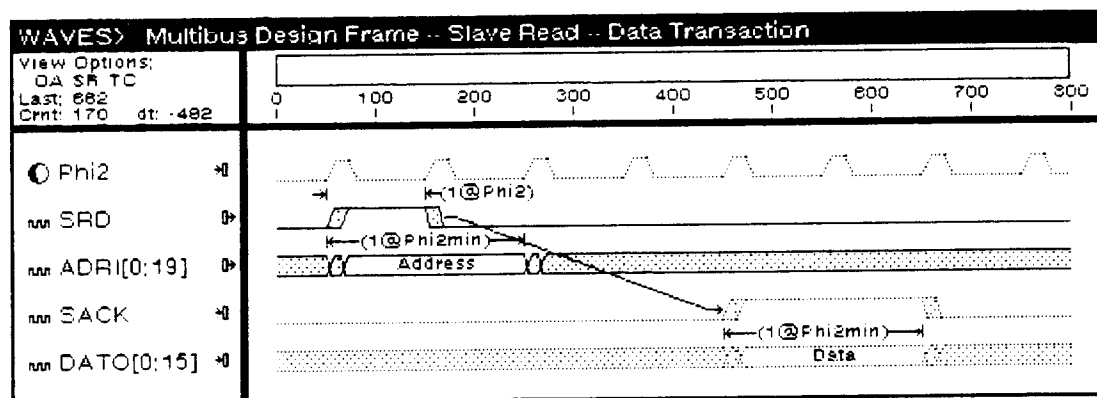


Figure B.6. The Multibus Design Frame Slave Read operation.

The MDF operations are straightforward to describe (see Figure B.6). The slave read operation consists of a pulse exactly one cycle wide to the internal circuit that signals an incoming read request that must be serviced (the address is valid for at least as long as the duration of the pulse). The internal circuit of the design frame is then expected to respond some time later with a pulse at least one cycle wide (and data to be returned to the requestor that is valid while the pulse is asserted). All signals are expected to be synchronous to the clock of the internal circuit, changing while Phi2 (the second phase of the clock) is asserted.

Slave Write

B.2.2

The specification of the slave write operation is almost identical to that of the slave read (see Figure B.7). The differences are as expected. The data is presented to the internal circuit at the same time as the address and no data is expected to be returned in the other direction.

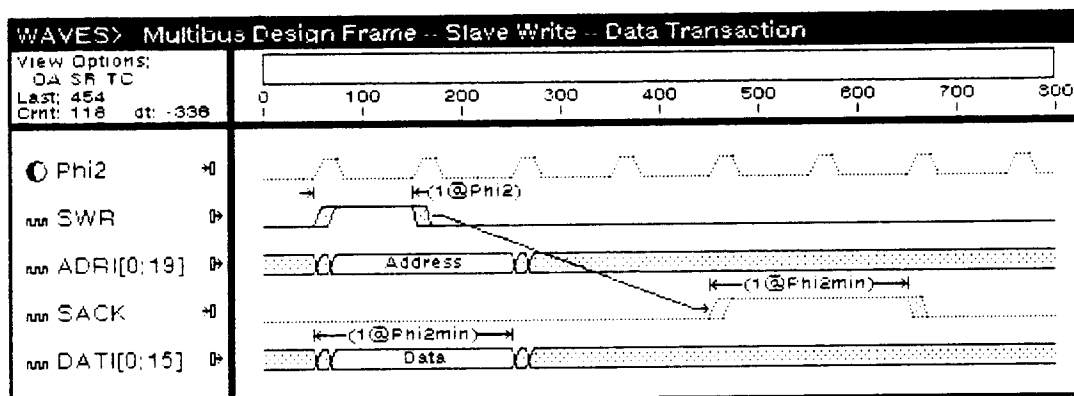


Figure B.7. The Multibus Design Frame Slave Write operation.

Master Read

B.2.3

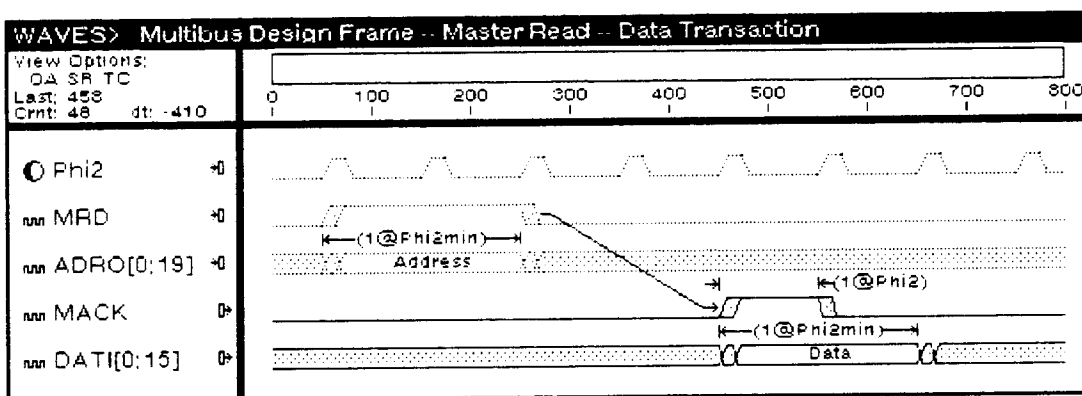


Figure B.8. The Multibus Design Frame Master Read operation.

The master operations are initiated by the internal circuit of the MDF. For a read, a pulse at least one cycle wide is generated to be followed some time later with an exactly one-cycle wide acknowledgement pulse and the data read (see Figure B.8). The MDF translates this request into a Multibus arbitration cycle and read transaction.

Master Write

B.2.4

Again, the diagram for the master write operation is almost identical to that of the master read (see Figure B.9). The simplicity of the MDF internal interface is obvious when one compares the diagrams of this section with those of section B.1.

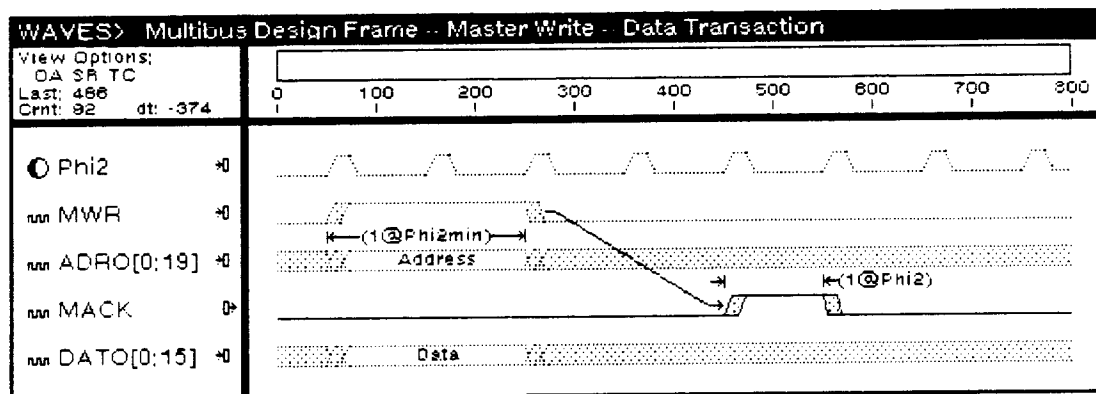


Figure B.9. The Multibus Design Frame Master Write operation.

Handshaking protocols are used to interlock the transfer of data between two circuit blocks. Two commonly used protocols in delay-insensitive or self-timed logic are the 2-phase and 4-phase protocols [Molnar85, Sproull86]. They are distinguished by the fact that the 2-phase protocol is sensitive to signal transitions while the 4-phase protocol is sensitive to signal levels.

2-Phase Protocol

B.3.1

The 2-phase protocol initiates an operation whenever there is a change of logic level (a transition) on its request line (see Figure B.10). The transfer is acknowledged by a change of state on the acknowledge line at some later time. This can be represented in *Waves* by the two diagrams above. One diagram represents the operation that begins when the Req line goes high and the other diagram when Req goes low. To ensure that two operations are not concurrent, Boolean conditions are attached to the two events on the Req line. In the first diagram, this condition is that Ack is low ((NOT Ack)) (i.e., the previous operation has been acknowledged and no operation is pending). In the second diagram, the corresponding condition is that Ack is high.

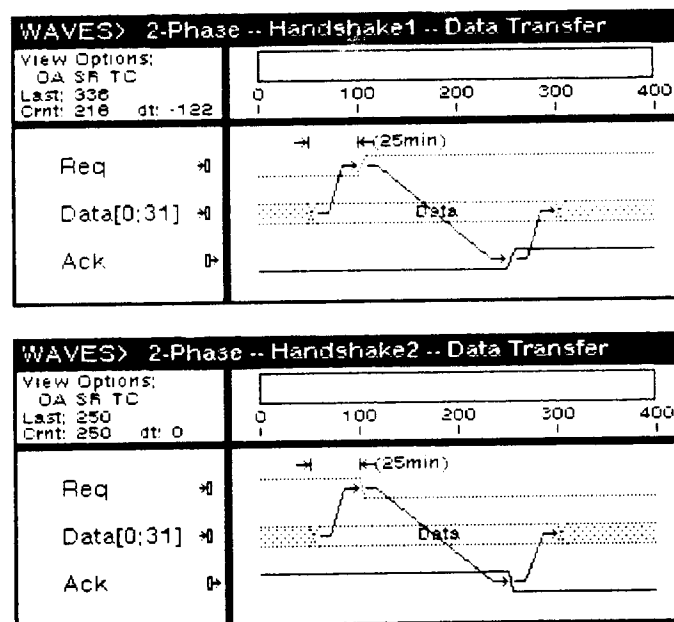


Figure B.10. The 2-Phase Protocol.

The minimum timing constraint between the assertion of data on the Data lines and the Req transition indicates that Req may be used to latch the data if this should be required and a latch is available with an adequate setup time (i.e., less than the value of the timing constraint).

4-Phase Protocol

B.3.2

The 4-phase protocol is very similar to the 2-phase except that the start of an operation is signalled by Req being high (see Figure B.11). Both Req and Ack must return to a logic low level before another operation may begin. This wastes some time compared to the 2-phase protocol but usually results in simpler interface circuitry (level, rather than transition, sensitive). In Appendix D, *Janus* will be used to synthesize an adapter circuit that converts the 2-phase protocol to 4-phase.

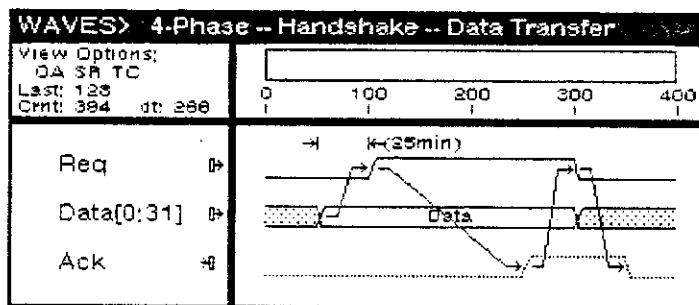


Figure B.11. The 4-Phase Protocol.

SPUR is a VLSI multiprocessor workstation [Hill86]. Each CPU includes a sophisticated cache controller that implements a snooping bus cache coherency protocol directly in hardware. The cache controller is composed of two parts: a processor cache controller and a snooping bus controller. The processor cache controller performs read and write requests made by the CPU, usually finding the data item resident in the cache. A cache miss or flush operation requires data to be transferred between the CPU and main memory over the system bus. The snooping bus controller (SBC) services these operations and monitors data transfers occurring on the system bus between other CPUs and main memory. The SBC services some bus requests in the place of main memory when it detects that its cache holds the most recently modified contents of a particular memory location.

The two components communicate with each other via handshaking operations that guarantee that only one unit will access the single data cache at any one time. This handshaking circuit is the SPUR PCC-SBC Interface [Gibson86]. There are two basic operations supported by this circuit: a PCC request for data to be sent or retrieved from main memory by the SBC and an SBC request for use of the processor caches to service another CPU's request for data.

Rather than describing each interface separately, this section groups diagrams for the same operation together in the same subsection. This is for explanatory purposes only.

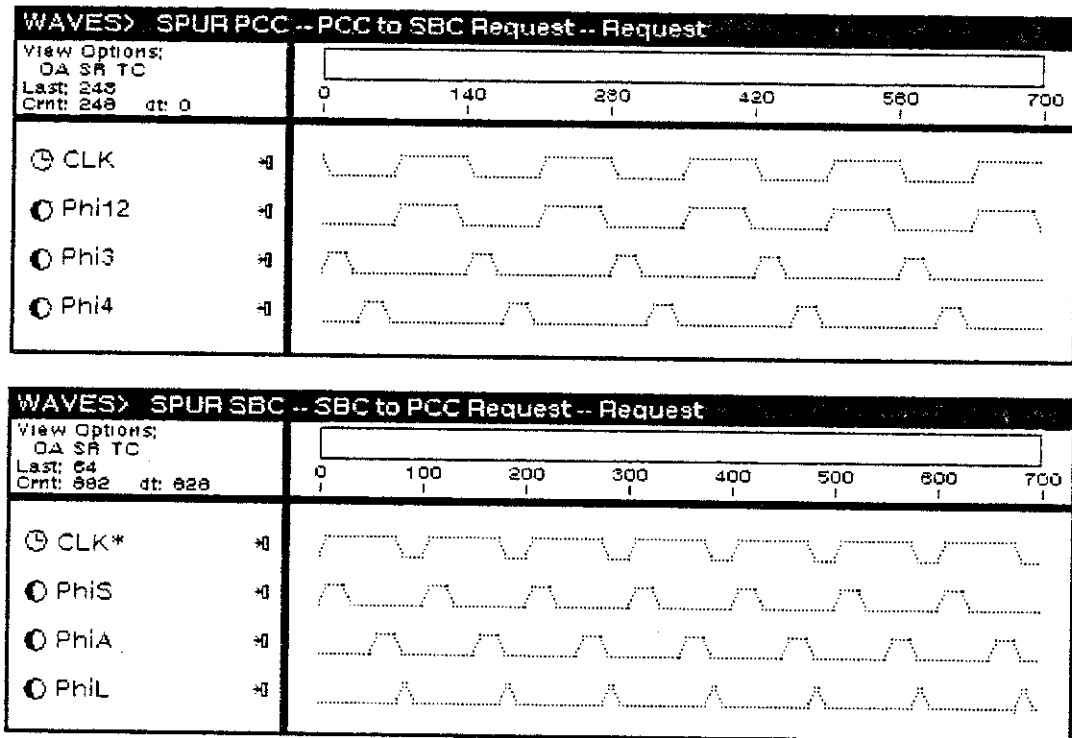


Figure B.12. SPUR PCC and SBC clocks and sub-phase relationships.

The two timing diagrams of Figure B.12 specify the relationship of clock phases in the two components. They are shown once here so as not to clutter the *Waves* diagrams describing the interface operations. The two clocks are completely asynchronous and have different sub-phase relationships.

PCC to SBC Request

B.4.2

The PCC issues a request to the SBC (PCCReq in the top diagram) along with a code for the type of operation it is requesting (PCCReqC[0:3]) (see Figure B.13). The SBC performs the operation and returns a code (SBCAckC[0:3]) with its acknowledge pulse.

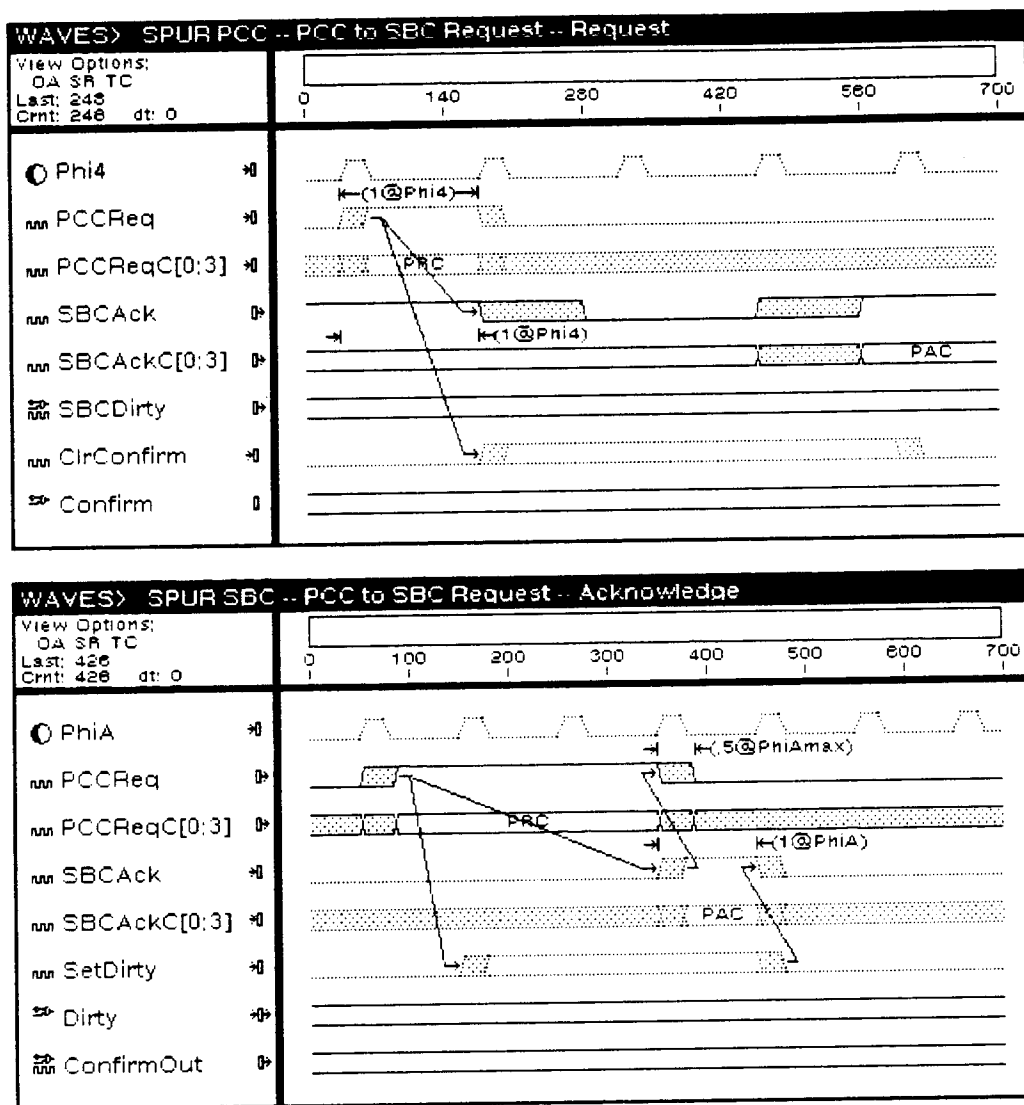


Figure B.13. The SPUR PCC-to-SBC Request operation.

Two signals complicate this operation. The PCC can cancel a request by asserting the ClrConfirm signal. The Confirm signal is used to implement this protocol. It is a computed signal whose function is to latch the occurrence of PCCReq and then optionally clear it if ClrConfirm is asserted during the operation ($\text{Confirm} = (\text{SR PCCReq ClrConfirm})$). The ConfirmOut signal to the SBC is another computed signal that simply adds the synchronicity constraints of the SBC interface to the internal Confirm signal ($\text{ConfirmOut} = \text{Confirm}$). The other complication arises from the SBC which may assert its SetDirty line to indicate a data status for the operation. This would normally be a part of the acknowledge code but the signal is transitory and may occur before the acknowledge pulse is ready. Therefore it, too, is latched by a computed signal ($\text{Dirty} = (\text{SR SetDirty PCCReq})$) and then transferred to the PCC interface via a second computed signal ($\text{SBCDirty} = \text{Dirty}$).

An SBC request is similar to those of the PCC except for some values of timing constraints (see Figure B.14). Another difference arises because the SBC also forwards interrupts directed to its CPU. Therefore, the request code is composed of two parts: a request code (SBCReqC[0:2]) and an interrupt type number (SBCInum[0:3]). There are separate latch signals for the two codes (SBCReq and SBCInumL) because the interrupt code may no longer be available when the SBC is ready to issue the request one cycle after the interrupt occurs. There are no special signals that need to be latched as in the case of the PCC request.

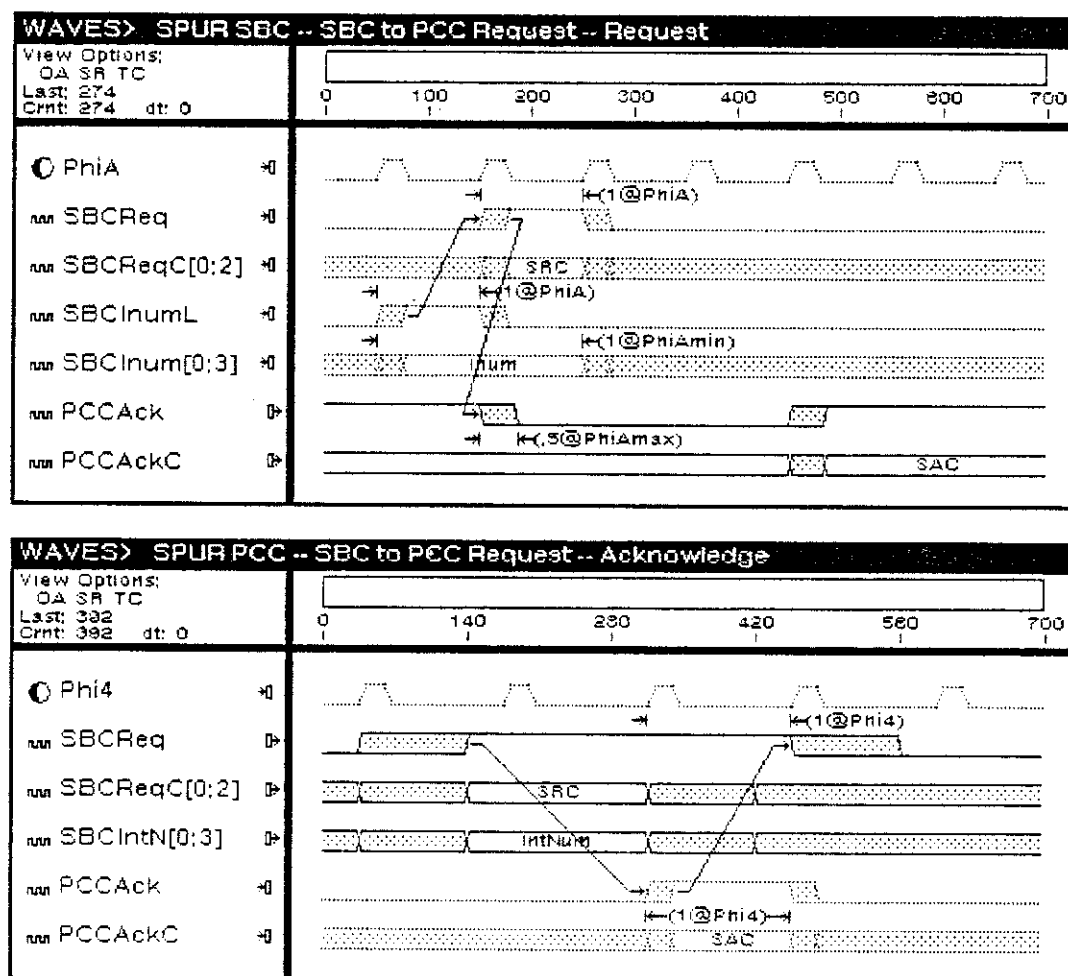


Figure B.14. The SPUR SBC-to-PCC Request operation.

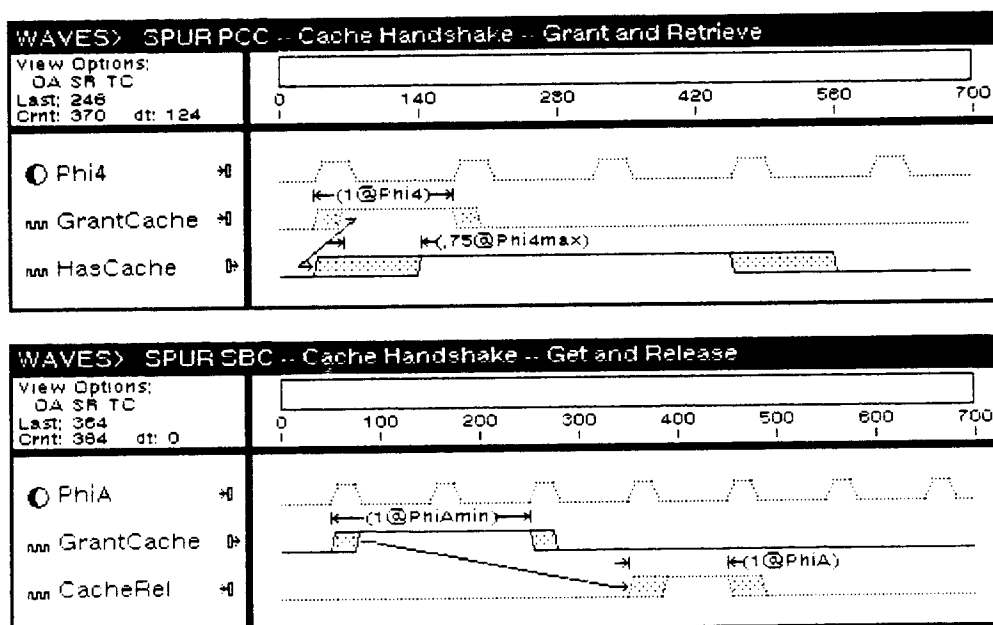


Figure B.15. The SPUR PCC-SBC Cache Handshaking operation.

The cache handshaking protocol is really part of the SBC request operation (see Figure B.15). However, since it uses a completely separate set of signals, it is represented here as a distinct interface operation. Since there is no data transferred in these diagrams, two ordering labels are used to interconnect the event sequences of the two diagrams. The labels exist between the first events on the GrantCache signals and from the first event on the CacheRel signal of the SBC to the second event of the HasCache signal of the PCC. Furthermore, a constraint also exists between these last two events. The time between CacheRel going high to HasCache going low must be at least two cycles of Phi4. This is implemented via a delay condition on HasCache ($(\text{DELAY CacheRel } 2 @ \Phi 4 \text{ NIL})$, delay a minimum of 2 cycles with no maximum constraint).

The Texas Instruments NuBus is a higher performance bus than the Intel Multibus [TexasInstruments85]. It is the system bus of several commercial workstations and the Berkeley SPUR workstation [Hill86]. Unlike the Multibus, both arbitration and data transactions are performed synchronously. All interrupt operations are memory mapped and fall into the category of data read and write operations. This section consists of three parts. The first specifies the arbitration protocol and the other two describe the master data operations. Of course, the arbitration specification is used together with both the read and write specifications.

Arbitration

B.5.1

The Nubus has a straightforward arbitration algorithm that utilizes a four-bit identification/priority value for each of the masters on the bus. The diagrams of Figure B.16 are two views of a single diagram that includes a conditional branch and a looping segment. The operation begins by asserting the RQST* line only if RQST* is unasserted in the previous cycle (a Boolean condition attached to the event on RQST* in the first diagram segment). The bidirectional ARB[0:3] lines are asserted on the previous edge of the system clock. The value of these lines is determined from a Boolean equation provided in the NuBus specification and is based on the value of the ID[0:3]* lines. The functions are shown below:

$$\begin{aligned} \text{ARB0*} &= (\text{NAND}(\text{NOT ID0*})(\text{OR}(\text{NOT ID3*})\text{ARB3*})(\text{OR}(\text{NOT ID2*})\text{ARB2*}) \\ &\quad (\text{OR}(\text{NOT ID1*})\text{ARB1*})), \\ \text{ARB1*} &= (\text{NAND}(\text{NOT ID1*})(\text{OR}(\text{NOT ID3*})\text{ARB3*})(\text{OR}(\text{NOT ID2*})\text{ARB2*})), \\ \text{ARB2*} &= (\text{NAND}(\text{NOT ID2*})(\text{OR}(\text{NOT ID3*})\text{ARB3*})), \\ &\quad \text{and ARB3*} = \text{ID3*}. \end{aligned}$$

The next event is based on the value of the ARB[0:3] lines 2 cycles after they are asserted. If they are equal, then the *Get Bus Immediately* segment is enabled, otherwise the *Wait for Others* segment is. The first corresponds to the master winning the arbitration contest and beginning its bus transaction and the other corresponds to another master winning and the requesting CPU having to wait for the other master's transaction to complete before it can again test to see if it has won. The enabling events for the *Get Bus Immediately* segment is the event on RQST* together with the 2@CLK* (2 cycles of CLK*) constraint. This reads as follows: RQST* can be deasserted if, after 2 cycles of CLK* from the assertion of RQST*, the ARB[0:3]* lines are equal to the ID[0:3]* lines. Another master will assert START* (and then ACK*) if they are not equal at that time. The master continues to assert request and the ARB[0:3]* lines until the other master has completed the transaction and a new decision is made. This method ensures that each master will get serviced in priority order and, because

RQST* cannot be asserted when other masters are already involved in an arbitration contest, no master will be starved.

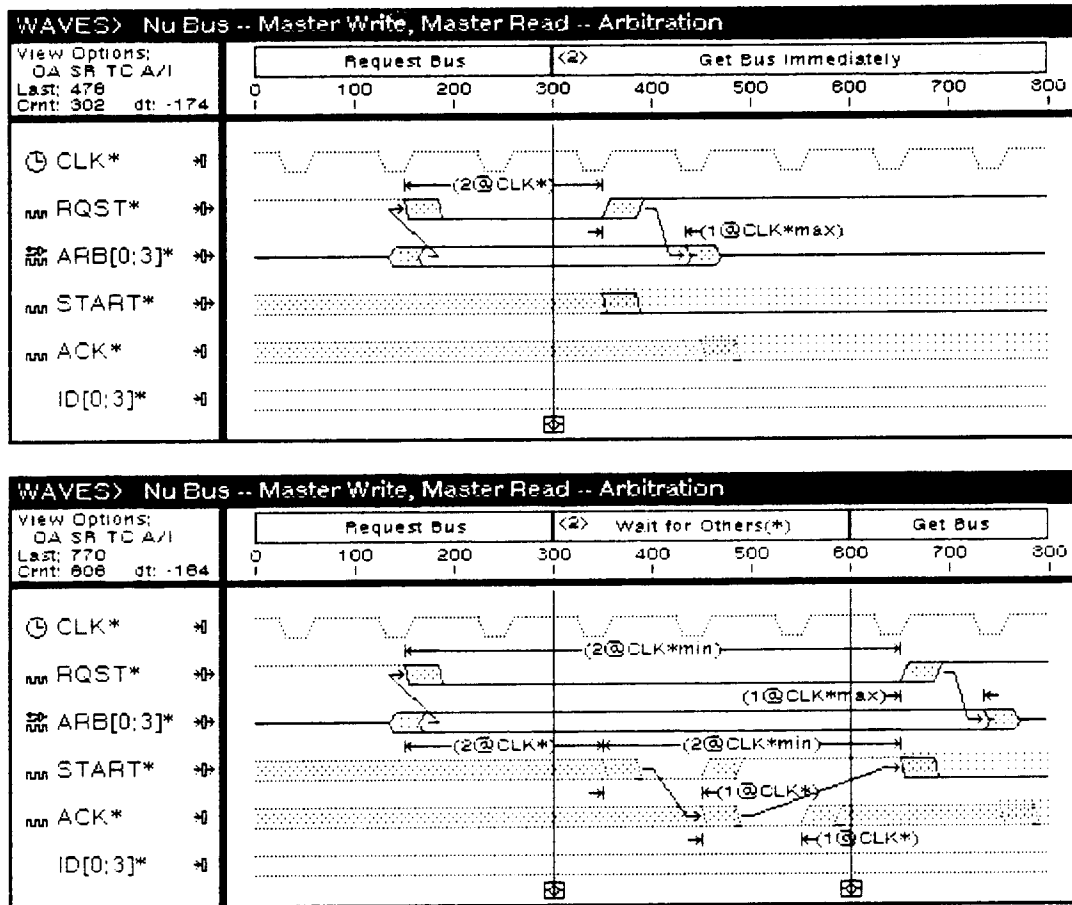


Figure B.16. The NuBus arbitration sequence (two views of the same diagram showing different segments).

When the master finally gets its turn, it deasserts the RQST* and ARB[0:3]* lines and begins one of its transactions. These are represented by the diagrams of the next two sections. They are merged with the arbitration diagram at the two events on START* and ACK* (both shaded) in the *Get Bus Immediately* and *Get Bus* segments. These events are used as place holders for ordering and timing constraints between the diagrams.

Master Read

B.5.2

A master read operation consists of events defining two basic cycles (see Figure B.17). The first cycle specifies the operation (read or write) and address and another cycle returns the data and status code to the requesting master. Both TM1* and TM0* high during the START*

pulse signify to the slave that the operation is a read. For these diagrams and those of the next section, all events occurring at the same clock edge are related by a simultaneity relation.

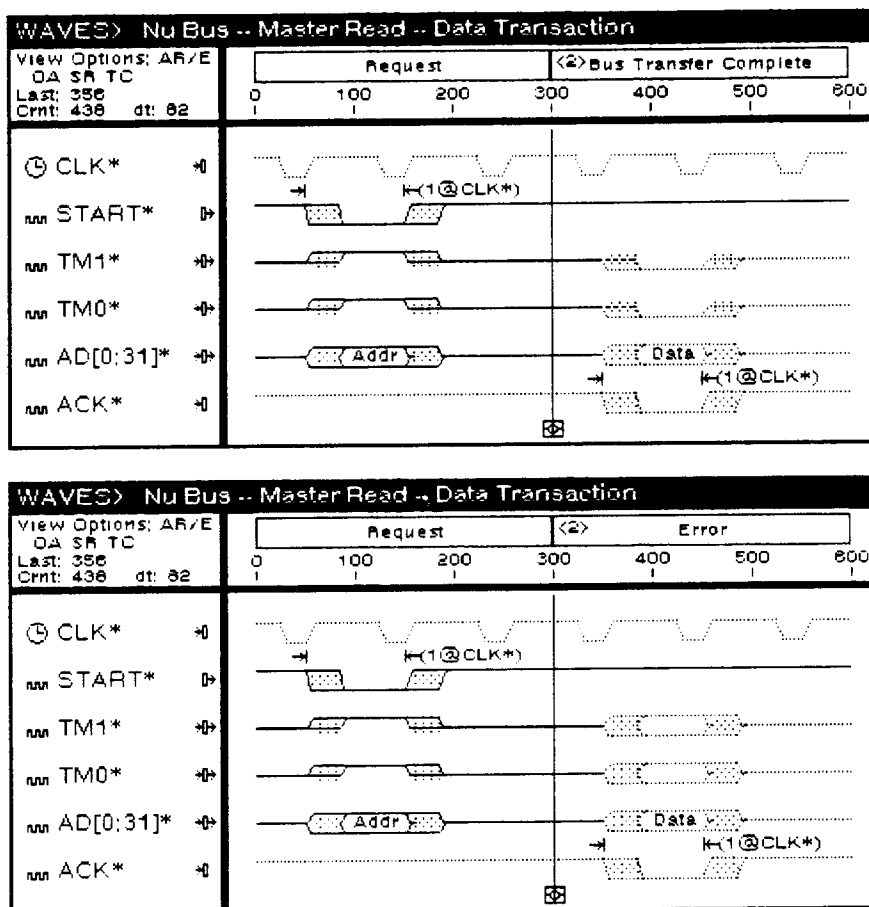


Figure B.17. The NuBus Master Read operation (two views of the same diagram showing different segments). This diagram is combined with that of Figure B.16.

Sometime later, the slave will respond with an ACK* pulse and values of TM1* and TM0* corresponding to an operation status code (timeout, error, etc.). The operation may end correctly (top diagram) or result in an error (bottom diagram). The enabling event for the two segments is the first event on ACK* and the two are distinguished by whether TM1* and TM0* are both low or at least one is high when the ACK* event occurs.

These diagrams demonstrate the specification of a multiplexed data line (AD[0:31]* carries both Addr and Data) and conditional ending of an operation. This diagram may be linked to another interface that also distinguishes between these two conditions and has a conditional ending for its semantically equivalent operation or to one that does not distinguish between the two and performs the same sequence of events regardless of the status code. In the former case, this is implemented with two ordering labels on different events in mutually exclusive segments. In the latter case, the two ordering labels will be on the same event.

Master Write

B.5.3

The data write operation is identical to the data read except that TM1* is asserted low (to signal a write to the slave) during the START* pulse and the data to be written is available on the AD[0:31]* lines immediately after the address and until the ACK* pulse is received (see Figure B.18).

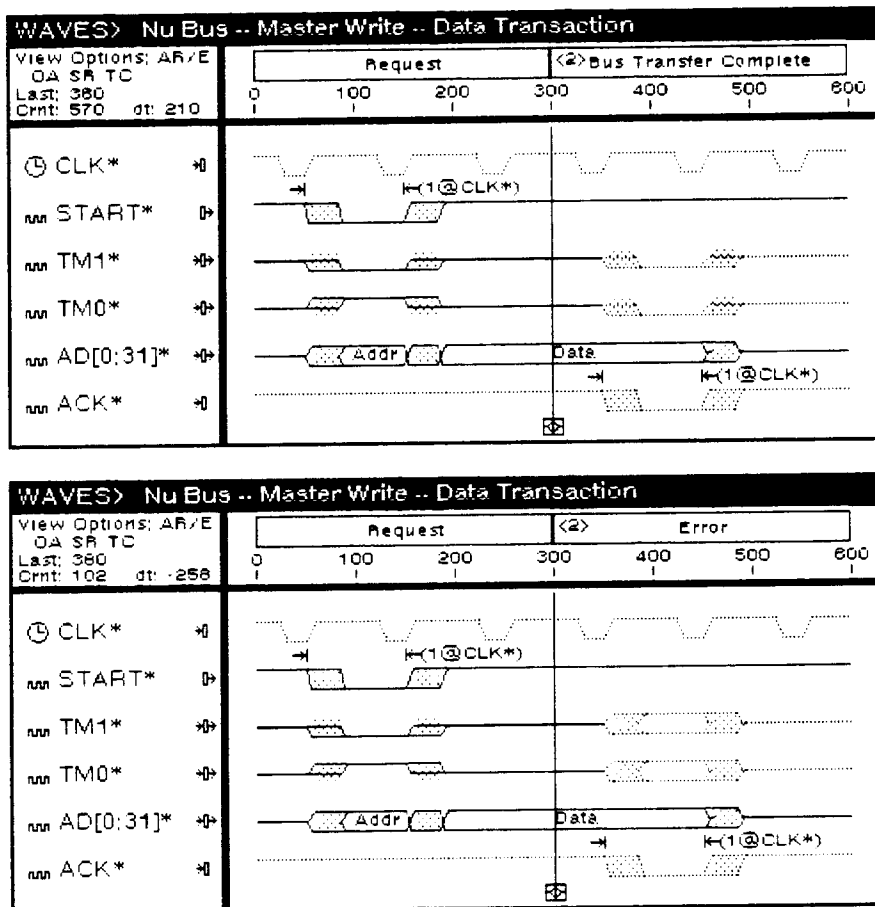


Figure B.18. The NuBus Master Write operation (two views of the same diagram showing different segments). This diagram is combined with that of Figure B.16.

The Motorola 68000 is a widely used microprocessor [Motorola81]. This section includes *Waves* diagrams for the memory interface of this processor. The microprocessor interface signals are all synchronous to the system clock (CLK) except for the data lines that have setup and hold time requirements relative to the acknowledge line (DTACK/). The signals of the interface include signals for: address (A[1:23]), data (D[0:15]), processor status (FC[0:2]), read/write select (RW/), address and data strobes (AS/, DS/), and an acknowledge (DTACK/). The specifications below are for a 10MHz 68000.

Read

B.6.1

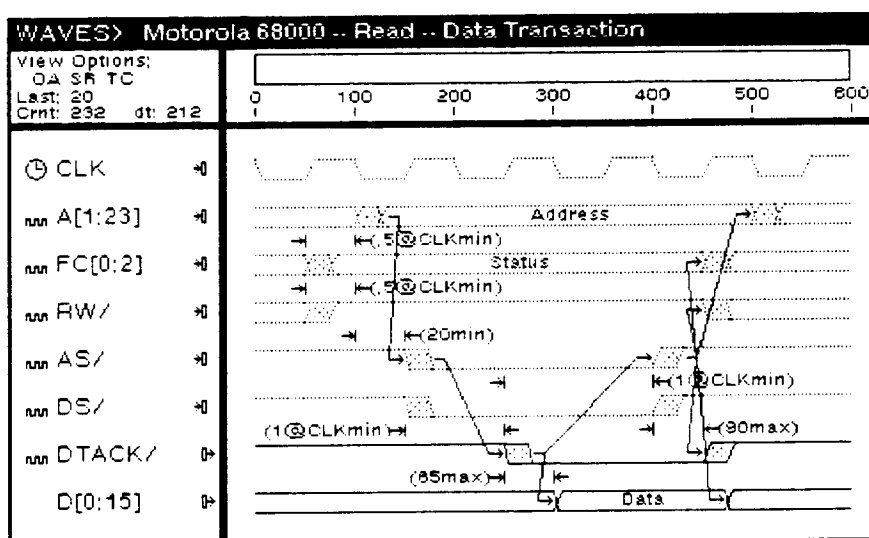


Figure B.19. The Motorola 68000 Read operation.

The read operation uses a minimum of four clock cycles to complete. This can be seen by adding up the constraints between the events of the sequence (see Figure B.19). A special point to notice is that the acknowledge line can be asserted by the responding circuit (e.g., memory) before the data is actually available, but no more than 65ns earlier. This constraint ensures that data is always valid in the cycle after the acknowledge is seen. The assertion of the address strobe (AS/) signals the start of the operation.

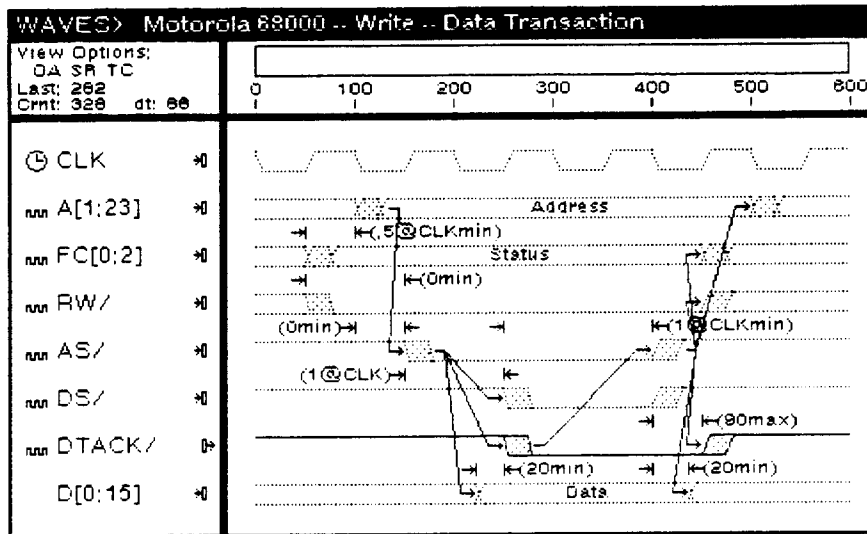


Figure B.20. The Motorola 68000 Write operation.

The write is similar to the read except in the case of the acknowledge and data lines (see Figure B.20). The data has 20ns setup and hold times relative to the data strobe (DS/) and the data strobe occurs exactly one cycle after the address strobe. The acknowledge signal may be asserted immediately by a fast memory or much later by a slow one. There is no constraint on the precise position in time of the acknowledge pulse, only that its leading edge occurs while the address strobe is asserted.

Static RAM memories are commonly connected to custom and off-the-shelf components. The Lattice Logic SR64K4 high-speed static memory chips are typical memories of this type [Lattice84]. In the diagrams below, the interface of the SR64K4-35 (the 35ns access time version of this class of memories) is specified.

Read

B.7.1

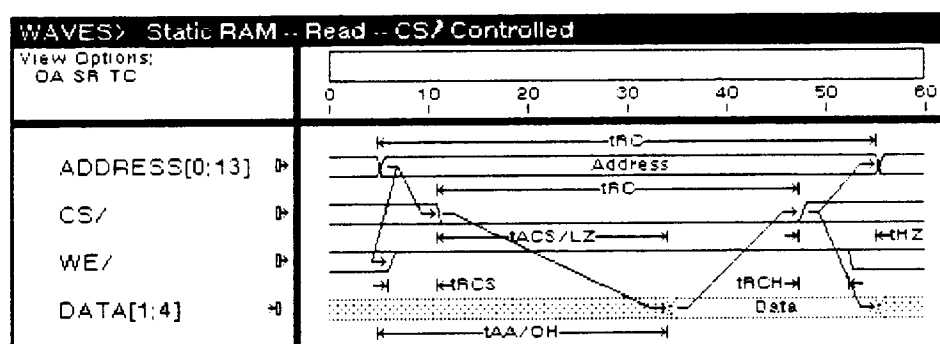


Figure B.21. The Lattice Logic SR64K4-35 Read operation.

The overall constraint on the read operation is that it take 35ns to complete (the access time of the chips). This is reflected in the constraints labelled t_{RC} (a minimum), t_{AA} (a maximum), and t_{ACS} (a maximum access time) (see Figure B.21). There are also setup and hold time requirements on the write enable signal ($WE/$) relative to the chip select line ($CS/$). These constraints are labelled t_{RCS} and t_{RCH} and are both 5ns minimum. Other constraints include a minimum response time to asserting data (t_{LZ} , 10ns) and a maximum time for data to remain valid after the $CS/$ is deasserted (t_{HZ} , 20ns).

Write

B.7.2

As usual, the write operation is similar to the read operation (see Figure B.22). The overall constraints are now still present (t_{WC} , t_{CW} , and t_{AW}) and are all 35ns minimum. In addition, there is a minimum constraint on the width of the $WE/$ pulse (t_{WP} , 20ns), a setup time on data being valid before the write pulse ends (t_{DW} , 30ns), a maximum time on deasserting data (t_{OW} , 10ns) (before the memory may start driving the data lines), and a minimum time before data can be asserted on the memory inputs (t_{WZ} , 20ns) (the memory must be allowed to turn off its drivers first).

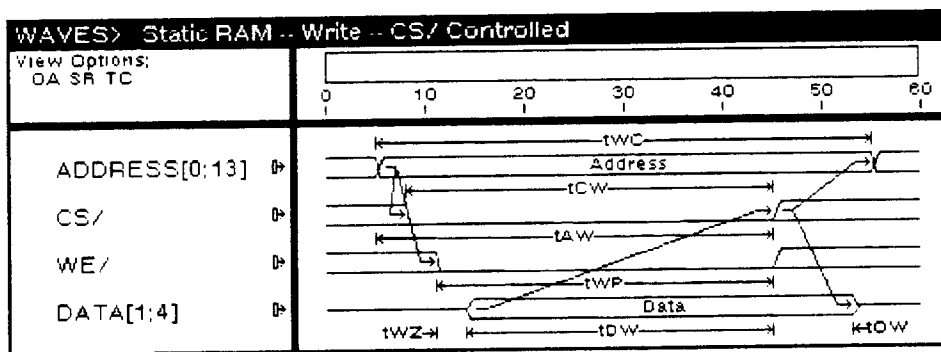


Figure B.22. The Lattice Logic SR64K4-35 Write operation.

Janus Implementation

C

Janus is a tool for the automatic synthesis of interface transducers. Given *Waves* timing diagrams describing the two interfaces to be connected, it generates the logic specification for a transducer. *Janus* is implemented in LOOPS, an object-oriented programming extension to the Interlisp-D programming environment, on Xerox 1109 workstations.

This appendix is composed of six sections. The first section provides a detailed description of the data structures used in *Janus*. The second section explains the restrictions on the input specifications. Section 3 describes an event graph browser that was developed as a debugging aid. The fourth section covers the *Janus* logic circuit library and how its collection of primitive elements is accessed. Section 5 outlines the procedure used to simulate and validate the output of *Janus* using the RNL simulator. The appendix concludes with a section on portability issues.

Janus is implemented in the same programming environment as *Waves*, Interlisp-D/LOOPS running on Xerox 1109 Lisp machines. Having both tools in the same address space avoids the need for generating and parsing intermediate files containing the information already present in the data structures. *Janus* generates its data structures by directly translating *Waves* diagrams. *Janus*' data structures for a particular transducer, like those of a *Waves* diagram, can be saved in a separate file. The object inheritance lattice for *Janus* is shown in Figure C.1.

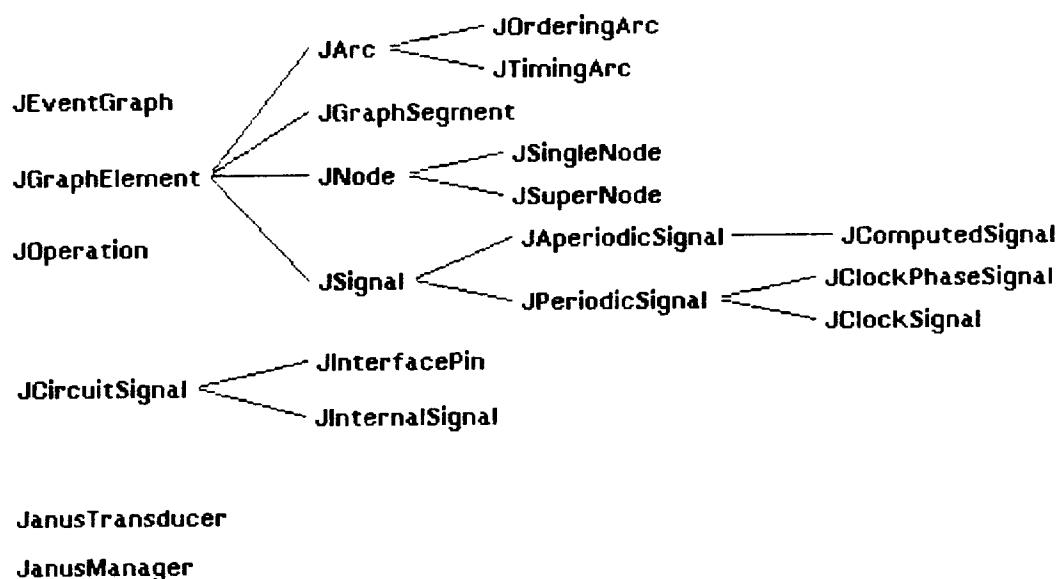


Figure C.1. The *Janus* object class inheritance lattice.

There are three principal groups of objects in *Janus*. The first consists of two object classes: *JanusManager* and *JanusTransducer*. The *JanusManager* displays itself as the *Janus* icon and holds pointers to all the transducers currently loaded in the address space and has an analogous function to the *WavesManager* object. An instance of *JanusTransducer* holds the specification of a transducer, namely, the two interfaces being connected and the operations supported by the transducer. It also contains pointers to the objects describing the details of each interface operation (*JOperation*). *JTransducer* is equipped with methods that combine the resulting circuits for each operation into a single optimized circuit. The *JOperation* object holds pointers to the *Waves* diagrams that specify the details of the operation and to the event graphs corresponding to the diagrams. This object manages the generation of the independent event graphs for the two sides of the transducer and interconnects them based on data transfer analysis and merging of events. It also holds the description of the circuit that will be used to control whether the operation is active (see section 5.4).

The second group of objects is used to represent the event graphs of each operation. The high-level object is *JEventGraph* and it holds pointers to all the nodes and arcs of the graph as well as each of the signals and segments defined in the diagrams. Nodes are either single events (*JSingleNode*) or multiple events related by simultaneity constraints or merge labels (*JSuperNode*). The common methods to both node types are in the *JNode* object. The two types of arcs are represented by *JOrderingArc* and *JTimingArc*. Again, common functionality is contained in the super class *JArc*. Nodes have pointers to all their incident and emanating arcs and arcs have pointers to their head and tail nodes. Nodes also contain a description of the circuit that will be used to generate their event(s). Timing arcs store minimum and maximum constraint values. No circuit information is held in arc objects. *JGraphSegment* is the precise analog of the *Waves* diagram segment (*WDiagramSegment*) and holds similar information. In addition, it includes a description of the circuitry that will be used to signal whether the segment is currently active (see Section 5.5). A segment holds a description of the circuitry that will be used to enable its events (to which it has pointers). Each node also has a pointer to its enabling segment. The last set of objects in this group represents the transducer signals which have a similar hierarchy to their *Waves* counterparts (see section A.7). The four basic signal types are represented by *JAperiodicSignal*, *JComputedSignal*, and the two subclasses of *JPeriodicSignal* (*JClockSignal* and *JClockPhaseSignal*). Their common methods and instance variables are in *JSignal*. These objects hold pointers to their events and are used to store information such as signal direction, quiescent levels, and synchronicity constraints. Furthermore, they hold a pointer to their corresponding signal in the circuit.

The third, and last, group of objects is used to represent the circuit realization. A circuit is composed of internal signals (*JInternalSignal*) and input/output pins (*JInterfacePin*). These contain pointers to identically named signals in the various operations of the transducer and the structure of the circuit that will be used to generate each output signal. Therefore, there is a many-to-one mapping between signals (*JSignal*) and circuit signals (*JCircuitSignal*). This is the top-level data structure used to represent the circuit (see section C.4). Interface pins correspond to input/output nets (e.g., input/output pads, in the case of a chip interface). Internal signals correspond to internal nets including segment and operation control signals.

Janus aids the designer in validating the input specifications. *Waves* supports a more general model of interface specification than is supported by *Janus*. For example, *Janus* requires that the logic value of each input and output signal is identical at the beginning and end of each interface operation. *Waves* imposes no such restriction on the signal waveforms that can be drawn. Other types of restrictions occur when diagrams are combined. *Waves* provides no checking facilities for merge operations while *Janus* must ensure that merges are well-formed.

There are quite a range of validity checks that *Janus* can make on the input specifications. Some may be only *warnings* (i.e., the specification can still be used to synthesize a circuit) while others are *errors* (i.e., *Janus* cannot continue). An example of an error is a cyclic data transfer. An example of a warning is a maximum timing constraint between two events that are eventually merged into a super-node. All of the checks are performed during the translation from *Waves* to *Janus* data structures after which the user can view a summary of the errors and warnings. The following partial list gives an idea of the types of checks performed:

- Constraints on periodic events, with the exception of synchronicity constraints, are acceptable.
- All signal waveforms must start and end at the same logic level in all diagrams where the signal is present.
- No simultaneity relations can exist between input and output events, either explicitly or as the result of merges.
- Place holder events for cross-diagram constraints must be consistent (i.e., the events to be merged represent the same logic transition).
- Signal directions and periodic signals must have consistent properties across diagrams (e.g., electrical parameters, clock periods and duty-cycles, etc.).
- Valid levels on all signals, except computed signals, must be labelled as data carriers.
- Events with constraints relating them to events on computed signals must include a Boolean condition for the logic value of the computed signal.
- The two sides of an operation specification must be consistent (e.g., at least one event with an empty interval of occurrence, a cyclic constraint due to data transfers, etc.).

A graphical event graph browser was developed as a debugging aid during the development of *Janus*. It is also useful in presenting to the user the result of translating a set of *Waves* diagrams into event graphs. An example browser, showing the event graph corresponding to the example specification of section 1.2.1, is shown in Figure C.2. The signals have already been split into their input, output, and enable components (see section 5.2.6).

The browser provides access to each of the nodes of the event graph and shows their interconnections using only the ordering arcs. Therefore, successor nodes appear below their predecessors. There is no implication of timing relationships in the separation between nodes in both the horizontal and vertical dimensions. Had the timing arcs also been included, the graph would have been much too cluttered to be a useful visualization.

The browser is a specialization of the LOOPS class hierarchy browser that is part of the Interlisp-D/LOOPS environment [Stefik86]. Menus of functions are available by pushing mouse buttons over the nodes. The menu items invoke various procedures on the selected node or the graph as a whole. For example, in debugging *Janus*' algorithms, node data structures can be inspected to ascertain whether modifications were made correctly. This is much easier and faster with a visual representation of the data structure as opposed to manually chasing pointers. Another example comes from experiences in developing the data-path generation algorithms. Here, the browser was used to run the interval of occurrence algorithm from a selected node and then view the resulting intervals on the other nodes.

The browser is also useful for the designer that has just completed specifying a design and would like another form of visual verification of the specification. For example, it is obvious an ordering or simultaneity constraint is missing when the structure of the graph is viewed directly. The specification can then be corrected and the diagrams retranslated.

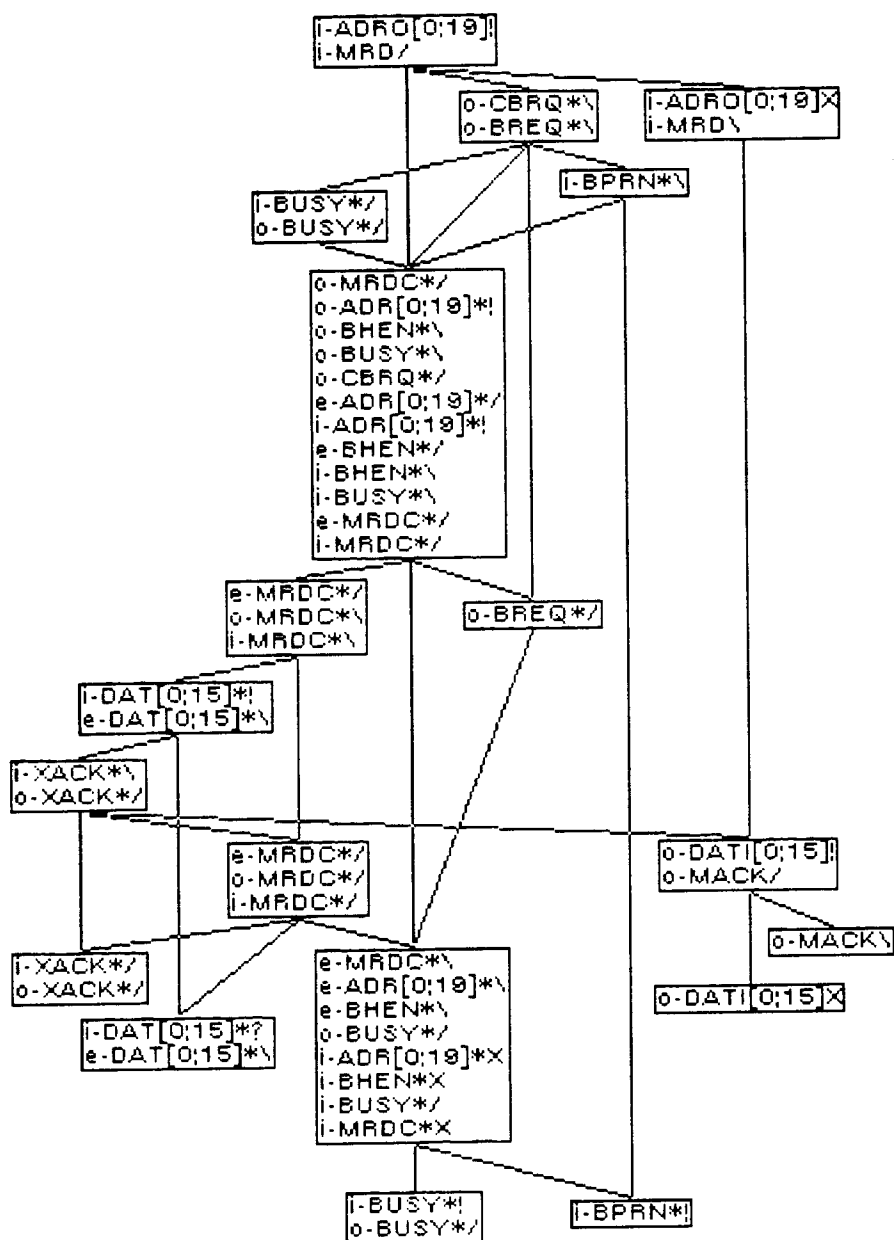


Figure C.2. An example of a Janus event graph browser. The arcs in the browser correspond to the ordering arcs of the event graph. Signals have already been split into their input, output, and enable components (the prefixes i-, o-, and e- prefixes, respectively). The suffixes /, \, X, !, and ? correspond to the change in logic level represented by the node (rising, falling, don't care, valid, and tri-state, respectively).

The *Janus* circuit library supports four types of queries. The first accesses technology dependent parameters, the second returns the name of a primitive element, the third determines the propagation delay of a subcircuit, and the last generates textual output describing the subcircuit to a file. These four queries (or procedures) insulate *Janus* from the details of a specific circuit library. *Janus* assumes the existence of a limited set of primitive elements and is not concerned with the details of their implementation.

The first procedure takes the form:

(*JLibraryParameter parameterName*).

It is used whenever *Janus* requires knowledge of a technology dependent parameter. Currently, the only parameter supported is latch setup time (*latchSetupTime*).

The second procedure takes the form:

(*JLibraryGet primitiveType inputList*).

It is used whenever *Janus* requires a primitive element. The primitive type is one of: AND, NAND, OR, NOR, NOT, SR, D-FF, SYNC, LATCH, DELAY, MUX, and PAD. For the Boolean operators, the *inputList* is simply a list of the inputs to the element. SR has only two inputs, set and reset. D-FF, SYNC, and LATCH are different in that their input list has only two elements (input and control). For example, for a synchronizer the input list is (*input clockToSynchronizeTo*), for a D-type flip-flop it is (*input clock*) and for a latch it is (*input control*). For a DELAY the *inputList* is (*input minDelay maxDelay*). For a MUX the *inputList* must be of the form (*defaultInput (input1 selector1) ... (inputN selectorN)*). For a PAD the *inputList* is (*type output enable*) where *type* is one of Input, Output, OCOOutput, TriInputOutput, or OCInputOutput and *output* and *enable* are the two possible inputs to the pad (i.e., an input pad has NIL for both *output* and *enable*). The *JLibraryGet* function returns a form for the primitive element and its inputs, that is, the name of the primitive circuit element appended to the front of the input list (e.g., (*ANDgate input1 input2 ... inputN*)). This model supports the selection of different primitives depending on the names of the inputs. For example, the specifier of the library may want to choose a two-state or one-stage synchronizer depending on the clock to which the signal is being synchronized or different types of delay elements depending on the amount of delay required.

The third procedure takes the form:

(*JLibraryDelay circuitForm input*).

It is used to calculate the delay of a path in the circuit from an *input* through the specified circuit (*circuitForm*). The *circuitForm* is an S-expression constructed using *JLibraryGet*. The value returned is a pair of delays of the form (*minDelay . maxDelay*). These represent the minimum and maximum delay from the *input* to the top-level of *circuitForm*. *JLibraryDelay* calls itself recursively while traversing *circuitForm* in a depth-first search. The search ends

when the function encounters an element of the list *circuitForm* that is equal to *input*. If *input* is never matched then it is not an input to the circuit described by *circuitForm* and the returned value is (*NIL* . *NIL*). This format permits the determination of delay along any circuit path. The minimum and maximum propagation delay from all inputs of a subcircuit is returned when *input* is *NIL*.

Finally, the last library procedure takes the form:

(*JLibraryOutput* object *circuitForm* *outputFile*).

It is used to write an RNL description of *circuitForm* onto the *outputFile*. The *Janus* object calling the function is recorded and a net name is created to correspond to that object. The name will be used to identify the output net of *circuitForm*. Any other *Janus* objects encountered in the traversal of *circuitForm* are also assigned a name. This list of object-net correspondences ensures that the subcircuits output onto the file will be properly interconnected within the simulator and by implementation tools. When *circuitForm* is *NIL*, the RNL macro definitions of the library are output. Therefore, (*JLibraryOutput* *NIL* *outputFile*) is called first, before any other circuit structures are written to the file.

Janus generates a description of the synthesized circuit via the library described in the previous section. The output file includes a list of all the circuit primitives used and a net list describing their interconnections. A gate level description of each primitive element is also included (also obtained from the library). This is done to ensure that the design will be completely specified within a single file. This makes *Janus* output more easily portable to other tools that may not recognize the same circuit libraries.

The current format of the file is identical to the input format for the RNL simulator [Terman87]. RNL is used to simulate the circuits synthesized by *Janus* and validate the synthesis process.

There are two reasons for this choice. The first reason is that RNL has a macro capability with a Lisp-like syntax. The *Janus* circuit library includes RNL macros for each of the primitive elements. For example, an S-R* latch is specified as:

```
(macro sr (s r q)           ; name of macro and i/o signals
  (local qbar)              ; internal nodes
  (cnor qbar s q)           ; one of the two CMOS NOR gates
  (cnor q r qbar)           ; the other cross-coupled gate
)                             ; end of macro
```

Most logic level simulators require that circuits be input in a flattened description that only contains primitive gates. In some extreme cases, a flattened transistor level description is the only allowable input format.

The second reason is that RNL's event-driven simulation engine includes a model of timing delay. To model propagation delay of circuit blocks and delay elements all that is required is that a node specification includes the amount by which to delay all simulation events that occur on that node (e.g., (delay x 10 10) means that both rising and falling transitions on node x are delayed by 10 time units).

Portability Issues

Waves and *Janus* are implemented in the same single address space programming environment. However, their data structures are completely independent due to issues of portability to other environments. Obviously, of the two, *Waves* is more difficult to port due its heavy use of user interface and graphics primitives. *Janus* has no such emphasis. The event graph browser may fall into this category, but since it builds on a programming tool, its functionality is likely to be present in mature object-oriented programming environments. The algorithms in *Janus* are written in a subset of Interlisp-D and LOOPS that should make them automatically translatable to Common Lisp and CLOS [Bobrow88].

As with *Waves*, *Janus* will need to interact with a CAD database such as OCT to retrieve interface specifications and store descriptions of the synthesized circuits [OCT87]. This capability should further improve the level of interconnection between *Waves* and *Janus* by selectively loading and storing objects from both programs when they are referenced, rather than requiring them to be simultaneously resident in virtual memory. A practical application of this is in providing the user with more direct feedback during specification consistency checking. *Janus* could point directly to the *Waves* diagram objects that are causing difficulty rather than providing just a textual description of the errors and warnings (see section C.2).

Janus Synthesis Examples

D

In this appendix, *Janus* is applied to three transducer synthesis problems. The first example is the Multibus Design Frame, a mixed synchronous and asynchronous design. The second example is fully asynchronous design, an adapter between 2-phase and 4-phase handshaking protocols. The third example is the SPUR PCC-SBC Interface which interconnects two synchronous subsystems that have asynchronous clocks. The *Waves* specifications for each of the six interfaces (two for each example) can be found in Appendix B.

This appendix is divided into three sections corresponding to each of the three examples. Each section is further divided into three subsections. The first subsection shows the event graphs generated by *Janus* from the *Waves* diagrams. The second subsection details two circuits for each example: the first a manual design and the second synthesized by *Janus*. The last subsection concludes with a summary and comparison of the two designs.

The Multibus Design Frame connects a simple synchronous interface to the Intel Multibus [Borriello85]. Four operations are implemented by this transducer: slave read, slave write, master read, and master write. The *Waves* specifications for each operation can be found in sections B.1 and B.2.

The only modifications made to the specification are to include ordering labels on the two write operations that prevent the output acknowledge signals from being asserted until the input acknowledges are detected. These are from the falling edge of XACK* in Figure B.2 to the rising edge of SACK in Figure B.7 and from the rising edge of SACK in Figure B.9 to the falling edge of XACK* in Figure B.5. This is not necessary for the read operations because the data dependencies already enforce a similar constraint.

Event Graphs

D.1.1

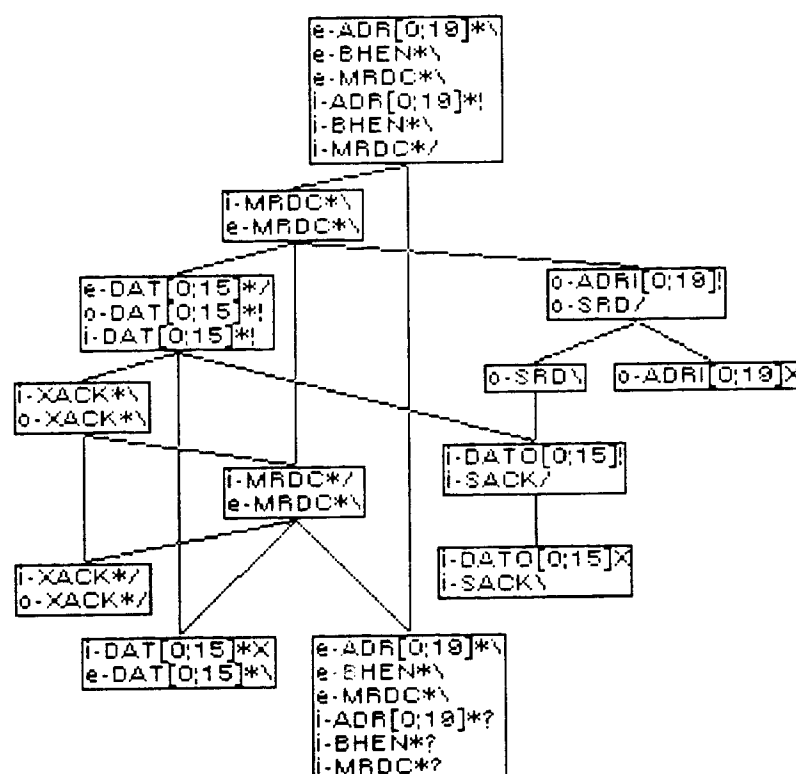


Figure D.1. Event graph for the Slave Read operation of the Multibus Design Frame.

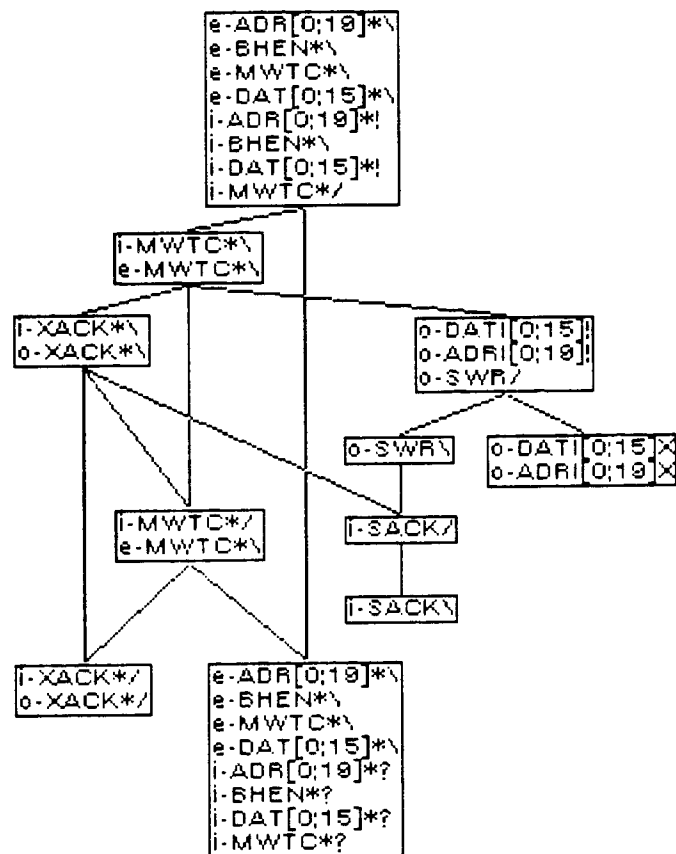


Figure D.2. Event graph for the Slave Write operation of the Multibus Design Frame.



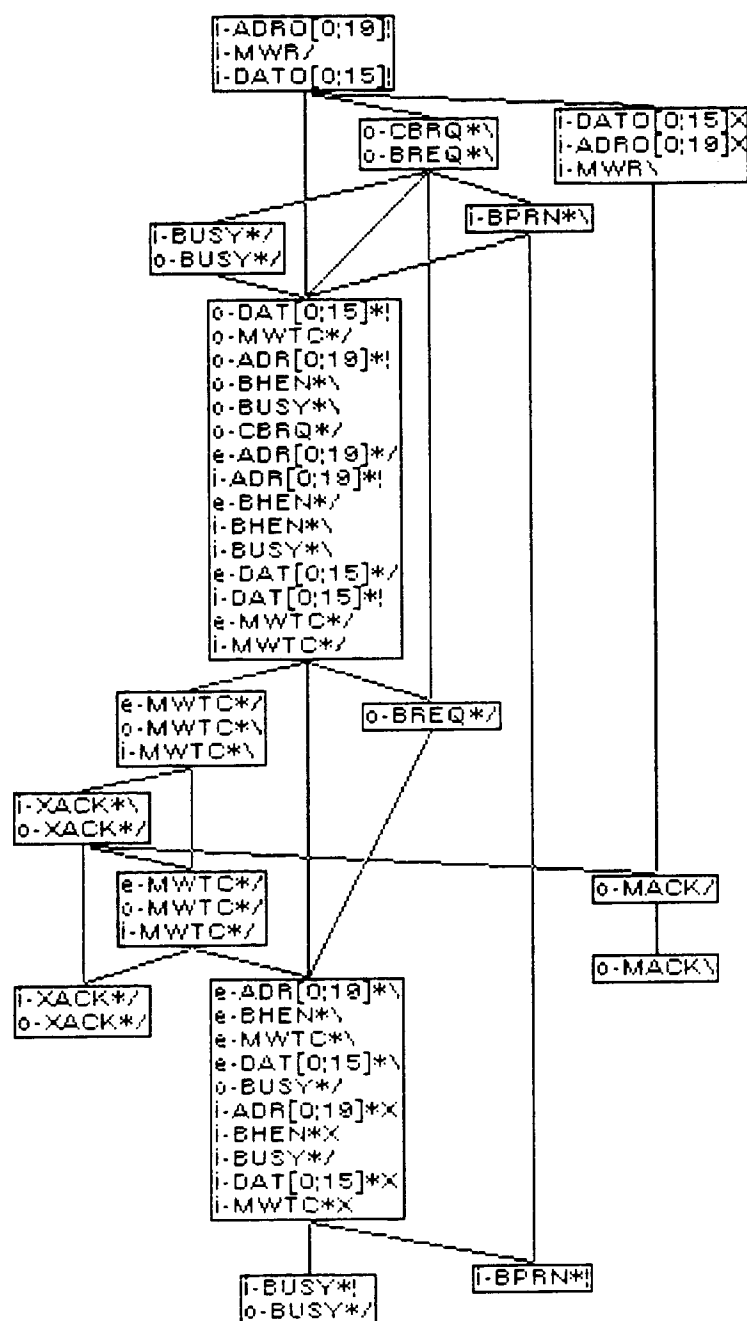


Figure D.4. Event graph for the Master Write operation of the Multibus Design Frame.

The manually generated design for the Multibus Design Frame is presented in two parts (see Figures D.5 and D.6). The design generated by *Janus* is presented in four parts (Figures D.7 through D.10). The *Janus* design, as shown, has circuitry for each operation combined into a single circuit but has not been further optimized. For example, circuitry is clearly duplicated in many cases and some possible transformations to combine flip-flops and synchronizers are not shown.

In all the figures of this section, the Multibus Design Frame interface is on the left and the Multibus is on the right. Only signals that connect to one of the two boundaries are part of the interface. Internal signals are not connected to either side. In the *Janus* circuit, signal names in brackets correspond to the operation enable signals generated for each interface operation. When these signals appear under an S-R* latch it signifies that the latch is reset whenever the operation enable signal is low.

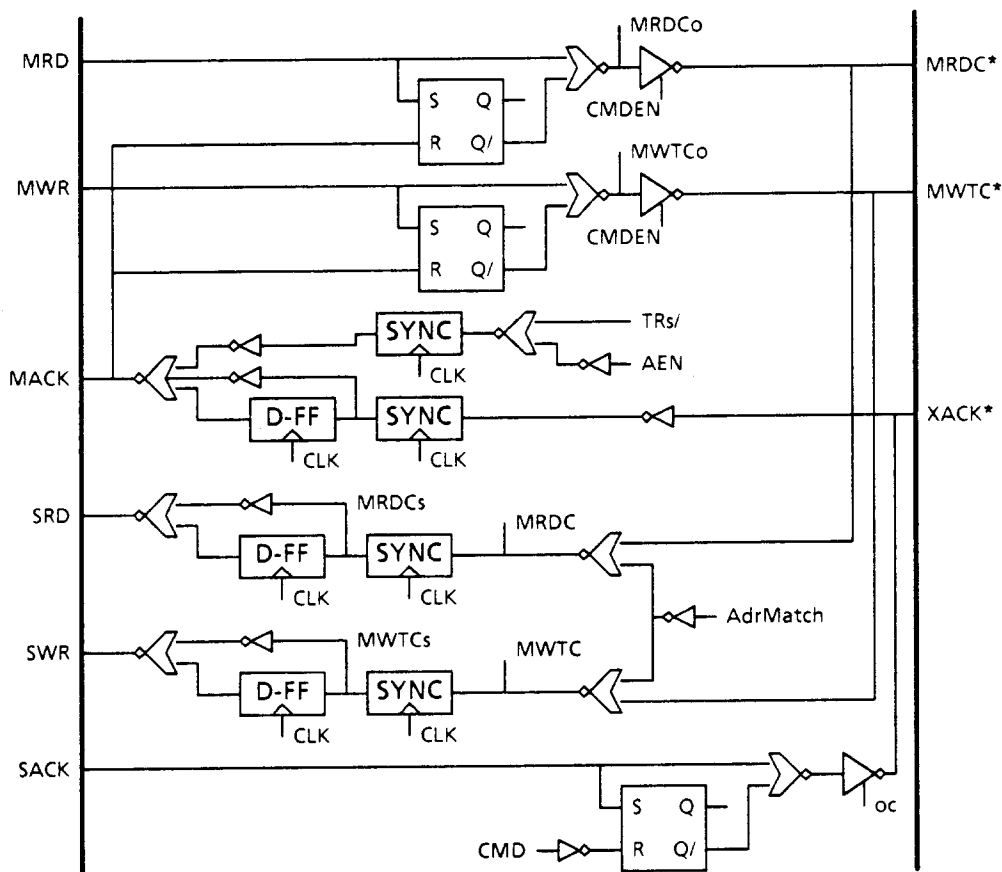


Figure D.5. Manually designed circuit for the Multibus Design Frame (part one of two, see Figure D.6) [Borriello85].

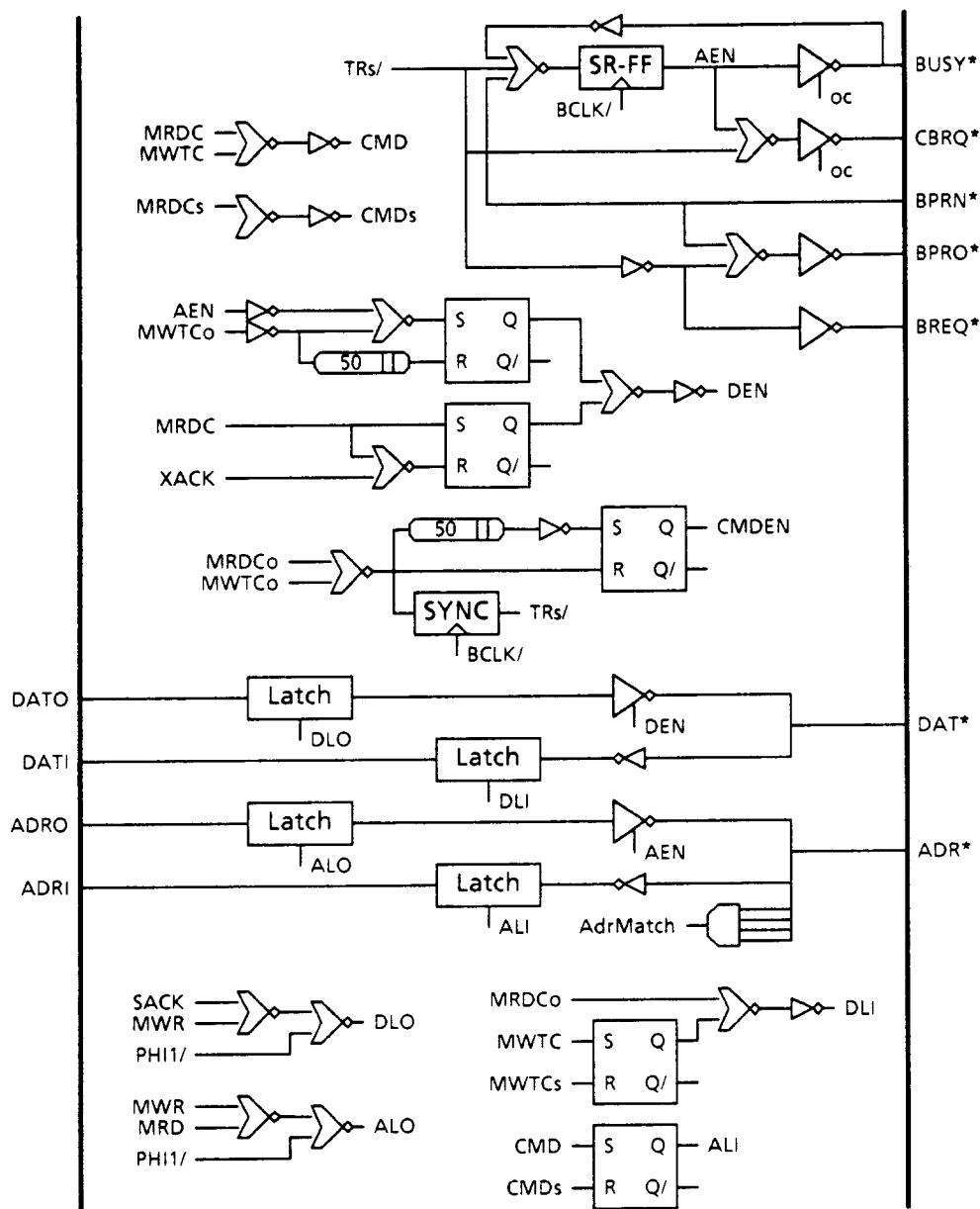


Figure D.6. Manually designed circuit for the Multibus Design Frame (part two of two, see Figure D.5) [Borriello85].

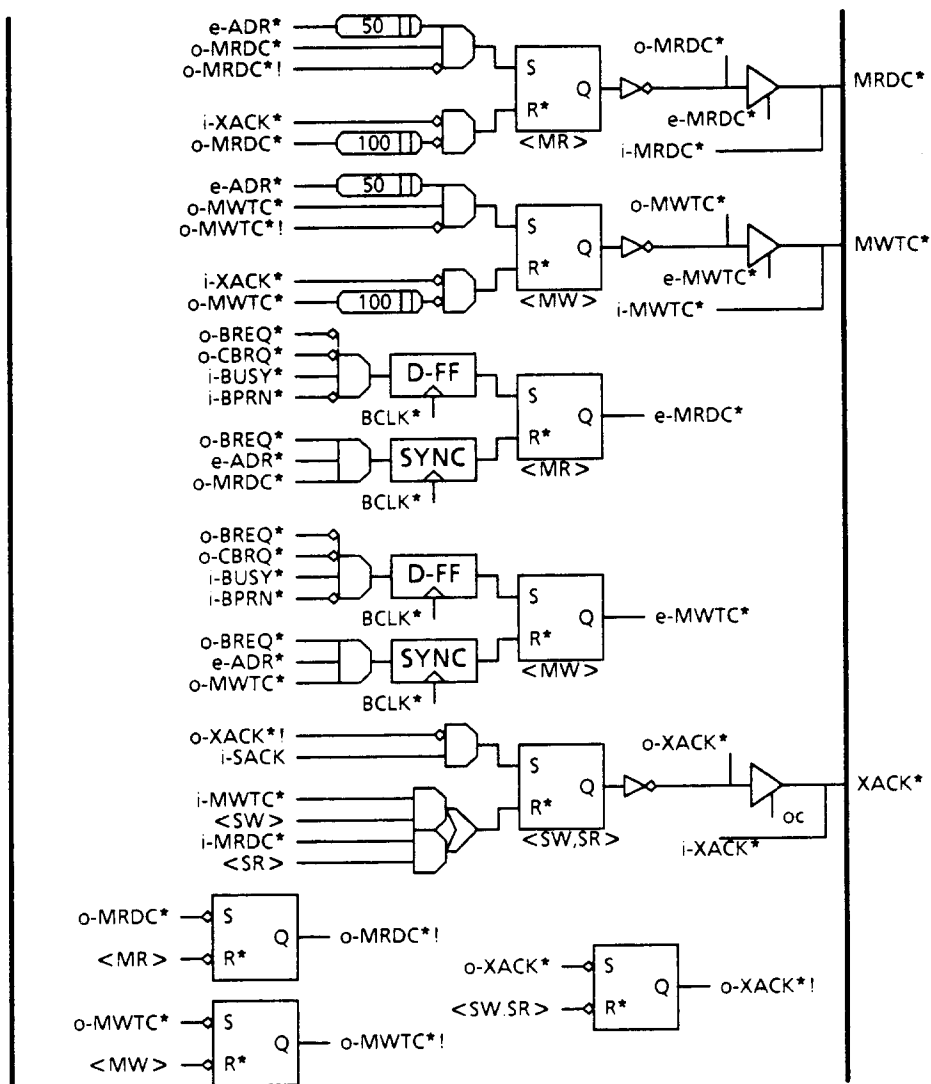


Figure D.7. Janus designed circuit for the Multibus Design Frame (part one of four, see Figures D.8, D.9, and D.10).

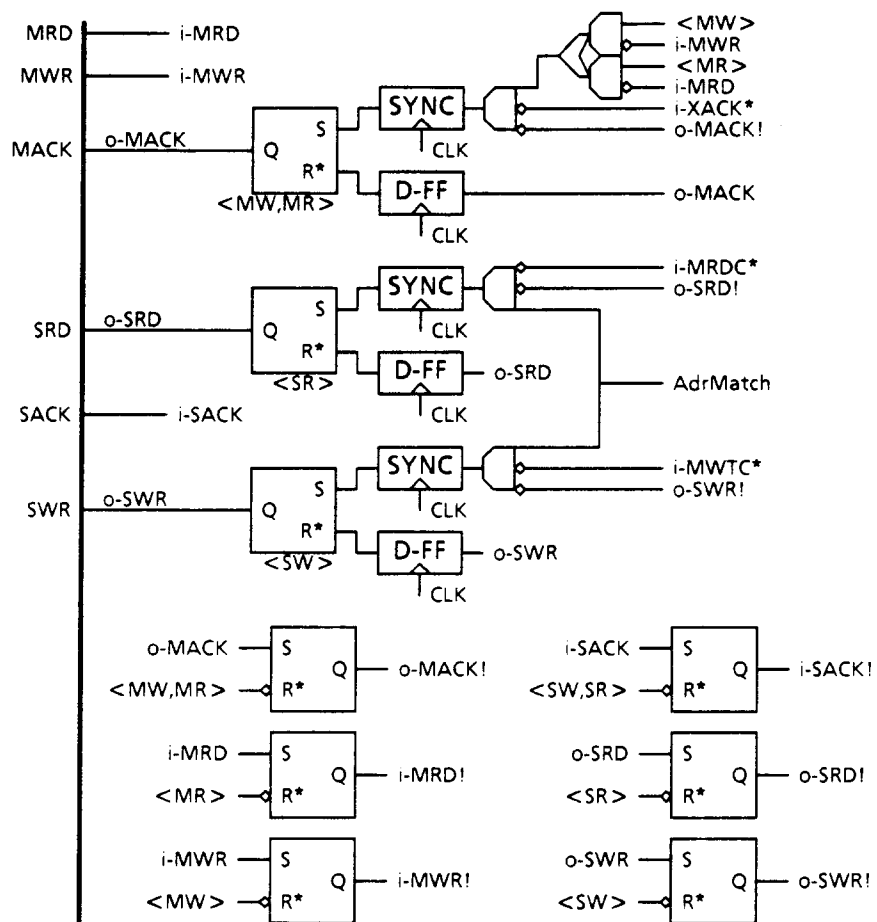


Figure D.8. Janus designed circuit for the Multibus Design Frame (part two of four, see Figures D.7, D.9, and D.10).

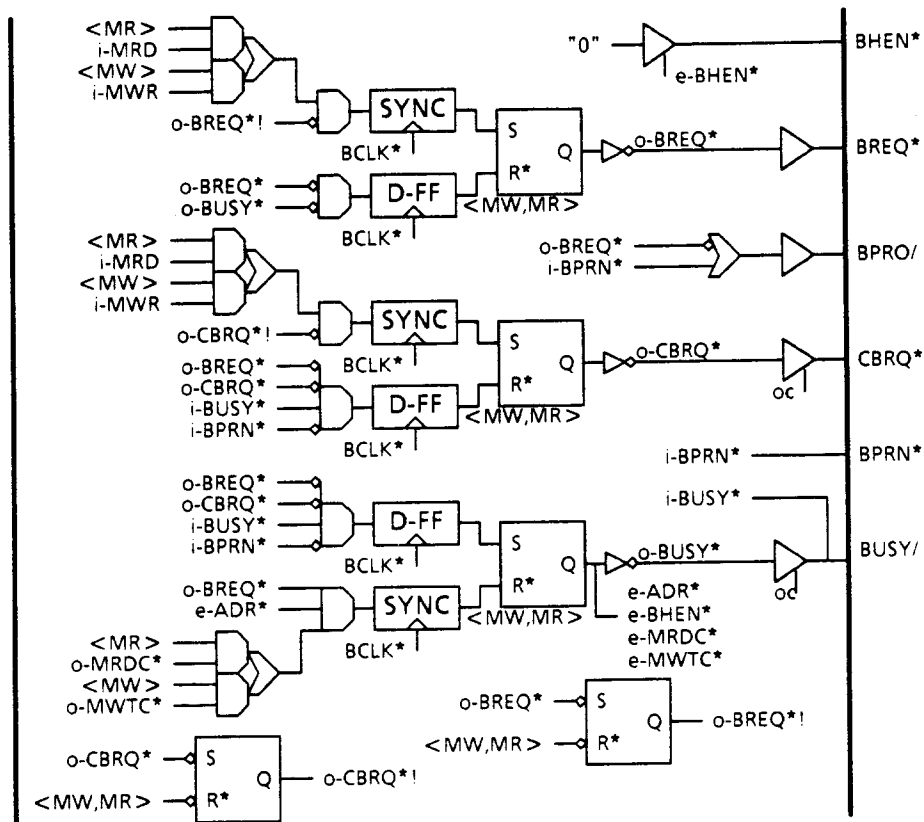


Figure D.9. Janus designed circuit for the Multibus Design Frame (part three of four, see Figures D.7, D.8, and D.10).

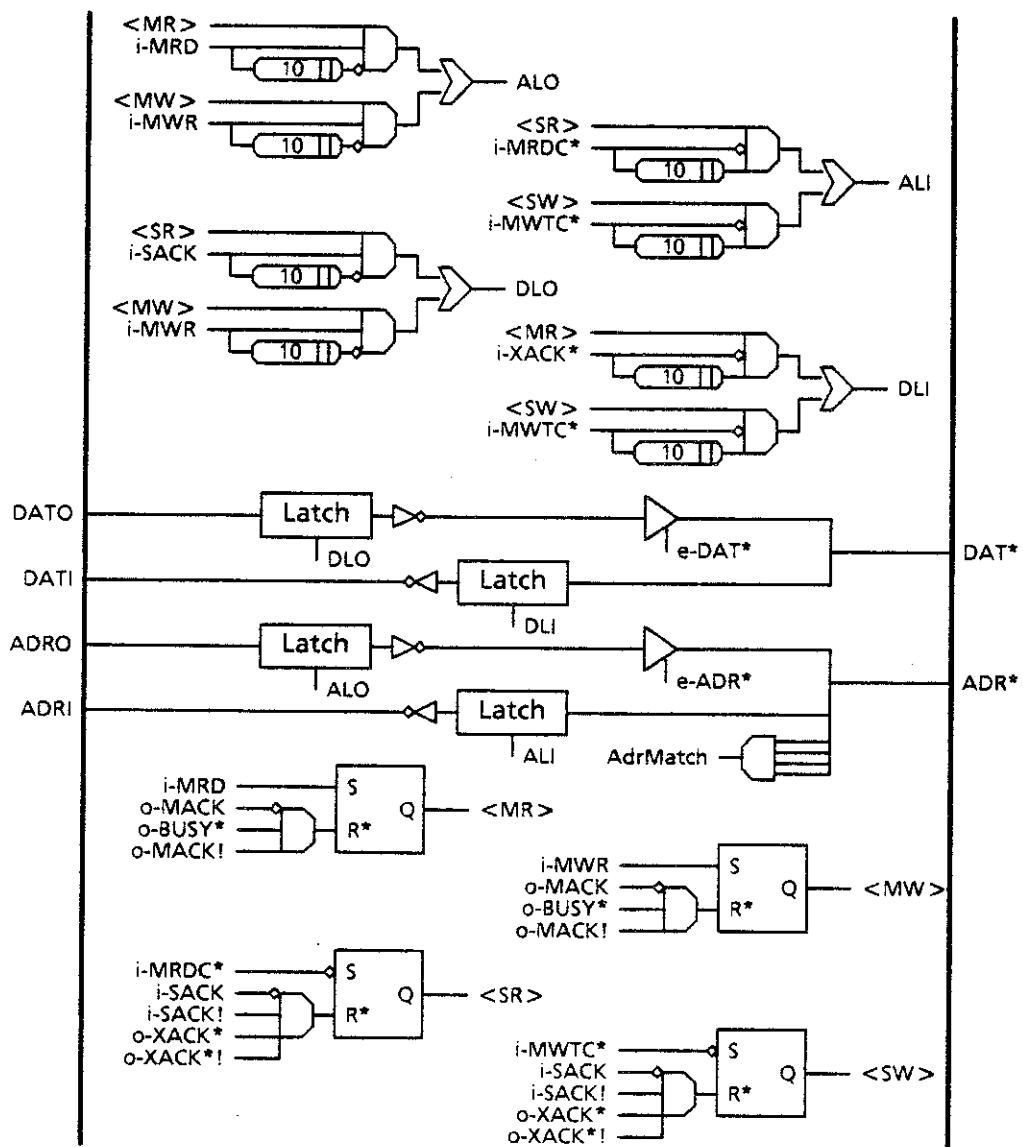


Figure D.10. Janus designed circuit for the Multibus Design Frame (part four of four, see Figures D.7, D.8, and D.9).

The many assumptions and simplifications that designers make during the design process makes it difficult to compare two implementations of a circuit as complex as the Multibus Design Frame. In Table D.1, the two circuits are compared in terms of the total number of logic gates used. Based on this metric, Janus generates a design that is 17% larger than the manually generated design. The numbers in the table are easily derivable from the circuit diagrams of section D.1.2. The only complication is that the counts reflect the size of the Janus circuit after optimizations to remove redundant circuitry. This can be seen in the logic to generate the following groups of signals: e-MRDC* and e-MWTC*; o-MACK, o-BREQ*, and o-CBRQ*; and o-CBRQ* and o-BUSY*. Optimizations to transform an S-R* latch and its two flip-flop or synchronizer inputs into a single synchronizer are possible in the circuitry that generates o-SRD and o-SWR.

Manual			
Part	# Used	Gates	Total
Logic gate	120	1	120
SR	8	2	16
D-FF	3	10	30
SR-FF	1	11	11
Synchronizer	4	10	40
Latch	72	5	360
50ns delay	2	5	10
TOTAL			587

Janus			
Part	# Used	Gates	Total
Logic gate	124	1	124
SR	24	2	48
D-FF	3	10	30
Synchronizer	6	10	60
Latch	72	5	360
10ns delay	8	2	16
50ns delay	1	10	10
100ns delay	2	20	40
TOTAL			688

Table D.1. Comparison of the two circuits for the Multibus Design Frame. The circuit synthesized by *Janus* is 17% larger and 9% faster than the manually designed version. Input/output pads are excluded from the gate counts (there are 44 pads).

Although the circuit generated by *Janus* is larger in size, in terms of performance, it is 9% faster than the manually generated design. This is due to a simplifying assumption on the part of the designer that decreases the amount of parallelism in the circuit. The specification of the MDF states that operations begin on the leading edges of MRD and MWR, however, in the manual design the trailing edges of these signals are used to start the operations. This simplification completely orders the event graph and eliminates many possible race conditions for which Janus generated corrective circuitry. The 9% is derived from the expected duration of the operation (1000ns) and the savings in not waiting for the trailing edge of MRD/MWR (90ns).

Janus is restricted to interface operations whose logic signals start and end at the same logic level. Furthermore, operations cannot overlap in time. Unfortunately, the specification of the 2-phase and 4-phase protocols in section B.3 do not meet these criteria. A new set of specifications that does is shown in Figure D.11.

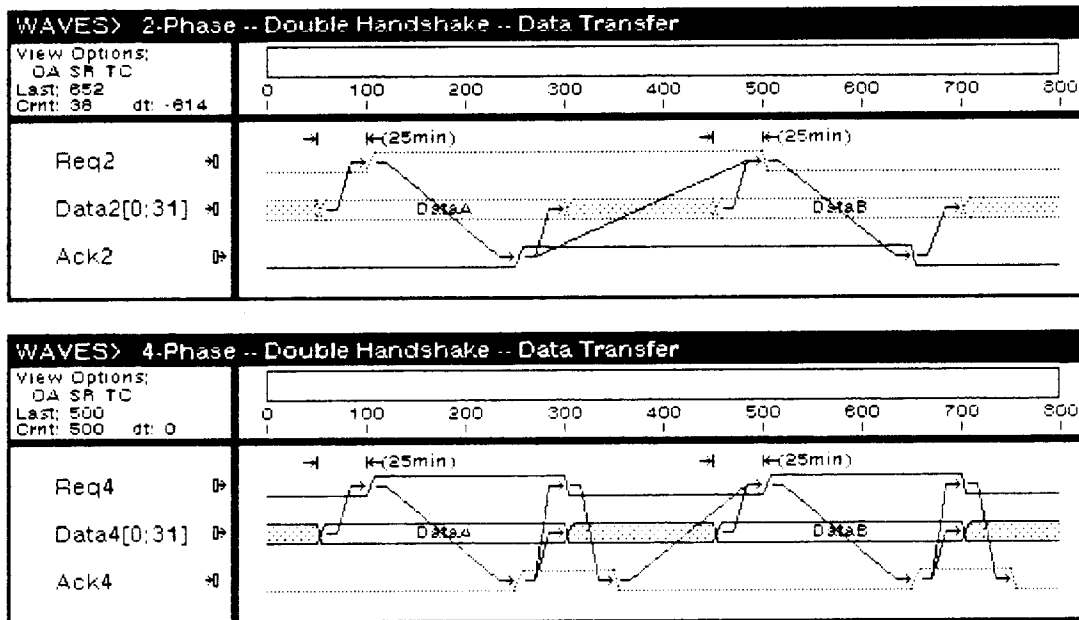


Figure D.11. Waves timing diagram specification for the 2-phase to 4-phase protocol adapter. These are different than those of section B.3 due to the restriction imposed by *Janus* that, for every interface operation, all signals start and end at the same logic levels.

Basically, the operations have been duplicated into a double handshake operation whose signals do return to the same levels. Furthermore, ordering labels exist between the first rising edge of Ack4 and the rising edge of Ack2 as well as the second rising edge of Ack4 and the falling edge of Ack2. Another set of ordering labels is needed to enforce the constraint that the operations not overlap. These labels are placed from the first falling edge of Ack4 to the falling edge of Req2. Unfortunately, this is not quite the same specification as was used to manually design the circuit [Sproull86]. Therefore, *Janus* should generate a smaller, less parallel, circuit for this example.

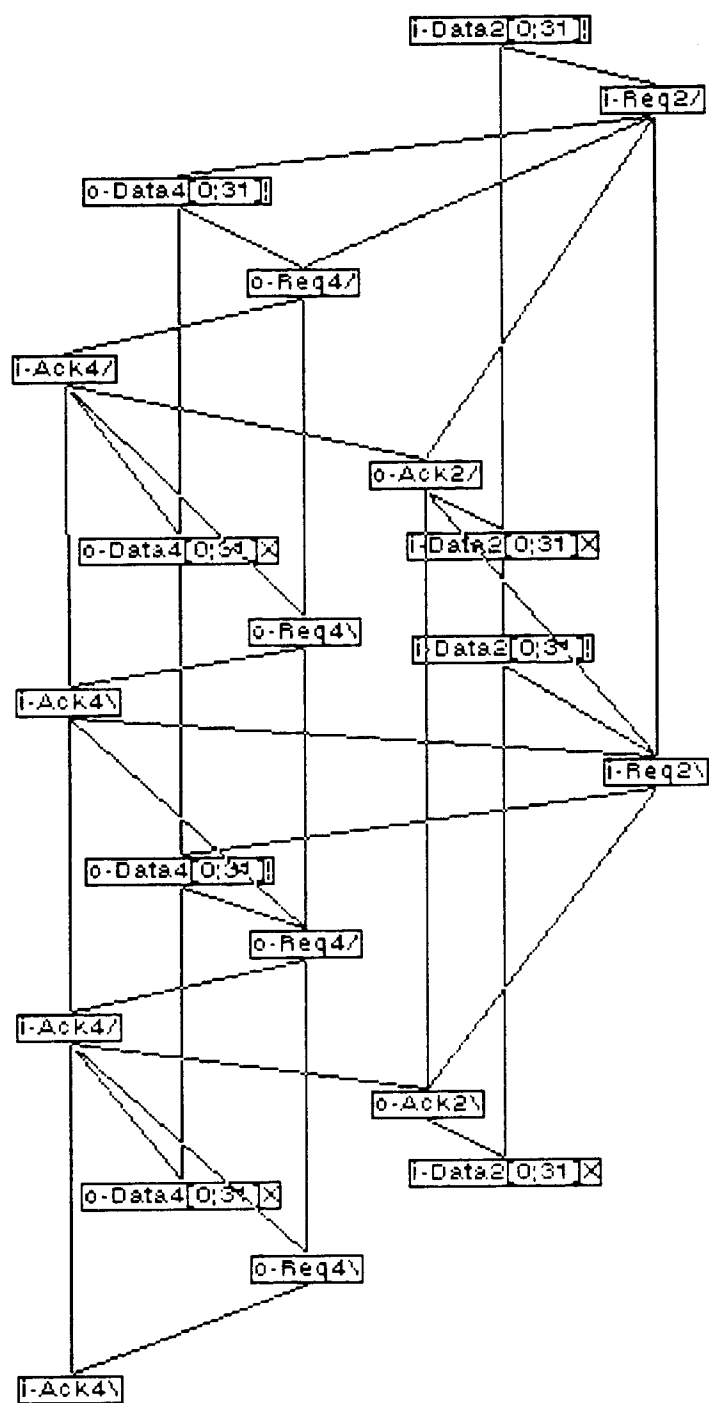


Figure D.12. Event graph for the double handshake operation.

In the two figures of this section, the 2-phase protocol interface is on the left and the 4-phase protocol interface is on the right. Only signals that connect to one of the two boundaries are part of the interface. Internal signals are not connected to either side. In the *Janus* circuit, the signal name in brackets corresponds to the operation enable signal generated for the double handshake operation. When this signal appears under an S-R* latch it signifies that the latch is reset whenever the operation enable signal is low.

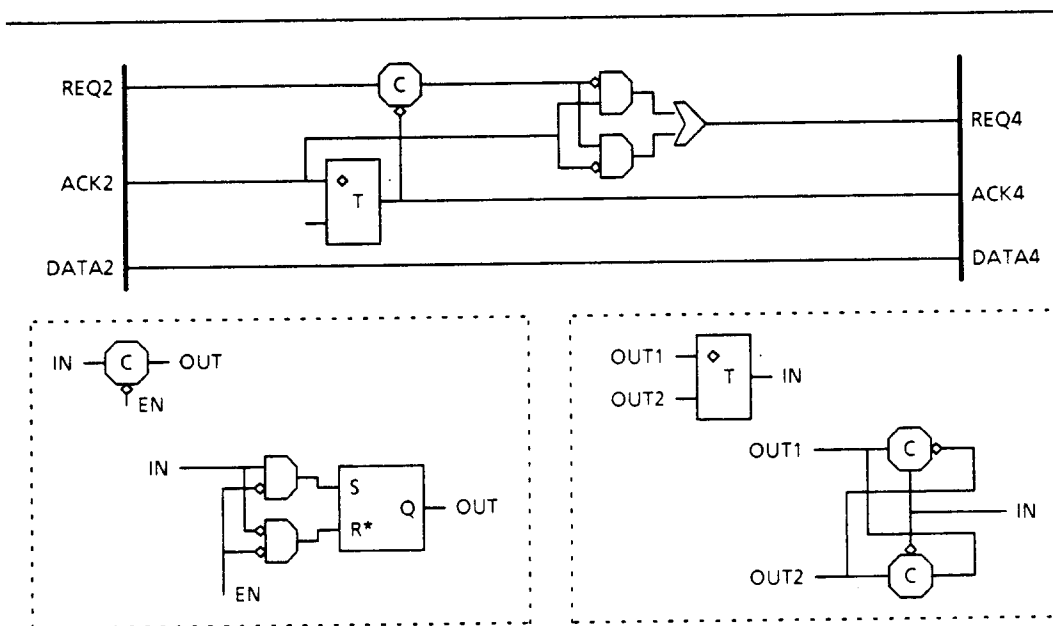


Figure D.13. Manually designed circuit for the 2-phase to 4-phase protocol adapter [Sproull86].

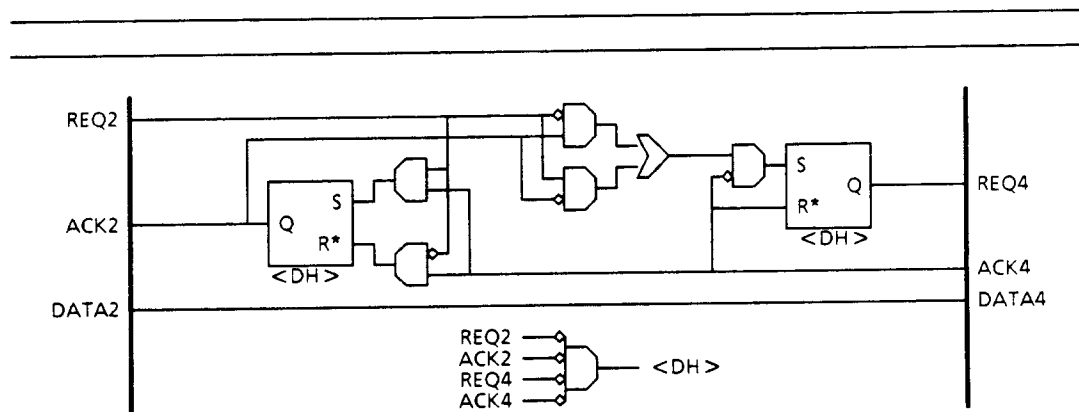


Figure D.14. Janus designed circuit for the 2-phase to 4-phase protocol adapter.

Summary and Comparison

D.2.3

The circuit generated by *Janus* is of equivalent performance to the manually designed version. Its size, as expected, is smaller than the manual design (by 39%) since the circuit implements a less parallel specification. Note also that the operation enabling circuit is a simple AND gate rather than the S-R* latch of the previous example. This is true because there is only one transducer operation and non-quiescent levels on its signals indicate when the operation is in progress.

<hr/>				
Manual				
Part	<u># Used</u>		<u>Gates</u>	<u>Total</u>
Logic gate	6		1	6
C-element	1		5	5
Toggle	1		12	12
TOTAL				23
Janus				
Part	<u># Used</u>		<u>Gates</u>	<u>Total</u>
Logic gate	10		1	10
SR	2		2	4
TOTAL				14

Table D.2. Comparison of the two circuits for the 2-phase to 4-phase protocol adapter. The circuit synthesized by *Janus* is 39% smaller and of equal performance to the manually designed version.

The SPUR PCC-SBC Interface is an interface transducer with three interface operations. The specifications for these and their interrelations are given in section B.4.

Event Graphs

D.3.1

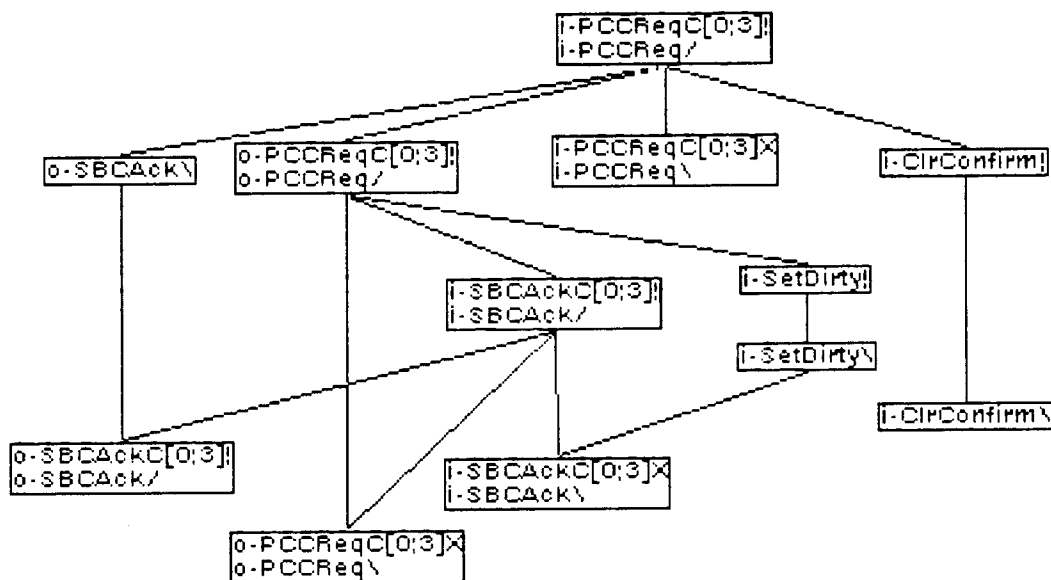


Figure D.15. Event graph for the PCC-to-SBC Request operation of the SPUR PCC-SBC Interface.

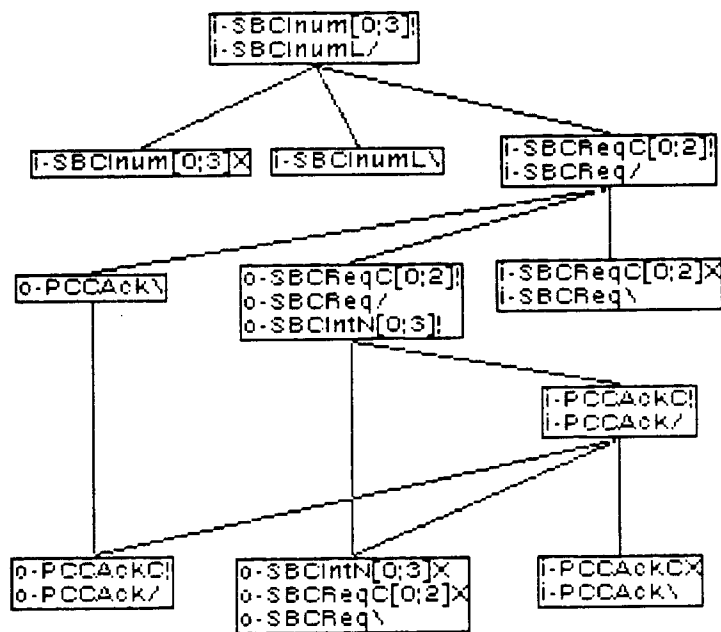


Figure D.16. Event graph for the SBC-to-PCC Request operation of the SPUR PCC-SBC Interface.

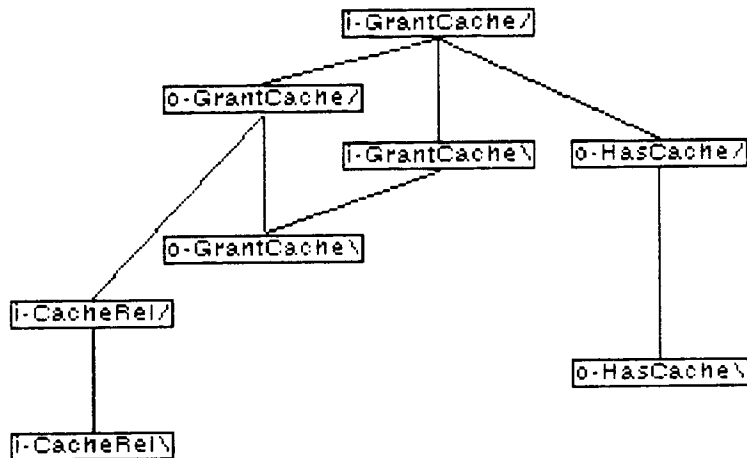


Figure D.17. Event graph for the Cache Handshake operation of the SPUR PCC-SBC Interface.

In the two figures of this section, the SPUR Processor Cache Controller interface is on the left and the SPUR Snooping Bus Controller interface is on the right. Only signals that connect to one of the two boundaries are part of the interface. Internal signals are not connected to either side. In the *Janus* circuit, signal names in brackets correspond to the operation enable signals generated for each interface operation. When these signals appear under an S-R* latch it signifies that the latch is reset whenever the operation enable signal is low.

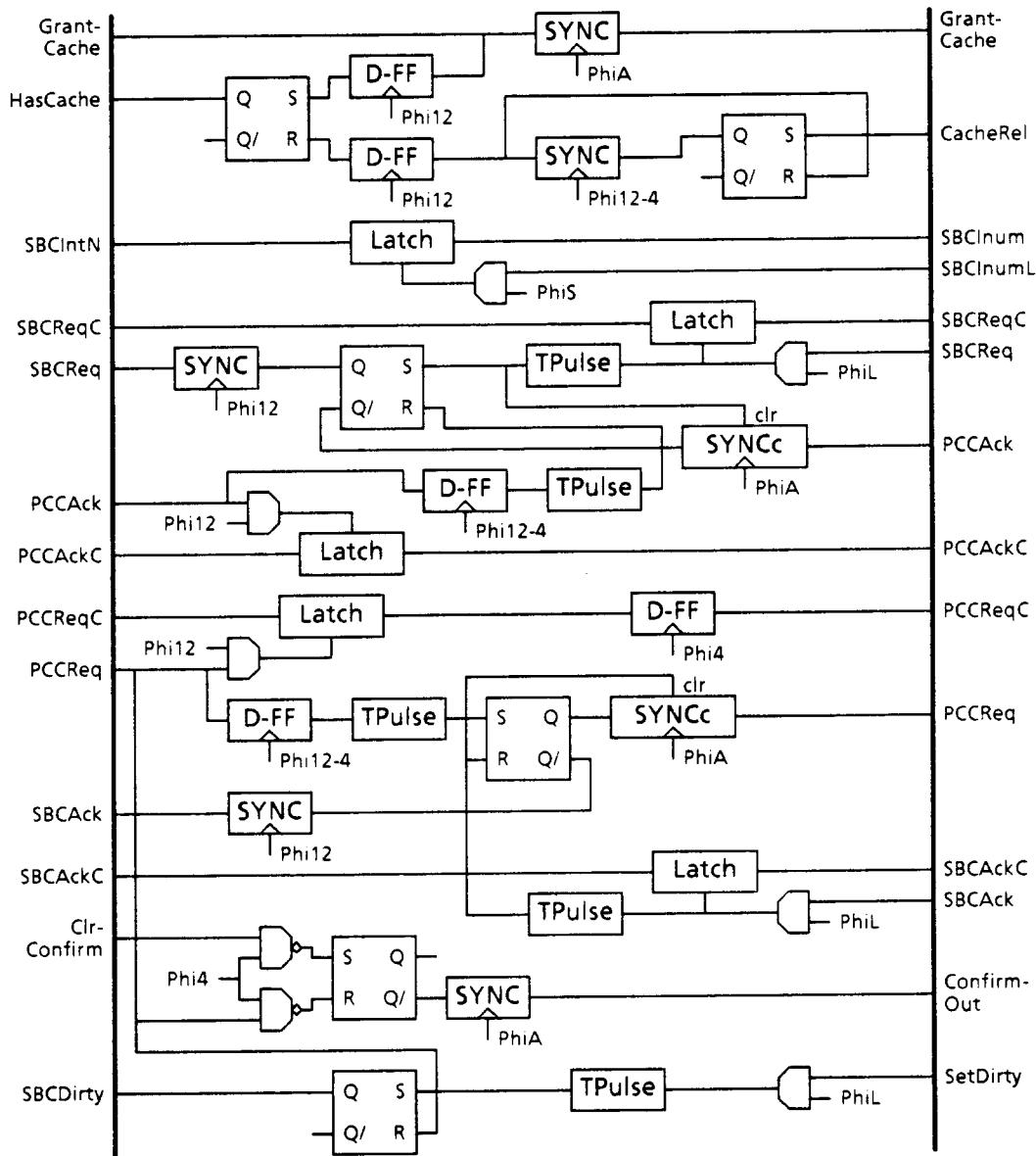


Figure D.18. Manually designed circuit for the SPUR PCC-SBC Interface [Gibson86].

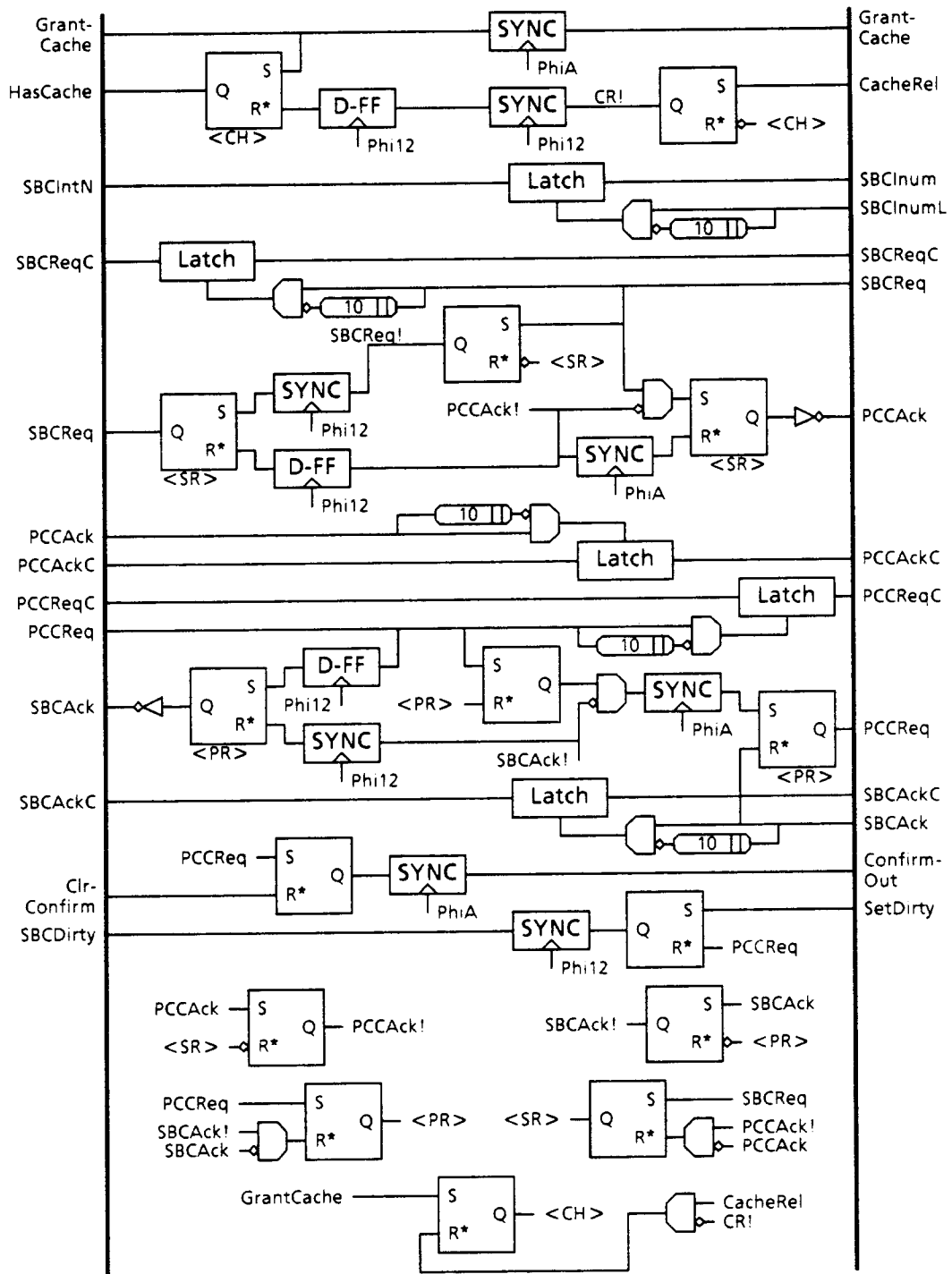


Figure D.19. Janus designed circuit for the SPUR PCC-SBC Interface.

Summary and Comparison

D.3.3

The circuit generated by *Janus* is 11% smaller for this example with similar performance. The transition pulse generator used in the manual design is identical to the latch control circuits synthesized by *Janus* and are therefore given the same gate count in Table D.3. The actual SPUR PCC-SBC Interface is a dynamic-CMOS design. Since *Janus* only synthesizes static designs, the comparison is based on static equivalents of the circuit elements in the manual design. This method of comparison can easily account for the difference in circuit size.

<hr/>			
Manual			
Part	# Used	Gates	Total
Logic gate	8	1	8
SR	6	2	12
D-FF	7	5	35
Synchronizer	5	5	25
Synchronizer (with clear)	2	5	10
Latch	15	5	75
Transition pulse (leading)	5	6	30
TOTAL			195
<hr/>			
Janus			
Part	# Used	Gates	Total
Logic gate	12	1	12
SR	13	2	26
D-FF	1	5	5
Synchronizer	7	5	35
Latch	15	5	75
10ns delay	4	6	24
TOTAL			177

Table D.3. Comparison of the two circuits for the SPUR PCC-SBC Interface. The circuit synthesized by *Janus* is 11% smaller and of equal performance to the manually designed version.

References

The references are divided into groups corresponding to the three main parts of this dissertation: Part I (Chapters 2 and 3), Part II (Chapters 4 and 5), and the four appendices (A, B, C, and D). Some references may be duplicated if they are referenced in more than one part.

Part I

- [Agerwala79] T. Agerwala, Putting Petri Nets to Work, IEEE Computer, December 1979.
- [Altman80] A. Altman, A. Parker, The Slide Simulator: A Facility for the Design and Analysis of Computer Interconnections, ACM, 1980.
- [Arnold85] J. Arnold, The Knowledge-Based Test Assistant's Wave/Signal Editor: An Interface for the Management of Timing Constraints, Proceedings of the Second Conference on Artificial Intelligence Applications, December 1985.
- [Barbacci76] M. Barbacci, The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator, Technical Report, Department of Computer Science, Carnegie-Mellon University, April 1976.
- [Barbacci81] M. Barbacci, Instruction Set Processor Specification (ISPS): The Notation and its Applications, IEEE Transactions on Computers, January 1981.
- [Bobrow83] D. Bobrow, The LOOPS Manual, Xerox Artificial Intelligence Systems, December 1983.
- [Bochmann82] G. Bochmann, Hardware Specification with Temporal Logic: An Example, IEEE Transactions on Computers, Vol. C-31, No. 3, March 1982.
- [Borriello85] G. Borriello, R. Katz, Design Frames: A New System Integration Methodology, Chapel Hill Conference on VLSI, May 1985.
- [DenBeste86] W. Den Beste, Tools for Test Development, VLSI Systems Design Magazine, July 1986.
- [Dill85] D. Dill, E. Clarke, Automatic Verification of Asynchronous Circuits Using Temporal Logic, Chapel Hill Conference on VLSI, May 1985.
- [Goos80] G. Goos, J. Hartmanis, Eds., Net Theory and Applications, Lecture Notes in Computer Science 84, Springer-Verlag, 1980.
- [Granacki86a] J. Granacki, A. Parker, A Natural Language Interface for Specifying Digital Systems, First International Conference on Application of Artificial Intelligence to Engineering Problems, Southampton, England, April 1986.
- [Granacki86b] J. Granacki, Understanding Digital System Specifications Written in Natural Language, Ph.D. Dissertation, Department of Electrical Engineering, University of Southern California, Report No. CRI-87-02, December 1986.
- [Intel82] Intel Multibus Specification, Intel Corporation, 1982.
- [Ikos86] Ikos Systems, Accelerated Stimulation and Simulation, Product Showcase, VLSI Systems Design Magazine, June 1986.
- [Kara86] L. Kara, R. Rastogi, K. Kawamura, TDS: An Expert System to Automate Timing Design For Interfacing VLSI Chips in Microcomputer Systems, International Conference on Computer-Aided Design, November 1986.
- [Katz83] R. Katz, S. Weiss, Chip Assemblers: Concepts and Capabilities, 20th Design Automation Conference, 1983.
- [Kelly84] V. Kelly, The CRITTER System: Automated Critiquing of Digital Circuit Designs, 21st Design Automation Conference, 1984.

- [Kimura87] S. Kimura, S Yajima, The Description and Verification of Input Constraints and Input-Output Specifications of Logic Systems Using a New Extended Regular Expression, International Conference on VLSI (VLSI87), August 1987.
- [Koomen85] C. Koomen, Algebraic Specification and Verification of Communication Protocols, in Science of Computer Programming 5, North-Holland, 1985.
- [Milner80] R. Milner, A calculus of communicating systems, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.
- [Misunas73] D. Misunas, Petri Nets and Speed Independent Design, Communications of the ACM, Vol. 16, No. 8, August 1973.
- [Molnar85] C. Molnar, T. Fang, F. Rosenberger, Synthesis of Delay-Insensitive Modules, 1985 Chapel Hill Conference on VLSI, May 1985.
- [Moszkowski85] B. Moszkowski, A Temporal Logic for Multilevel Reasoning about Hardware, IEEE Computer, February 1985.
- [Nestor86] J. Nestor, D. Thomas, Behavioral Synthesis with Interfaces, International Conference on Computer-Aided Design, November 1986.
- [Nestor87a] J. Nestor, Specification and Synthesis of Digital Systems with Interfaces, Ph. D. Dissertation, Department of Electrical and Computer Engineering, Carnegie-Mellon University, Report No. CMUCAD-87-10, April 1987.
- [Nestor87b] J. Nestor, private communication, July 1987.
- [Parker81] A. Parker, J. Wallace, SLIDE: An I/O Hardware Descriptive Language, IEEE Transactions on Computers, Vol. C-30, No. 6, June 1981.
- [Parker85] A. Parker, N. Park, Interface and I/O Protocol Descriptions, Section 3.3 of Advances in CAD for VLSI, Vol. 7: Hardware Description Languages, R. Hartenstein, Editor, North-Holland, 1985.
- [Petri62] C. Petri, Fundamentals of a Theory of Asynchronous Information Flow, Proceedings of the IFIP Congress 1962, Munich, North-Holland, 1962.
- [Rony80] P. Rony, Interfacing Fundamentals: Timing Diagram Conventions, Computer Design, January 1980.
- [Stefik86] M. Stefik, D. Bobrow, Object-Oriented Programming: Themes and Variations, Artificial Intelligence Magazine, Volume VI, Number 4, Winter 1986.
- [TexasInstruments85] Texas Instruments NuBus Specification, Texas Instruments Incorporated, 1985.
- [Thomas83] D. Thomas, et al., Automatic Data Path Synthesis, IEEE Computer, December 1983.
- [Vissers76] C. Vissers, Interface, A Dispersed Architecture, Proceedings of the Third Annual Symposium on Computer Architecture, 1976.
- [Xerox86] Xerox Corporation, Interlisp-D Reference Manual (Koto Release), Xerox Artificial Intelligence Systems, 1986.

Part II

- [Bobrow83] D. Bobrow, The LOOPS Manual, Xerox Artificial Intelligence Systems, December 1983.
- [Brayton87] R. Brayton, et. al., MIS: A Multiple-Level Logic Optimization System, IEEE Transactions on Computer-Aided Design, Vol. CAD-6, No. 6, November 1987.
- [Burns86] J. Burns, A. Newton, SPARCS: A New Constraint-Based IC Symbolic Layout Spacer, Proceedings of the Custom Integrated Circuits Conference, 1986.
- [Chu86a] T. Chu, On the Models for Designing VLSI Asynchronous Digital Systems, Integration, the VLSI journal, Vol. 4, August 1986.
- [Chu86b] T. Chu, L. Glasser, Synthesis of Self-Timed Control Circuits from Graphs: An Example, Proceedings of the IEEE International Conference on Computer Design, October 1986.
- [Chu87] T. Chu, Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications, Proceedings of the IEEE International Conference on Computer Design, October 1987.
- [DeMicheli85] G. DeMicheli, R. Brayton, A. Sangiovanni-Vincentelli, Optimal State Assignment for Finite-State Machines, IEEE Transactions on Computer-Aided Design, Vol. CAD-4, July 1985.
- [Fisher81] J. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Transactions on Computers, Vol. C-30, No. 7, July 1981.
- [Gajski83] D. Gajski, R. Kuhn, Guest Editors' Introduction: New VLSI Tools, IEEE Computer, Vol. 16, No. 12, December 1983.
- [Girczyc85] E. Girczyc, R. Buhr, J. Knight, Applicability of a Subset of Ada as an Algorithmic Hardware Description Language for Graph-Based Hardware Compilation, IEEE Transactions on Computer-Aided Design, Vol. CAD-4, No. 2, April 1985.
- [Hollaar82] L. Hollaar, Direct Implementation of Asynchronous Control Units, IEEE Transactions on Computers, Vol. C-31, No. 12, December 1982.
- [Leive81] G. Leive, D. Thomas, A Technology Relative Logic Synthesis and Module Selection System, 18th Design Automation Conference, 1981.
- [Misunas73] D. Misunas, Petri Nets and Speed Independent Design, Communications of the ACM, Vol. 16, No. 8, August 1973.
- [Molnar85] C. Molnar, T. Fang, F. Rosenberger, Synthesis of Delay-Insensitive Modules, 1985 Chapel Hill Conference on VLSI, May 1985.
- [Nestor86] J. Nestor, D. Thomas, Behavioral Synthesis with Interfaces, International Conference on Computer-Aided Design, November 1986.
- [Nestor87] J. Nestor, Specification and Synthesis of Digital Systems with Interfaces, Ph. D. Dissertation, Department of Electrical and Computer Engineering, Carnegie-Mellon University, Report No. CMUCAD-87-10, April 1987.
- [Parker86] A. Parker, J. Pizarro, M. Mlinar, MAHA: A Program for Datapath Synthesis, 23rd Design Automation Conference, 1986.

- [Paulin87] P. Paulin, J. Knight, Force-Directed Scheduling in Automatic Data Path Synthesis, 24th Design Automation Conference, 1987.
- [Rudell87] R. Rudell, A. Sangiovanni-Vincentelli, Multiple-Valued Minimization for PLA Optimization, IEEE Transactions on Computer-Aided Design, Vol. CAD-6, No. 5, September 1987.
- [Sangiovanni86] A. Sangiovanni-Vincentelli, A Design System for the Automatic Synthesis of VLSI ICs, IEEE International Conference on Computer Design, October 1986.
- [Stefik86] M. Stefik, D. Bobrow, Object-Oriented Programming: Themes and Variations, Artificial Intelligence Magazine, Volume VI, Number 4, Winter 1986.
- [Thomas83] D. Thomas, et al., Automatic Data Path Synthesis, IEEE Computer, December 1983.
- [Walker85] R. Walker, D. Thomas, A Model of Design Representation and Synthesis, 22nd Design Automation Conference, 1985.
- [Xerox86] Xerox Corporation, Interlisp-D Reference Manual (Koto Release), Xerox Artificial Intelligence Systems, 1986.

Appendices

- [Bobrow83] D. Bobrow, The LOOPS Manual, Xerox Artificial Intelligence Systems, December 1983.
- [Bobrow88] D. Bobrow, et. al., Common Lisp Object System Specification, informally circulated document, March 1988.
- [Borriello85] G. Borriello, R. Katz, Design Frames: A New System Integration Methodology, 1985 Chapel Hill Conference on VLSI, May 1985.
- [Intel82] Intel Multibus Specification, Intel Corporation, 1982.
- [Gettys86] J. Gettys, X Version 10 Protocol Guide, Technical Report, Massachusetts Institute of Technology, 1986.
- [Gibson86] G. Gibson, D. Wood, S. Eggers, Detailed Functional Description of the Interface Between the Processor Cache Controller and the Snopping Bus Controller, SPUR Project Technical Documentation, Computer Science Division, University of California, Berkeley, 1986.
- [Hill86] M. Hill, et. al., Design Decisions in SPUR: A VLSI Multiprocessor, IEEE Computer, November 1986.
- [Lanning86] S. Lanning, Simple Windows Documentation, Xerox Palo Alto Reseach Center Internal Documentation, March, 1986.
- [Lattice84] SR64K4 High-Speed 64K Static RAM (16Kx4) Technical Data Sheets, Lattice Semiconductor Corporation, 1984.
- [Molnar85] C. Molnar, T. Fang, F. Rosenberger, Synthesis of Delay-Insensitive Modules, 1985 Chapel Hill Conference on VLSI, May 1985.
- [Motorola81] Motorola Microprocessors Data Manual, Motorola, Incorporated, 1981.
- [OCT87] OCT Tools Distribution 2.0, Electronics Research Laboratory, University of California, Berkeley, November 1987.
- [Sproull86] R. Sproull, I. Sutherland, Asynchronous Systems, Textbook in preparation, Sutherland, Sproull, and Associates, Incorporated, 1986.
- [Stefik86] M. Stefik, D. Bobrow, Object-Oriented Programming: Themes and Variations, Artificial Intelligence Magazine, Volume VI, Number 4, Winter 1986.
- [Terman87] C. Terman, RNL 4.2 User's Guide, VLSI Design Tools Reference Manual, Northwest Laboratory for Integrated Systems, University of Washington, February 1987.
- [TexasInstruments85] Texas Instruments NuBus Specification, Texas Instruments Incorporated, 1985.
- [Xerox86] Xerox Corporation, Interlisp-D Reference Manual (Koto Release), Xerox Artificial Intelligence Systems, 1986.