DESIGN OF SYSTEM-LEVEL INTERFACES

by

Jane Shih Sun

Memorandum No. UCB/ERL M92/105

10 September 1992

DESIGN OF SYSTEM-LEVEL INTERFACES

by

Jane Shih Sun

Memorandum No. UCB/ERL M92/105

10 September 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DESIGN OF SYSTEM-LEVEL INTERFACES

by

Jane Shih Sun

Memorandum No. UCB/ERL M92/105

10 September 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

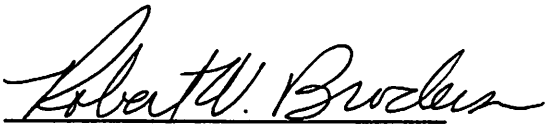
Design of System-Level Interfaces

by

Ph.D. **Jane Shih Sun** **Department of EECS**

Abstract

Complex digital systems are designed with hardware modules that interact by transferring information and synchronizing their inputs and outputs. The modules can be constructed from a variety of single IC components to subsystems, and typically they have incompatible I/O and communication protocols. A large portion of the integration time is thus devoted to designing the interfaces between interacting modules. This thesis presents a design methodology and behavioral synthesis techniques for integrating hardware modules into a system. The interface between modules, which can obey arbitrary protocols, is generated from a high-level system specification developed especially for describing inter-module communication. Central to the design methodology are libraries that contain system level module generators and a strategy to capture the protocol and timing information necessary for interface synthesis. Application of the interface design methods toward systems for general-purpose computing and real-time robot control are also described.



Robert W. Brodersen

Chairman of Committee

To Mom and

especially Dad.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

When I first came to Berkeley to get a M.S. degree, I had not considered the Ph.D. degree. After seven years, the later has happened, with support and encouragement from many people along the way. My advisor, Bob Brodersen, provided incredible research facilities, technical and administrative support staff, guidance, and the financial support that made my pursuit possible, productive and enjoyable. The seed idea of this dissertation owes a lot to him, as does the kick to get this dissertation written. Two of the biggest lessons I learned through him was the importance of "getting to the heart" of the issue, and looking for the "big-picture" and the "real significance". I sincerely appreciate his support.

Next, I want to recognize those who contributed to my research whether directly or indirectly. The other members of my dissertation committee Carlo Sequin and Charles Stone gave valuable feedback on my research and writing. The work presented in this dissertation combines concepts from different research areas within electrical engineering. Discussions with Mani Srivastava and Gaetano Borriello really helped me to realize the significance of each area and the relationship between them. The concepts in Section 6.1.2 come from the examples and informative talks Mani provided to me. Miodrag Potkonjak and Phu Hoang provided insightful discussions and valuable reading materials on high-level synthesis. The intricacies of asynchronous logic synthesis were clarified by Gaetano Borriello, Teresa Meng, Gary Jones and Cho Moon among others.

When it came time for implementing research ideas, thanks to Tod Amon and Karen Bartlett and Gaetano Borriello for providing the OEsim simulator, Phu Hoang for help in using the HYPER infrastructure, Dan Tabori for implementing the IDL grammar (and Monte Mar's pointers), and Ricky Tang for the CLOVER translator. Good examples are essential for testing and refining research ideas. Much thanks to Gautam Doshi, Anantha Chandrakasan, Bill Baringer and Mani Srivastava for providing a variety of real world examples.

The string of people in "bjgroup" made life in Cory Hall lighter and fun. My long-term cubicle mates included Anantha Chandrakasan who brought social life into bjgroup, the Mani Srivastava who always had an answer to my questions, and Kevin Kornegay from whom I came to appreciate voice mail. I also won't forget those bike rides with Andy Burstein and Shankar Narayanaswamy. Also, much gratitude to Sam Sheng for help with my Mac.

Outside of Cory, my string of Co-op roommates or apartments each introduced me to something new about life. Heather Grey, Ilona Rutka, Edith Kaneshiro, Elaine Brasher ... Tomoko Negishi. Edith, I look forward to the completion of your dissertation. Tomoko, thanks for understanding my mood swings.

Finally, I want to acknowledge three very important people. My work demanded their patience as well as mine. To my mother, Juliet, thank you for your love and showing me what courage is. To my father, Arnold P.K., this dissertation is *especially* dedicated to you. Thank you for passing a sense of independence and ambition to me. This achievement is as much yours asit is mine. Without any asking, Stephen Meier (now my husband) has stood by my academic pursuit for years, providing active encouragement and good times along the way. In fact, ideas in Section 3.1 and 3.5 stem from technical discussions with him. Thank you, Steve.

CHAPTER 1

INTRODUCTION

The emphasis in today's computer systems is on interconnecting computation and memory modules so that information can be shared and distributed. This is evident across a variety of applications such as general-purpose computing, a network communication node, real-time robot control, and even a portable multi-media terminal integrating speech and video capabilities. The one common design approach they all have to reduce design time and costs is using modules to form new and larger modules which can be reused in other related applications.

Designing such systems is a complex problem because a hierarchical module is built using components and subsystems, or submodules, from various technologies and with a variety of timing strategies. Advances in the VLSI industry offer today's designer application-specific ICs (ASICs), multi-chip-modules (MCMs), field programmable devices (FPGAs and PLDs), memories, general-purpose digital signal processors (DSPs), and even system busses and card cages, among other off-shelf devices, as shown in the board-level examples in Figure 1-1. Using various technologies contributes to two main design problems. The first is how to partition the complete system function onto hardware and software. The second is how the make the hardware

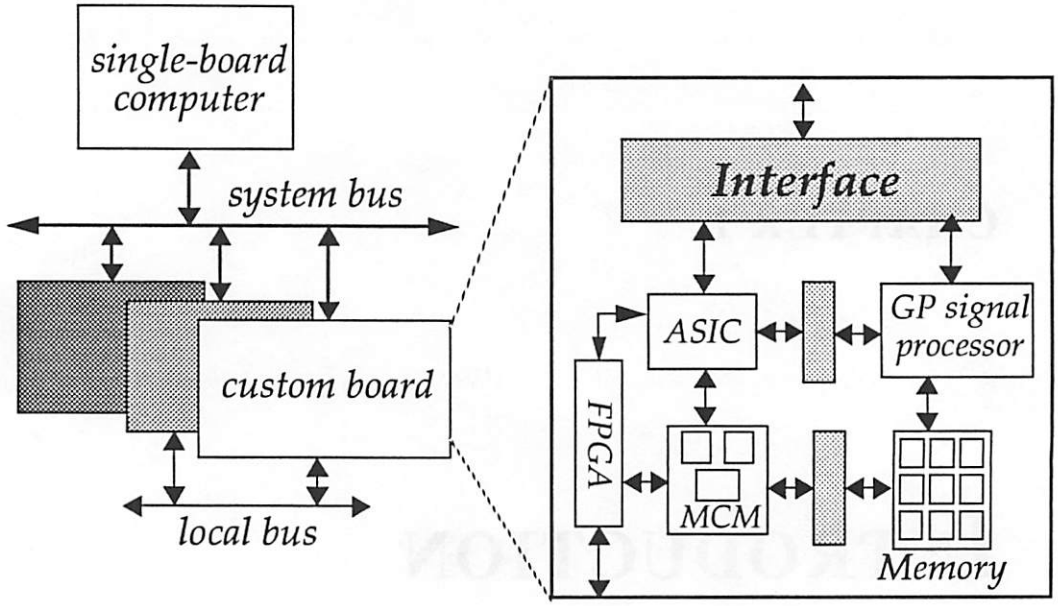


Figure 1-1 : Real-Time System using Various Technologies

modules communicate and synchronize properly when integrated into a system.

The *interface module* is a special system module whose function is to interconnect communicating modules. Integrating the hardware is a real design problem, because each of the interacting components may be using arbitrary and incompatible I/O protocols to communicate with its environment. This thesis focuses on interfacing which is one of the two main design problems.

The distinction between interface modules and other system modules is made, solely, because interfaces coordinate and synchronize the transfer of information between modules that, on the other hand, do the actual data processing by implementing the various computation algorithms required by the system function. Interfaces are concerned with the I/O structure and behavior of the processing modules to be interconnected, rather than what the processing modules internally do. Applications of interface modules range from simple protocol converters to intelligent I/O processors for communication between a microprocessor subsystem and peripheral devices. An

example of a protocol converter is a circuit that translates between the 2-phase and the 4-phase handshake protocols. Typical and more complex applications include system and multiprocessor bus interfaces, direct-memory-access (DMA) controllers, I/O processors, and even application-specific protocol processors that interconnect custom modules.

1.1 Design Issues

To clarify what is meant in this thesis by the term interface module[1] the following examples provide typical interface applications from various systems. These examples are also used to give a sense for the key issues and difficulties involved in designing interface modules.

VME System Bus Interface

Figure 1-2 shows an example of a system bus interface providing read and write access from the VME system bus to the shared memory port of a uni-processor module. During a write access, the VMEbus[VMEbus] is the source of address and data information, while the uni-processor port is the destination of address and data. In the read transfer, address information still flows from the VMEbus to the uni-processor port, but data flows in the opposite direction. The read and write access also requires the decoding of the upper VME address bits to determine if the uni-processor address space has been selected. Accordingly, the interface exercises conditional control over the communications. To synchronize the transfer with their environment, both the VMEbus and uni-processor port use a signaling protocol exercised on control signals and depicted with timing diagrams. The protocols shown are somewhat simplified to clarify the example. Both modules use different protocols, so the interface also adapts between the protocols to meet time constraints such as set-up and hold times and time between signal transitions (also called events). What is important to observe in this example is that the interface module performs system-level

[1] The word "interface" is also used in literature to refer to the physical, electrical and logical characteristics of a functional module's I/O boundary.

communication functions as well as local communication functions, the former being address decoding and the later being protocol synchronization and timing.

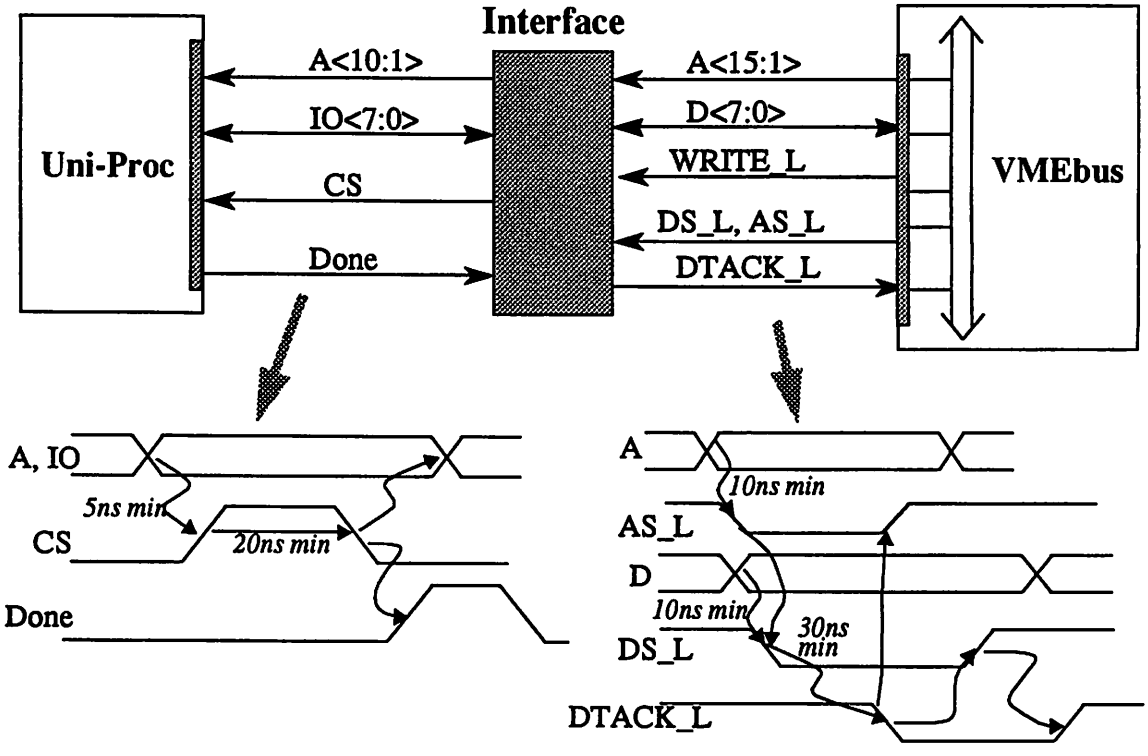


Figure 1-2 : Interface for VMEbus and Uni-processor

Protocol Processor for TMS320 to Optical Link

The next interface is an application-specific protocol processor that couples the TMS320 digital signal processor[TMS320] to a TAXI optical transmitter[TAXIxmt] for access to a robot control peripheral. As shown in Figure 1-3, the TMS port sends address and data words on parallel busses using the indicated handshake protocol shown. The transmitter device has a single bus that carries only one of the words at a time. To coordinate the communication, the interface accepts and places the two TMS words into internal storage and releases the TMS port handshake. Then it multiplexes the address and data transfers, attaching a header to either word before sending it to

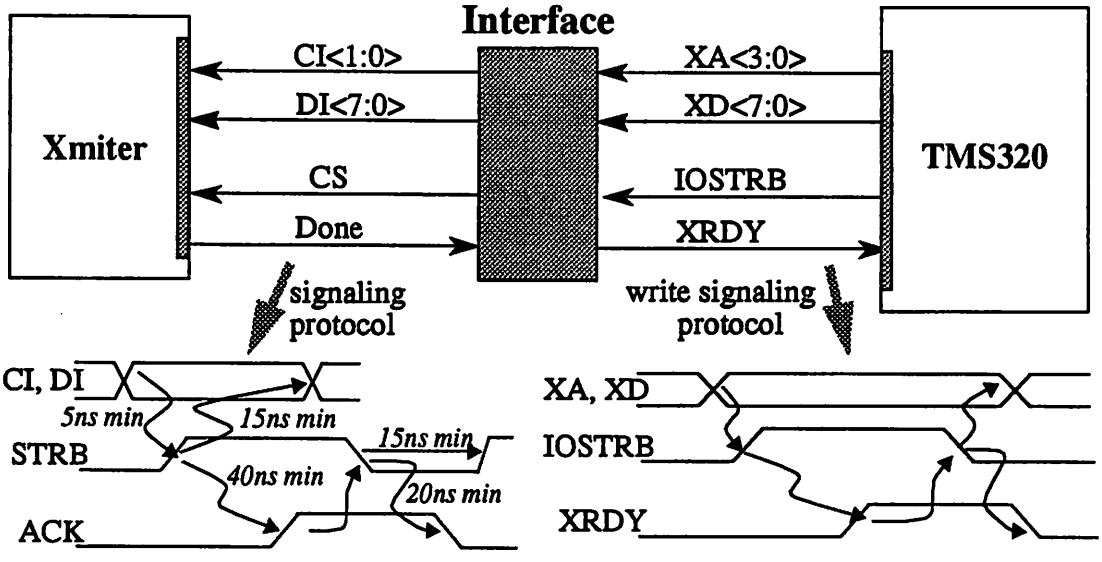


Figure 1-3 : Interface for TMS320 Signal Processor to TAXI Optical Transmitter

the transmitter on the CI and DI signal inputs. So, in this application, the interface performs data storage and formatting functions as well as the protocol resolution function. This example points out how the communication performance is affected by the interface design. If the interface did not store the TMS address and data, the TMS would have to hold those words until the interface completes the multiplexing. This reduces the potential bandwidth of the TMS port, especially if it communicates with modules other than the optical transmitter.

DMA Controller for Video Decompression

The last example, providing more complex control than the previous examples, is an interface module in a video decompression subsystem. It transfers blocks of pixel words from a frame-buffer memory to a digital-to-analog converter (DAC) upon command from the host module. In Figure 1-4, the host initiates the communication by sending a starting memory address and then handing the communication control over to the interface. The interface uses the start address to sequentially transfer consecutive pixels in memory to the DAC. During each transfer, the interface

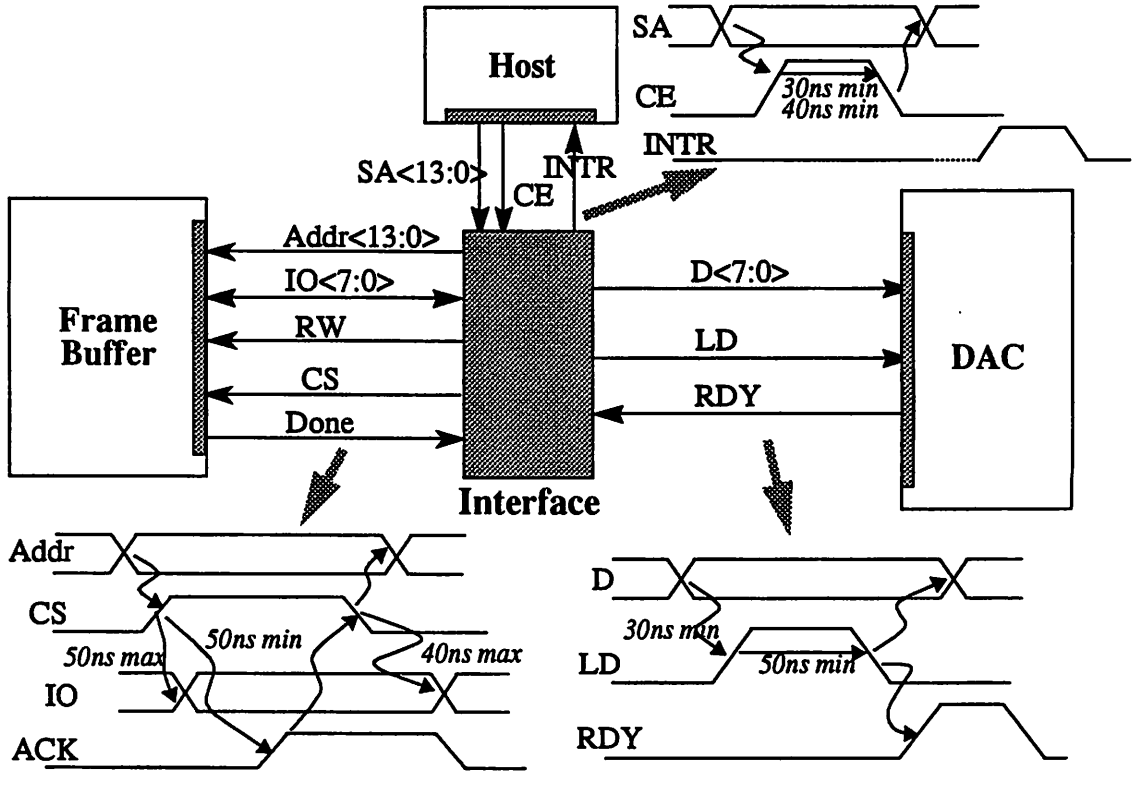


Figure 1-4 : Interface for DMA control of a Video Frame Buffer

generates a new memory address by incrementing the previous address, and interlocks memory and DAC protocol events to synchronize the pixel transfer. When a block of pixels is transferred, the interface module issues an interrupt to the host, completing the communication cycle. This interface exemplifies direct-memory-access communications, which requires the interface to have computation capabilities as well as control of communication between multiple source and destination modules.

All the examples described demonstrate the three primary functions of an interface module:

a. Establishing the physical path to move the basic information to be transferred from source to destination modules. This includes routing and storage of information, transformations such at formatting, logical and arithmetic computations, and even parallel-to-serial conversion. Basic information consists of data, address and read/write/status signals[Stone82].

b. Sequencing the transfers between the modules, such as exercising conditional or looped transfers. Controlling memory read/write access and decoding are examples of conditional control; a DMA block transfer is an example of looped control.

c. Synchronizing communication modules during a transfer. The I/O protocol of each module must be executed by interlocking and sequencing control events on the control I/O of modules to meet protocol and time constraints, such as set up and hold times, while meeting the system bandwidth requirements.

In brief, the interface physically links and resolves the differences between the inputs and outputs of communicating modules, while meeting module protocol/time constraints and system performance requirements. It is important to notice that interface modules exercise two levels of control. One is the sequencing of events as defined by the module's protocol, and the other is sequencing from transfer to transfer as defined by the inter-module communication behavior.

The examples described also highlight the main issues of interface design:

a. The specification method for inter-module communication behavior, which becomes the interface functionality and performance requirements.

b. The specification method for the I/O characteristics of an arbitrary module, especially the I/O structure and protocol, and include electrical characteristics.

c. Generation method for the interface module.

All these issues are addressed in this thesis.

1.2 Motivations and Objectives

The previous examples are typical interface modules for a variety of applications. Even for the VMEbus interface which is of medium complexity, these examples highlight the enormous amount of detailed information required to generate these interfaces. This includes local time constraints and precedences between signal transitions, and also system-level functions such as decoding or address generation for block transfers. In addition, generally there may be multiple source and destination modules. Since modules often use different I/O protocols to communicate, system timing does not follow a consistent scheme. A major amount of system design time can be spent on generating the interface circuitry between system modules.

To reduce the complexity of integrating hardware modules, this work explores and develops a design methodology, specification and synthesis techniques for the automated generation of the interface circuitry from a behavioral specification of the interface. Because of the large amount of details and design knowledge needed to design interface modules, the dominating goal of the approach to be presented is to raise the design abstraction to a level higher than the logic or timing diagram level. This means generating the interface module from a behavioral-style specification that is shorter than lower-level structural (logic) specifications, easier to write and to understand, and therefore to change. Previous approaches toward interface design tended to focus on a subset of the three design issues previously discussed, and did not integrate the approach with a system level design methodology and CAD tools.

The ability to synthesize interfaces is crucial to reducing the design complexity of system integration, and to ensure functionality and required performance. It also enables the design of reusable modules and supports modular system design, since it eliminates the need to design multiple modules that perform the same computation but satisfy different I/O protocol constraints which arise in different applications or due to component performance upgrades. Most importantly, it allows a non-expert to integrate hardware without having to understand the complex and detailed timing personalities of each individual module in the system.

1.3 Overview

Interface design is an integral part of a complete system design methodology. This section describes the methodology, shows how interface generation fits into the larger picture, and gives a brief description of an interface design approach that addresses the issues and objectives discussed in the previous two sections.

The system design environment is SIERA which supports the behavioral and physical design of real-time dedicated systems from a mixed behavior and structure description

[Srivastava91a][Sun91]. SIERA extends the concept of a VLSI silicon assembler and compiler to hardware module generation at the board level. Again, the system is composed of components from various technologies. As shown in Figure 1-5, system hardware is produced using a mix of module generators for ASICs [Chu89][Thon89][Rudell88][Shung91] and programmable logic [Yu91], and an IC module library. The module library stores design information about off-the-shelf

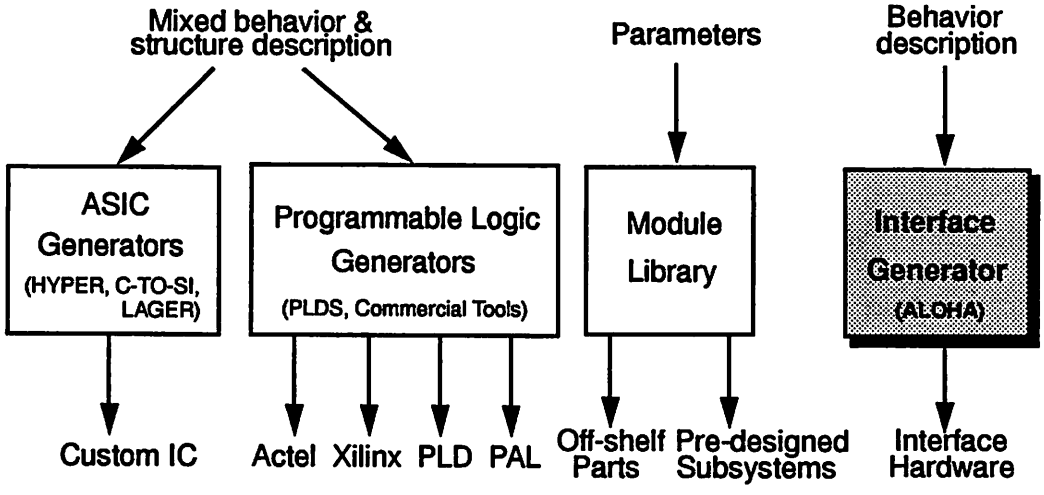


Figure 1-5 : Hardware Module Generation in SIERA

components and pre-designed subsystems, and it plays a central role in the interface design approach. Importantly, it captures information about the I/O structure and I/O signaling protocol, whether asynchronous or synchronous, of each module in the library.

Originally, integrating the ASICs, programmable logic, and library hardware into a hierarchical module or the entire printed-circuit-board was done through manual design. The interface module generator ALOHA converts a behavioral description of the interface module into logic, which is particularly useful at the system level. In addition, it is hierarchical in that ALOHA can integrate components from the three basic groups into a higher level module which can be put back into the module library for reuse in other applications.

Interface design should start with a user specification of the inter-module communication behavior for the particular application and also the I/O protocol constraints for each module involved. From the behavioral description, the interface logic is generated, which is then physically implemented by interconnected IC components such as FPGAs, off-the-shelf ICs and even ASICs. Since the goal is to achieve a high-level design abstraction, the generation process is automated with tools for different levels of abstraction. The design flow is shown in Figure 1-6. It has a high-level design phase and a low-level design phase.

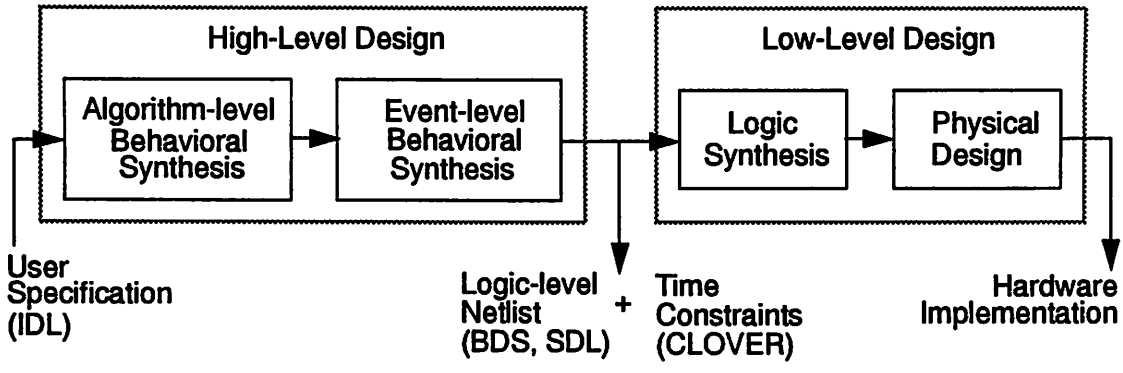


Figure 1-6 : Design Flow for Interface Generation

At the high-level phase, ALOHA provides for the user a high-level input language to describe the inter-module communication behavior, which is the interface functionality at the algorithmic level. The language, called IDL, models the communication as a network of modules that transmit and receive data through ports. The IDL description essentially specifies the temporal and spatial mapping of source data streams to destination data streams in a manner that is independent of the modules' protocols and technologies. The I/O protocols and timing of the modules is stored in the module library, thus hiding these low-level details from the user.

From here, ALOHA synthesizes the interface from the behavioral domain into the structural domain, elaborated in Figure 1-7. Synthesis between these two levels of abstraction is called *high-level* or *behavioral* synthesis. The strategy ALOHA takes is to first translate the user specification

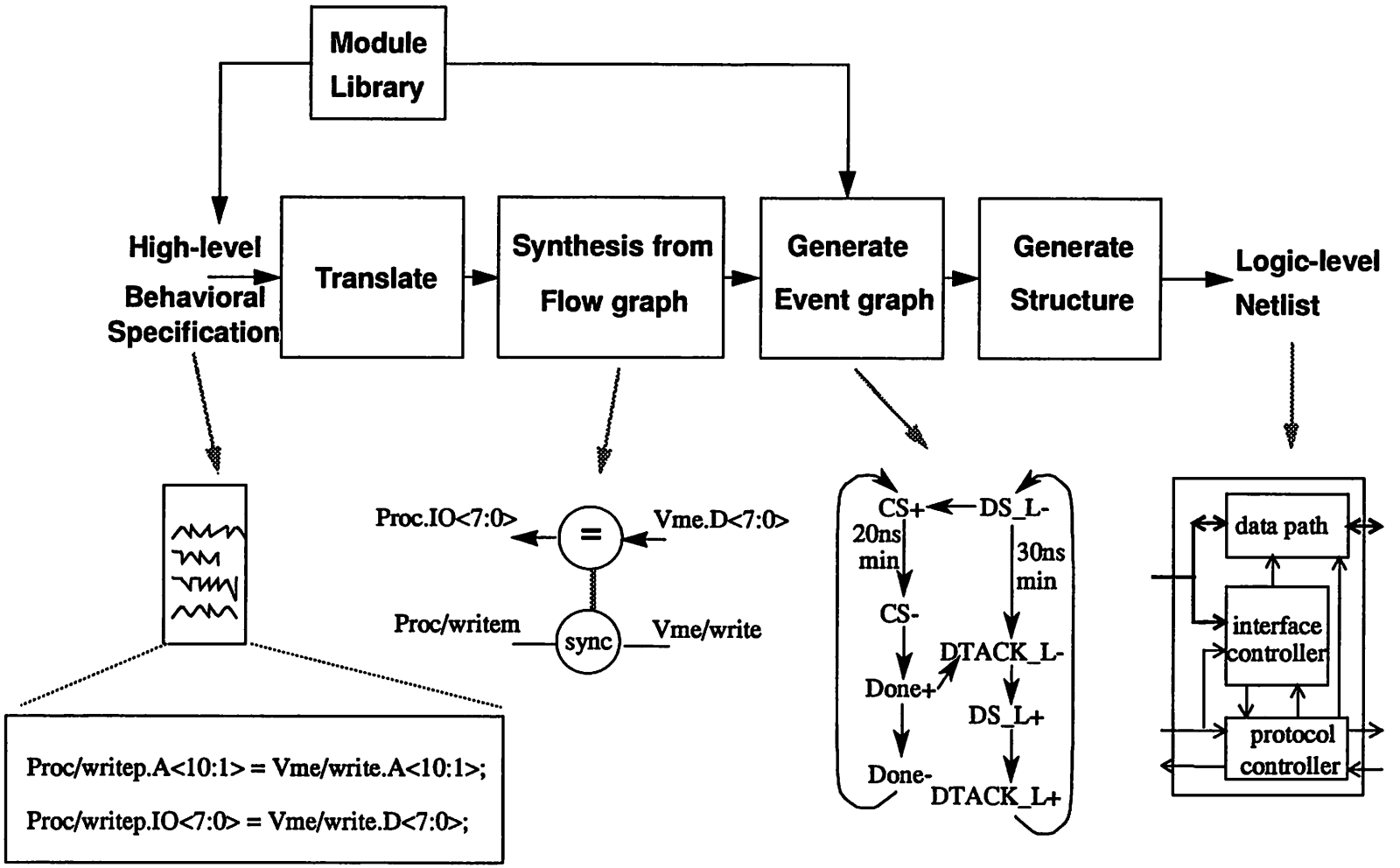


Figure 1-7 : Overview of High-level Design in ALOHA

into an internal representation, and then perform a series of transformations that refine the design specification through two levels of abstraction to get the interface logic. The abstractions are the flow graph representation[McFarland90], which effectively handles datapath and transfer sequencing requirements of the specified communication algorithm, and the event graph [Borriello87][Chu87a] (also called signal transition graph or STG), which effectively represents I/O events and time constraints. The event graphs corresponding to the I/O protocols for the communicating modules are merged according to the data flow described in the IDL description to produce a event graph for the entire interface module. From these two types of graphs and a generic interface template, ALOHA produces the register-transfer-level logic, netlist, and timing description for the interface module, in the BDS[Segal88], SDL[Richards], and ASTG [Moon91] and CLOVER[Doukas91] formats, respectively.

Finally, the low-level design phase takes the logic and timing description into physical implementation. Since logic synthesis techniques are at a fairly mature state, at this point the design system links in existing logic synthesis and verification tools to perform logic optimization and mapping to appropriate technologies. The logic level description can then be implemented with FPGAs or PLDs, off-shelf components specialized for system interconnect, and even ASICs using the module generators shown in Figure 1-5. Commercial tools are used for board placement and routing[Racal], bringing the integrated modules into the physical domain.

It is this top-down methodology that distinguishes this work from the previous works in interface synthesis and overcomes some of their limitations.

1.4 Summary

The following chapters describe previous work related to interface design and then the design approach and techniques used in this work. Chapter 2 summarizes previous work and shows their relationship to the work here. Since the original contributions of this work are made in the high-level design phase, a complete description will be given of the module library, the specification language, and the behavioral synthesis approach. Chapters 3 and 4 focus on the module library and language, respectively, while individual phases of interface synthesis are elaborated in Chapters 5 through 8. The low-level design phase and synthesis results are presented in Chapter 9. Chapter 10 concludes with a summary of contributions, and gives future research directions in both the high-level design and low-level design phases, as well as extensions of this work to support a wider range of applications.

CHAPTER 2

PREVIOUS WORK

The work described in this thesis applies a number of concepts from hardware system design, language and graph based specifications, behavioral synthesis, and synthesis of asynchronous control logic. This chapter presents previous work related to interface generation and highlights the features that are useful in addressing the design issues and in achieving the high-level abstraction goals discussed in Chapter 1.

The approaches toward the specification and synthesis of interface circuits in the past has developed in two different directions. This dichotomy stems from the different perspectives taken by researchers from the asynchronous design community and researchers from the behavioral synthesis community that typically implement algorithms using a synchronous design style.

2.1 Synthesis for Asynchronous Design

Over the past 30 years, there has been considerable research on synthesis for control logic using an asynchronous design methodology. Currently, as system complexity and physical size grows, the ever-increasing clock skew problem has given this research area a new thrust. The emphasis of this has been on communication timing and synchronization that is independent of a global clock. Interface generation typically starts from an event level abstraction. The control logic is produced using either graphs or flow tables (a finite state machine representation) or is directly mapped from a CSP-like language. Graph based algorithms are most common and are reviewed in this section. In these methods, a graph is used to represent the precedence and timing relationships between input and output signal events [Chu87a].

Section 1.1 of Chapter 1 discussed the three primary functions of an interface module. Of these three, the protocol synchronization problem has been most extremely analyzed, and with less results in handling the data path requirements and control from one transfer to the next. So, these techniques are applicable to synthesis of protocol converters, but do not address the needs of more complex interface modules.

Synthesis from Signal Transition Graphs

The work in synthesis of self-timed control logic based on the *signal transition graph* (STG) provides formalisms for specification and synthesis [Chu86][Chu87a]. The STG is derived from petri-net theory to represent behavior between input and output signal transitions. The approach assumes that all system modules, to be interconnected with the synthesized control logic, use the 4-phase handshake protocol. Also, the control logic can have unknown but finite gate delays and zero wire delays.

The input specification is also a STG that represents the interlocked handshake events between communicating modules. The STG representation for the handshake between a source and

destination is illustrated in Figure 2-1. The node is a signal transition (not value), such as the rising edge of Req1, denoted by Req1+. An edge shows the precedence between two signal transitions. So, the edge labeled "e1" indicates that the Req1 rising edge from the source causes the Req2 rising edge at the destination, essentially initiating the transaction. An underlined node is an output event from the interface. The entire graph is the sequence of protocol events that synchronize the transaction. The graph is cyclic, since the protocol repeats for the next transaction.

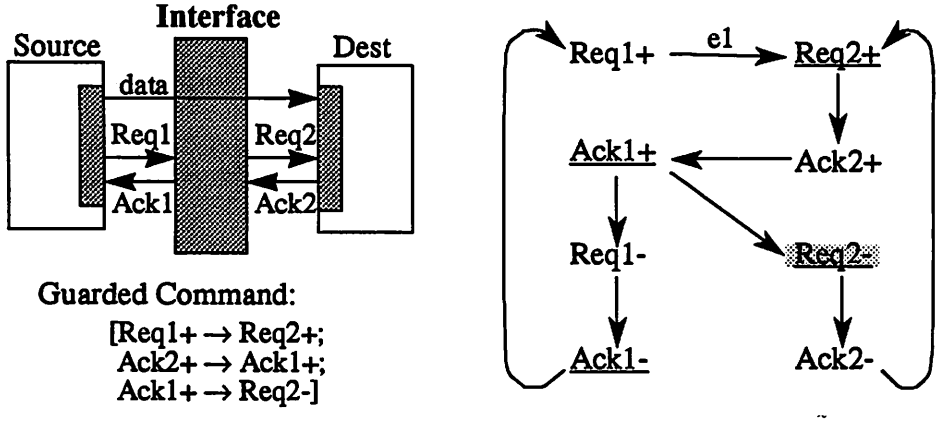


Figure 2-1 : Signal Transition Graph for a Handshake Communication

The input STG is checked for properties that display potential deadlock and other incorrect behavior (called hazard) in the implementation. An important property is called *persistency* and it specifies that when a transition is *enabled* (but has not occurred yet), the occurrence of some other transition does not disable it (or cancel it) [Chu87a]. The highlighted node, Req2-, in Figure 2-1 is a non-persistent transition, since it is enabled by the Ack1+ stimulus and may be disabled if Ack1- occurs before the Req2 output signal reacts to Ack1+, due to a long circuit delay between the two signals. The STG structure is then manipulated to eliminate all of these fatal properties. In Figure 2-2, the insertion of an additional edge "e2" makes the transition persistent, eliminating a possible hazard.

From the "good" STG, boolean equations are synthesized by transforming the STG into a lower-

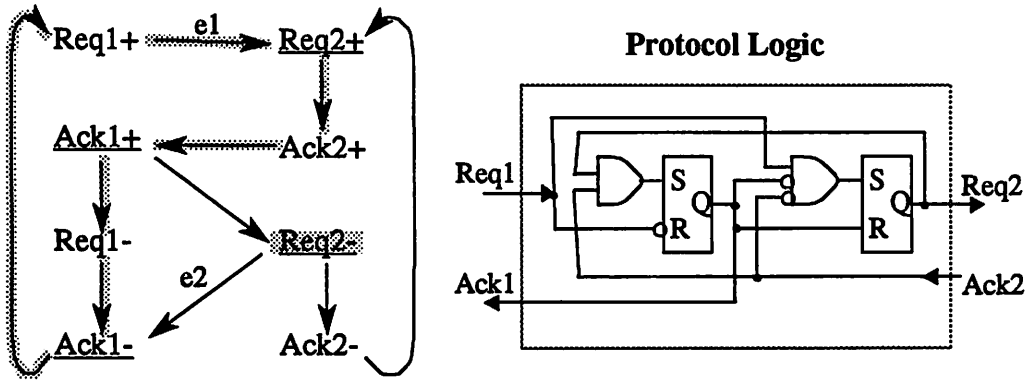


Figure 2-2 : Persistent STG and Logic Implementation

level state transition graph. The resulting circuit implementation is insensitive to the internal delays. Overall, synthesis is a set of formal transformations performed on the entire STG.

After the basic theory of STGs was developed, further work explored design abstraction and performance issues [Meng89]. All system modules used the 4-phase protocol, and this assumption allowed a user specification in which the handshake protocol is implied. In the Guarded Command specification [Meng89][Martin86][Dijkstra75], the basic statement is a precedence relationship(s) between source and destination handshake events, essentially abstracting only those precedences that interlock the communication. Figure 2-1 shows the guarded command for that example.

The full STG is then constructed from the guarded command and the individual handshake STGs. Synthesis proceeds using the techniques discussed above. Importantly, it was recognized that the STG structure itself can reveal the expected performance of the circuit to be synthesized. The cycle in the STG with the longest delay determines the performance limit. The delay is accumulated from the individual delays of the edges in the cycle. In the example of Figure 2-2, the critical cycle is highlighted, and the transfer throughput is the inverse of the cycle delay time. Other works related to performance evaluation from graphs is also presented in [Ramamoorthy80].

The winning feature of the STG method is the *representation* itself. The STG concisely and

effectively captures both event precedence and concurrency. The accompanying theory provides methods for checking if potential deadlock and hazard conditions exist, and for estimating the communication throughput, before the circuit is even synthesized. Although the STG can represent arbitrary protocols and time constraints, the synthesis technique does not work with these assumptions. This may not be a limitation within ASICs since the chip designer can determine the internal timing strategy, but it is a serious problem when using off-the-shelf components where the system designer has no control over the I/O specifications.

There also have been many other works that employ the STG or a similar representation, but they use alternative synthesis methods in an attempt to overcome some of the above limitations [Vanbekbergen90][Nowick91][Moon91][Lavagno91]. The synthesis techniques mentioned so far are based on formal transformations. However, there are also techniques that use compiler methods which take a specification based on the CSP language and map it directly to control logic using a set of built-in rules [Martin86][Berkel91].

Synthesis from Event Graphs

Synthesis using STGs is theoretically founded and insists on strict assumptions about the circuit environment and internal delays. The *event graph* is a very similar representation, but the assumptions are relaxed and synthesis based on it has used heuristic rule-based techniques [Borriello87]. In contrast to the STG methods, the communicating modules can use a mix of synchronous and asynchronous protocols that have time constraints. Given the gate delays of the implementing technology and the event graph for the interlocked protocol events, the control logic is produced to meet the required function and time constraints.

The event graph for the example of Figure 2-1 is shown in Figure 2-3. There are two differences between the event graph and signal transition graph. First, the event graph is acyclic, using start and end nodes which mark the first and last events of the protocols. Second, in addition to the precedence edge, there is an additional type of edge - the time constraint edge - which only

specifies a time constraint between two events and does not imply causality like the former. The time constraint edge is shown with a dashed edge, and the precedence edge may be weighted with time constraints and delays. The differences are superficial, since the STG can easily be extended into an event graph, and the event graph can easily be reduced into a STG.

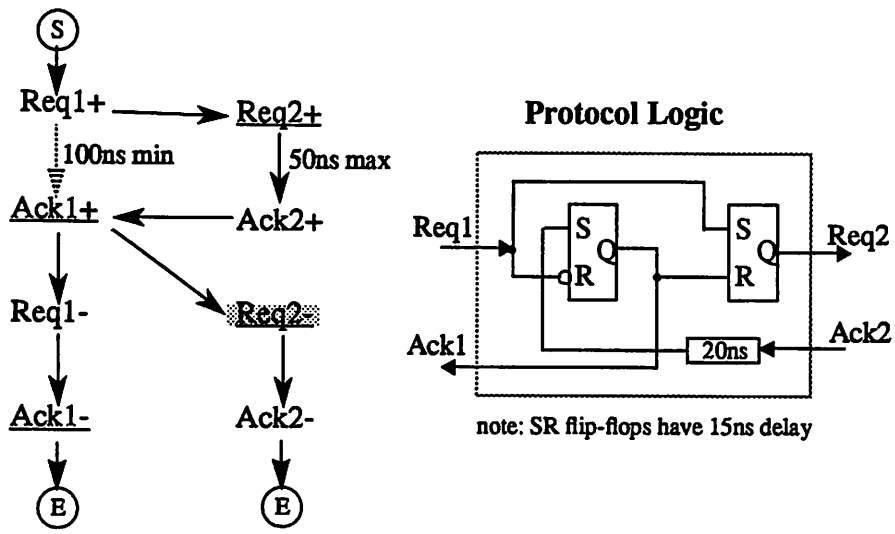


Figure 2-3 : Event Graph for a Handshake Communication

Janus [Borriello87] is a synthesis tool that generates the control logic in three phases. First, all output signals are driven by an SR flip-flop, and the events that cause a rising edge on the output are logically ANDed at the Set input of the flip-flop, while events that cause the falling edge are ANDed at the Reset input. This mapping is carried out for all output control signals until a "skeletal" circuit is constructed, also shown in Figure 2-3. Since the event graph models events on data signals, synthesis can construct simple datapaths from wires, latches, tri-state buffers and multiplexers. In the second phase, path delays are extracted from the logic network which is then checked for timing violations and hazard conditions. Janus adjusts the delays along the path causing the error, by selecting another gate from the library (same function but different delay) or inserting additional delay elements. In the example, the 20ns delay element is inserted to enforce the "100ns min" time constraint, because circuit delays are not long enough to meet it. It is during

this phase that problems like non-persistency are detected and fixed. However, the synthesis technique does not have the formalism to check for deadlock. Finally, Janus performs combinational and sequential logic minimizations on the "corrected" circuit to produce the "optimized" circuit.

In summary, Janus compiles a first-pass circuit from an event graph through local transformations, and then it detects and fixes timing violations and hazards in subsequent passes. The strongest feature of Janus is its ability to synthesize for time constraints. In contrast, the STG-based methods reverse that order; the initial STG is checked for hazards and corrected, and then the circuit is synthesized using global transformations. Its strongest feature is the representation and underlying theory that allows the intended I/O behavior and performance to be verified before synthesis. From a high-level design perspective, it is desirable to combine the best features of both.

2.2 Synthesis for Synchronous Design

For over a decade, there has been research in behavioral synthesis that starts from an abstract algorithmic specification of the system and creates the logic structure at the register-transfer level (RTL) [McFarland90][Walker91]. The research from this community has also made contributions toward interface specification and synthesis. The algorithm is usually specified textually with a hardware description language (HDL), and it describes the mapping of input data sequences to output data sequences [Hartenstein87], with little - if any - constraints on the internal structure. Historically, behavioral synthesis has its roots in the area of processor design. The methods use the synchronous design methodology where system timing relies on a global clock. Also, they were developed specifically to serve the data path and sequential control needs of a processor design. The specification and synthesis methods reflects these roots.

Not all HDLs have facilities to describe I/O event and timing behavior. The ISYN synthesis system [Nestor86] does provide the ISPS language [Barbacci81] for describing function and

extensions for entering local timing constraints. The CADDY system also allows timing constraints and checks them for consistency before synthesis [Camposano86]. These amount to a description of the event sequencing and timing in a linear program, which does not naturally expose concurrency like the STG or event graph.

The first step of synthesis is compiling the specification into an internal representation suitable for automated techniques. Usually, the representation is a graph that represents the control and data operation dependencies in the specification. These graphs are called control/data flow graphs or just *flow graphs*. Figure 2-4 illustrates a generic flow graph. A node is a data operation such as an addition, while a data edge shows data dependency and a control edge (dashed) represents sequential execution. I/O signal events are considered as an input or output operation [Nestor86], like the output node in the example. Time constraints are expressed as weighted control edges between nodes. Some synthesis systems such as HAL [Paulin89] and ELF [Girczyc84] require the designer to enter events and time constraints directly into the flow graph.

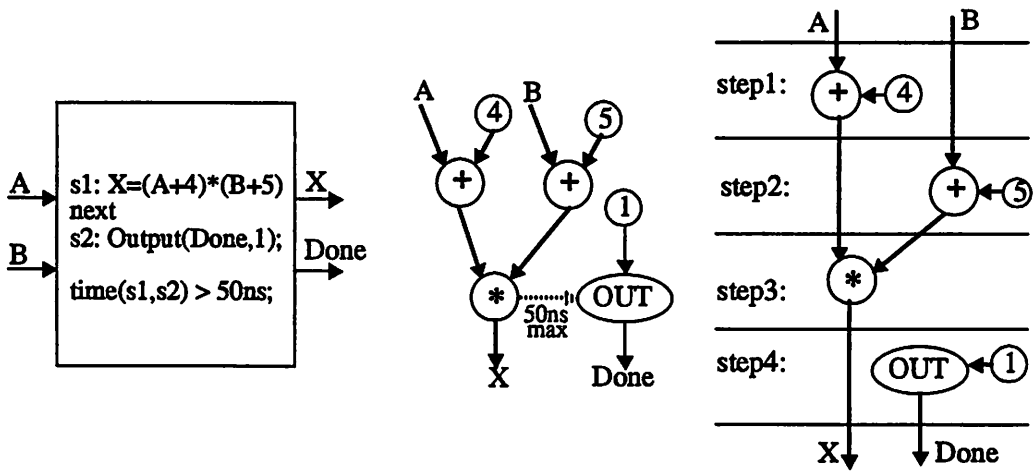


Figure 2-4 : Example of a Flow Graph and Scheduling

In the next core steps of synthesis, data operations are put into a control step for execution, or *scheduled*, such that data and control dependencies are not violated. A control step lasts one clock cycle which must be less than any time constraint specified. The ISYN and ADAM [Hayati89]

systems provide enhanced scheduling techniques for event operations and time constraints. Scheduling is done in conjunction with resource *allocation* which involves assigning the operations to hardware elements such as functional units, storage and busses. In other words, scheduling gives data operations a time to happen, and allocation gives them a place to occur. Their combined goal is to meet constraints on throughput, area, hardware utilization or even power. The last synthesis step is mapping, or *binding*, the operations to a network of datapath hardware from a library and a state machine description for the controller.

The flow graph representation and synthesis methods have been highly successful for processor-oriented applications. Here, data computation and high level control are the main functions, and, in contrast to STG and event graph methods, there is only a few event precedences and time constraints. Considering the three primary functions of an interface module, behavioral synthesis techniques can be most effective for handling the datapath requirements and control from one transfer to the next.

One of the main drawbacks of synthesizing interface modules using flow graph techniques is that it compromises the potential communication performance. Protocol events usually occur asynchronously. However, the flow graph method treats events as operations and they too are schedule into synchronous control steps to meet time constraint. This precludes events from occurring as soon as permitted by the protocol, and the resolution of the clock must be finer than the smallest time constraint.

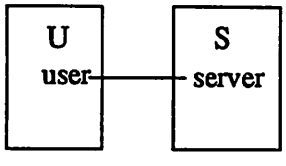
2.3 Other Specifications and Representations

In addition to the presented research related to interface generation, there are others that solely address specification or representation. The specifications model the system as a set of communicating processes. The representations are graphical, mixing data flow and event behavior.

In specification, the SLIDE language was developed specifically for describing I/O event and

timing behavior [Parker81]. The protocol behavior is described by a linear sequence of event statements which may be conditioned on an internal variable or external signal. The sequence may also include subsequences inside a loop. No clock signal is implied. A *wait* construct supports specification of time constraints between consecutive statements. Unlike the other HDLs presented, SLIDE is supported by a simulator and verification techniques, making the language useful for designing I/O protocols. However, the language makes modeling synchronous behavior tedious because every event must be described with respect to an explicitly stated clock edge, and text entry using a language makes modeling concurrency difficult.

From the distributed systems community, the SSCS specification language can abstract the inter-process communication behavior, as well as model the low level details of I/O event behavior [Bochman83]. Because the high-level specification used in ALOHA was most influenced by this specification, a SSCS example is given in Figure 2-5 which shows the abstract communication between an user and a server process (or module). The notation of interest is the expression $(user/request)_i.X$. It specifies that a process U communicates through its port named *user*, which employs a *request* type of interaction and delivers the parameter value *X*. The *user* port I/O structure and protocol have not been defined yet. The index *i* numbers individual interactions in a sequence of interactions.



process type *server* is
  user: service-access;
  constraint
    $(user/response)_i.Y = FUNCTION((user/request)_i.X)$
end *server;*

Figure 2-5 : An SSCS Specification

The key contribution of this work is that it recognizes the need for a step-wise refinement of the initial inter-process specification into detailed data types, port structure and protocol of the

interaction, although the work does not say how to do the refinement. In contrast, the behavioral synthesis methods presented in the previous section mixes the abstract algorithmic behavior with the more detailed I/O events, and attempt to synthesize directly from that one representation. This has made synthesis difficult, because behavioral synthesis methods were developed from the algorithmic perspective and then forced to work with the event level abstraction [McFarland90].

Other research in graphical representation are the OE-graph [Borriello88a] and DE-graph [Whitcomb92] representations. Both mix data operation and event behavior and have formalisms lacking in the flow graphs described in the previous section. The representations distinguish between two types of nodes: the data operation node and the event node. In the OE-graph representation, operation nodes are executed when specified input events arrive, and they produce output events or wire values according to a function described with a C++ (programming language) routine. The OE-graph representation is supported by the simulator OEsim [Amon91a]. It is easier to use for describing arbitrary I/O protocol behavior compared to the SLIDE language. However, not all OE-graphs are because of the freedom in specifying the node behavior which may not have an implementation. In the DE-graph, the operation nodes must come from a library of elementary nodes. The representation is supported by an underlying data base system. It can express only a subset of the behavior that OE-graphs can, but will have a hardware implementation that can be synthesized from an arbitrary and consistent graph, although no particular synthesis method is proposed.

2.4 Summary

This chapter has described much of the existing work related to generation of interfaces. Specifications ranged from graphs to languages, some of which mixed data flow and event behavior and others focused only on one aspect.

Relevant synthesis efforts are summarized in Figure 2-6. The vertical axis of the figure represents

behavioral abstraction level, which is the relationship between inputs and outputs of the communication. The horizontal axis indicates the types of structural components that realize the desired communication, which includes the datapath functional units, multiplexors and registers, the logic for system control flow, and the logic for protocol conversion. The figure illustrates the goal of this work, labeled ALOHA, that raises the level of design abstraction and performs a complete synthesis of the interface. From a *high-level* description of the system communication, ALOHA automatically creates the complete detailed interface implementation. Previous synthesis

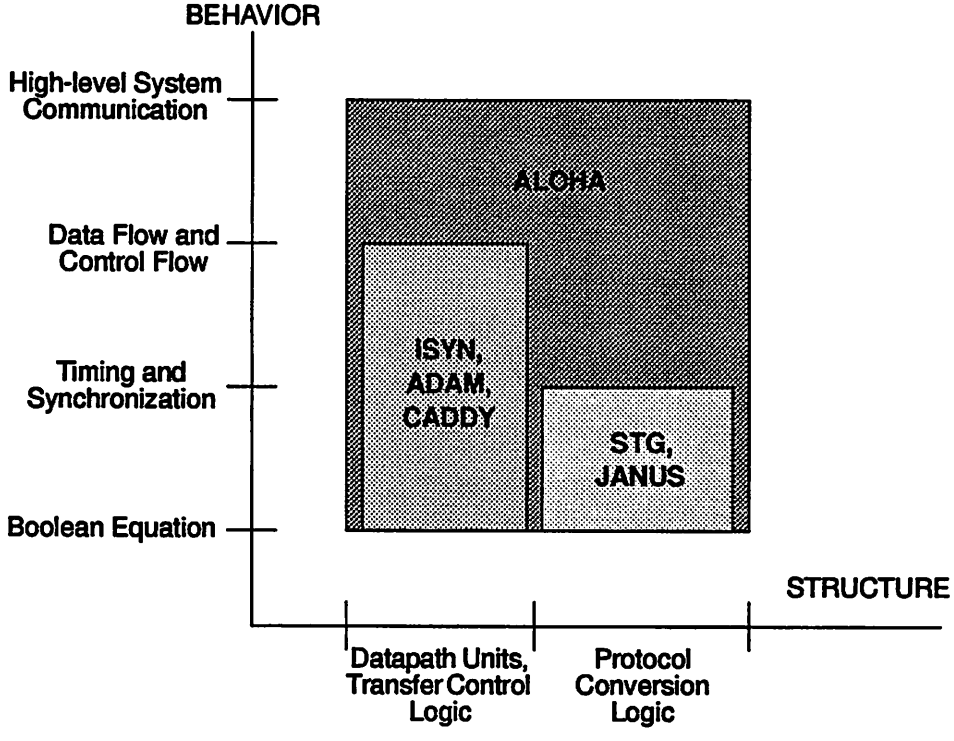


Figure 2-6 : Previous Synthesis Works and ALOHA

work has emphasized producing a partial solution based on just the flow graph or event graph representation, and has required a detailed specification of the I/O event and timing behavior from the designer. Enumeration of such protocol details does not provide enough automation for system design using various technologies.

This work draws from specification and synthesis concepts in asynchronous design, behavioral synthesis and distributed systems research, while introducing new ones to achieve its goal. An interface specification language, based on SSCS [Bochman83], is especially developed for high-level input that is independent of the module protocol and technology details through the use of a module library. For automatic synthesis, the design specification is refined into the flow graph and event graph representations. Concepts from behavioral synthesis are applied toward the data path and high-level control needs, while signal-transition graph and event graph methods are applied toward the I/O protocol and timing requirements.

CHAPTER 3

MODULE LIBRARY AND I/O SPECIFICATION

The design of a complex hierarchical system begins with IC components as the primitives. From these elements, multi-component modules are formed, and then these submodules in turn can be interconnected to generate higher level modules. A key strategy for minimizing the design effort is to form a collection of components and modules that can be reused among several applications. Just as VLSI chip design has reusable cell libraries, board-level system design also benefits from a module library. The modules range from single components to entire subsystems, such as a TTL logic component and a uni-processor subsystem.

The purpose of the module library is to hold information and models, related to each member module, to drive tools for module generation and simulation. The SIERA module library, introduced in Chapter 1, originally provided information about the module's algorithmic, structural and physical characteristics, such as combinational logic descriptions, I/O signal names, netlist of submodules and package types [Srivastava92].

These types of information are sufficient for driving the ASIC generators, programmable logic

generators, and board place and route tools. But, it is not enough to generate the interfaces, since this task requires knowledge of the I/O protocols that communicating modules use. This type of information is usually found in data sheets and specification manuals. To effectively drive the ALOHA interface generator, the *protocol* information was introduced in electronic form into the library. The module library supports interface generation during the high-level specification and the synthesis step as shown in Figure 3-1, repeated from Chapter 1 to highlight the role the library plays in interface design.

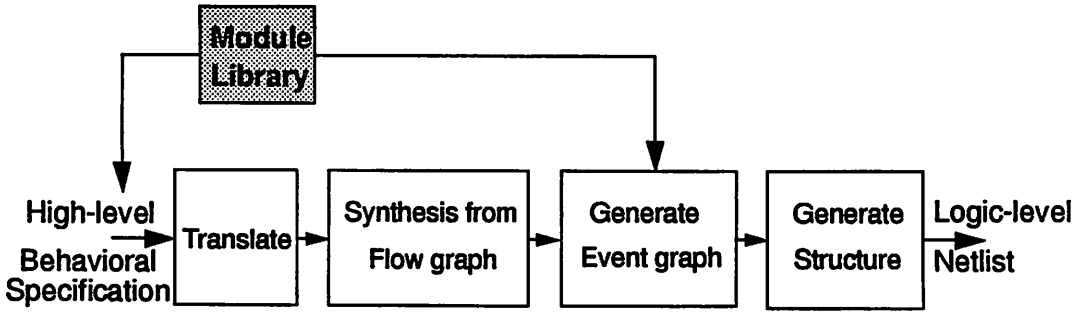


Figure 3-1 : The Module Library and Interface Generation

This chapter first presents an overview of the module library, followed by background on module I/O communications. This is provided to give an understanding of how I/O protocols are specified and represented in the module library, presented in the subsequent sections.

3.1 Overview of Module Library

The module library in SIERA contains structurally specified board-level hardware modules. To make the modules reusable, parameters are specified with a library module and given specific values when an instance is generated for a particular application, in effect customizing the module. SIERA actually carries three types of libraries meeting low-level to high-level design needs. A package library contains physical information about the geometries of various board level

packages such as through-hole DIP and PGA packages, and surface-mount packages such as SOIC or PLCC. The primitive library holds single IC chips, discrete components and connectors such as fiber optic receivers and transceivers, RAMs, digital signal processors, and even A/D and D/A converters. The subsystem library has hierarchical modules made up of the elements from the first two libraries and other subsystem modules. An extensive description of the SIERA module library and accompanying module generators is given in [Srivastava92].

A relatively simple example of a subsystem module and the parameter facility is a static RAM module. It is described by parameters corresponding to the memory size (in bytes) and the word width (in bits), a netlist of individual memory components and line drivers, and floor plan coordinates specifying how individual RAM components are tiled to generate the memory array. A complex example is a uni-processor module based on a TMS320C30 DSP core, multiple types of local SRAM memory, and a dual-port RAM for host communications, and described by a parameterized structure. Its parameters are the size of each memory type, number of memory mapped I/O slave devices, and the memory and I/O address map. The address decoder in the uni-processor is a single FPGA submodule created by the programmable logic generator [Yu91], and its parameter is the name of a file containing the combinational logic description. So, parameters also provide a way to mix behavior into the structural specification.

Overall, a parameterized module can be described in the behavior, structure and physical domains and with its simulation model. Algorithmic behavior is specified with either the applicative data-flow language SILAGE [Hilfinger85] or a C-like sequential language [Thon89], while combinational behavior is described with the BDS language. The structural netlist and physical information is textually specified in the SDL language. VHDL models or THOR functional models support simulation.

Among all this information that comes with a library module, the module I/O signals (structure) and I/O protocol (behavior) are the ones of interest for interface generation. The I/O structure is

specified by the SDL language, and the protocol is modeled with event graphs and specified with the AFL description language [Rabaey90], and the languages provide a user-interface into the OCT standard database [Harrison86].

Table 3-1 shows a subset of the library modules and their I/O description. The modules range from a D/A converter primitive to a frame buffer subsystem and a VME system bus. The size of a module's I/O structure is indicated by the data and address bus width, and the number of control signals. Modules employ one or more I/O protocols, depending on the type of communication to be carried out. But, only one protocol can be exercised at a point in time, such as the read or write protocol. Also shown is the number of I/O events, event precedences and non-zero time constraints that compose the signaling protocol. Most importantly, the numbers convey a sense of the enormous amount of low-level detailed information the library captures about each module. In fact, for the more complex modules, the numbers represent only a small fraction of the total amount. The table demonstrates the effort in design entry that a designer is saved from each time a library module is reused.

3.2 Module I/O Communications

So far, this thesis has covered the function of an interface, design goals and general interfacing techniques. Before progressing into the details of the module library implementation and design techniques, this section formally looks at the way modules communicate through their I/O ports.

The port is a collection of signals linking the module to its environment, hence the term *I/O port.* This really means more than just a set of wires. A port has three layers: the mechanical specification, the electrical specification, and the functional or logical specification. The mechanical specification is related to the physical aspects of the port and includes geometrical dimensions, connectors for mounting the module within a system, and also strength and reliability. The electrical factor governs the requirements that must be met by the input and output signals

Table 3-1 : Sample Modules from the SIERA Library

Module Name	Description - I/O Structure and Protocol
DAC811	Digital to analog converter - 12-bit data, 6 control signals. Digital port enabled by a load pulse with separate write and select signals. 14 events, 13 precedences, 2 time constraints.
AD7870	Analog to Digital converter - 12-bit data, 4 control signals. Digital port uses 3-wire four-phase handshake; issues interrupts. 8 events, 10 precedences, 4 time constraints.
Am7968	Transmits parallel data over a serial fiber-optic link - 8/9/10-bit data, 4/3/2-bit command word, 2 control signals. Digital port uses a 2-wire four-phase handshake. 8 events, 9 precedences, 7 time constraints.
IDT7134	4K x 8 Dual-Port static RAM module- 8-bit bidirectional data, 12-bit address, 3 control signals. Read and write accesses controlled by a chip enable signal, and acknowledges upon access. 20 events, 23 precedences, 6 time constraints total.
Framebuffer	Two bank video frame buffer based on dual-port SRAMs - Primary access port: 8-bit data, 13-bit address,4 control signals. Read and write accesses use four-phase handshake with bank select. 10 events, 13 precedences, 3 time constraints each. Interrupt port: Separate 5/32 input/output data bus, 3 control signals. Input and output transfers use four-phase handshake. 8 events, 13 precedences, 5 time constraints.
VMEbus	VMEbus standard system bus (IEEE P1014/D1.2) - 32-bit data, 32-bit address,7 address modifiers, 1 write control, 3 data transfer control,10 interrupt control, 11 arbitration control, 6 misc. control signals. Read and write data transfers: 3-wire four-phase handshake. 36 events, 38 precedences,22 time constraints. Interrupt acknowledge and pass protocols: combination of the four-phase handshake and daisy-chain token passing. 30 events, 51 precedences, 19 time constraints. Bus arbitration protocol: combination of handshake and daisy-chain token passing. 6 events, 7 precedences, 2 time constraints.

such as load capacitances, voltage levels for logical zero and one, and noise margins. The functional specification deals with the I/O structure and the sequencing and timing of signal events that the module port and its environment must comply with to ensure a proper exchange of information. This signaling convention is called the *I/O protocol.*

Overall, the port provides a black-box view of the module. Its specification is only concerned with what happens on the boundary of the module rather than what is happening internally, and compliance with its specification guarantees proper communication between the module and its environment. The other modules communicating with the module of interest may be a processing module (if it uses a compatible port specification) or an interface module if communication is not straight-forward. Of the three layers, the interface generation methods discussed here focus on the protocol specification. The following discusses issues of port I/O structure and protocols in more depth.

3.2.1 I/O Structure

The I/O structure of a module port consists of a set of input and output signals. Some of these signals convey the information of interest. Other signals implement an I/O protocol to synchronize the information transfer between the module and its external world. This leads to a natural partitioning of the port signals into two logical subgroups. Not discussed here is a third group of signals that provide DC power to the module, usually supplied by a system bus.

The first is called the *information* group, consisting of signals such as data, address and command. What is important about these signals is their value or meaning represented by their logic levels. Once transferred, they are used by modules to evaluate functions, invoke a response or change the internal state. Figure 3-2 illustrates the I/O structure of a TAXI optical receiver module and the VME system bus module. The TAXI module uses data lines only, while the VMEbus has data, address and command (the WRITE_L signal).

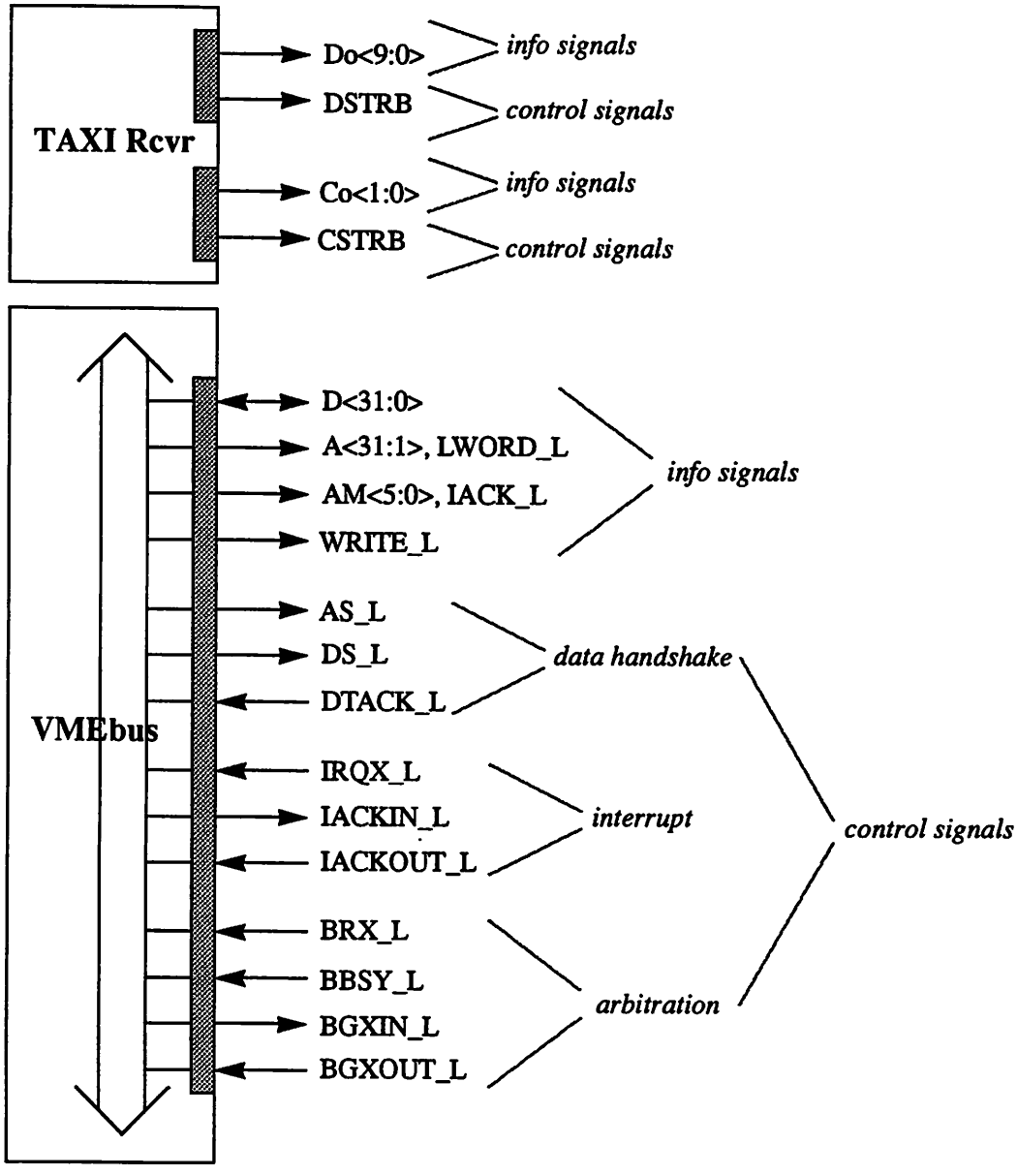


Figure 3-2 : I/O Structure of the TAXI Receiver and the VMEbus

The second is the *control* group. These signals carry out the I/O signaling events according to a protocol to synchronize communications. The important characteristics of these signals are the signal transitions and the timing between them. Three common types of control signals are data handshake lines for the timing of a data transfer, the interrupt lines for timing of requests for

services, and arbitration signals for resolving conflicts over access to a shared resource. Practically all modules use the data handshake lines, like the two modules in Figure 3-2. Interrupt and arbitration control is usually confined to microprocessor modules, peripheral devices and microprocessor-based system busses [Stone82], such as the VMEbus.

Possibly more than in any other area, economics influences the I/O structure used by a module port [Clements87]. Ports that use parallel data and address ports with dedicated lines for interrupts and arbitration have higher costs than a port employing a serial data line and no hardware support for interrupts and arbitration. Of course, the parallel port will provide higher communication bandwidth especially when used in local communications only, as opposed to system communications. The NuBus standard attempts to a compromise between communication performance and cost by providing a parallel set of information lines on which connected module ports multiplex data and address [NuBus].

Modules can have more than one I/O port, of course, since the port itself is just a logical grouping of all the module I/O signals. An example is the TAXI optical receiver with two separate ports for the C data and the D data, shown in Figure 3-2. However, ports may share information signals but not control signals. Another example is a dual-port RAM. Each independent port has its own data and address information signals, and control lines to time the read and write accesses. The read and write access have different signaling and timing requirements, and this example shows that a module port may use more than one protocol for different types of communications with the environment.

3.2.2 I/O Protocols

To communicate with the system, a module synchronizes each information transfer through its port using an I/O protocol. The protocol defines the sequence of I/O events and time constraints that must be obeyed during a transfer, and also possibly data formatting and representation rules.

The I/O protocols accompanying a library module can come from many sources. The I/O protocols used by off-shelf components in the primitive library can be found in TTL, memory and component data books from the commercial vendor. They are depicted with timing diagrams and tables of time constraints (also called switching or AC characteristics). In the case of ASICs and FPGAs, the port structure and signaling protocol is defined by the designer of the circuit, rather than the manufacturer of the technology, and typically documented with timing diagrams too. The I/O protocols of subsystems formed from these various components are also user-defined. In the last two cases, the system designer can determine the cost and performance requirements of the module I/O ports, and customize the protocol to optimally meet the needs.

In general, the I/O protocols in use today fall into two broad classes of I/O protocols [Thurber72], categorized by how the timing of a transfer is controlled. At one end of the spectrum are *synchronous* protocols which use a clock control signal to time each data transfer. At the other end are *asynchronous* protocols which use handshake control signals to interlock module port timing with its environment.

Figure 3-3 illustrates the typical synchronous protocol used by an ASIC. All information signals stabilize within a set-up time of a rising (or falling) clock edge, and must remain stable within a hold time. A transfer lasts one clock period, with the rise event marking the beginning and the fall event marking the end of the transfer. In this way, communication with the module occurs in lock-step advancing with each new clock cycle. Synchronous protocols are easiest to implement of the three, because they require only a clock oscillator to control the timing. They work well for local communications because they offer fast transfers on physical interconnects that have negligible clock skew, provided that each module can communicate at the designated clock speed.

If a module communicates to a mix of other modules with varying operating speeds, the asynchronous I/O protocols offers more efficient use of the interconnection bandwidth. Figure 3-4 shows the fully asynchronous VMEbus protocol for data/address write access. The key feature of

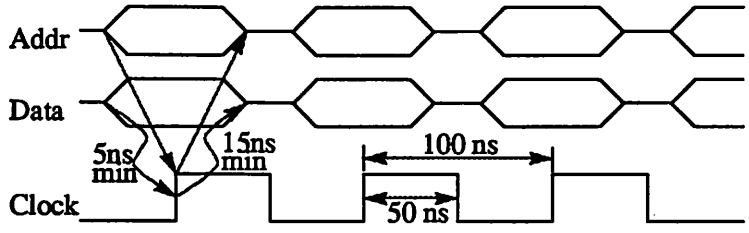


Figure 3-3 : Synchronous I/O Protocol (ASIC)

this class of protocols is a pair or request and acknowledge signals, where an event on the former marks the beginning of a transfer, and an event on the later is issued to acknowledge that the transfer has been successful. The time between the request and acknowledge event is the access time of the module, and in this way fast and slow modules can each participate in the communication at their own speed. The VMEbus request signals are AS_L (address strobe) and DS_L (data strobe), and DTACK_L is the acknowledge. Information signals, like data D and address A, are defined to be stable with respect to the handshake events data strobe DS_L and AS_L, respectively. The VMEbus protocol uses a 4-phase handshake protocol where the request and acknowledge signals must be reset before the next transfer can occur, leading to two overhead events per transfer. Other examples of asynchronous protocols are the 4-phase handshake MultiBus standard [Multibus] and the 2-phase handshake Futurebus standard [Futurebusa][Futurebusb]. Asynchronous protocols are generally used by system busses or general-purpose processors which communicate with modules of various speeds, and where the physical interconnect length and capacitance is non-negligible. Compared to synchronous protocols, asynchronous I/O protocols are non-trivial to implement.

Of course there are variations within the two classes, and also and hybrids between the two. The semisynchronous protocols use handshake signals synchronized to a clock signal. The NuBus protocol is an example of this.

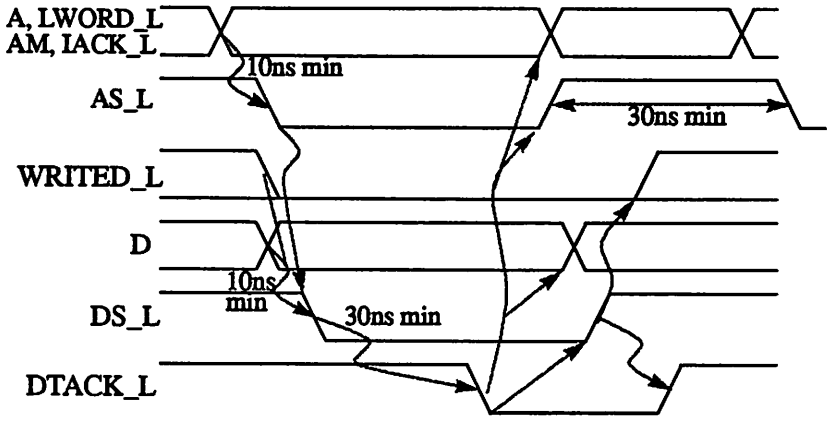


Figure 3-4 : Asynchronous I/O Protocol (VMEbus)

I/O ports can be master or slave ports. Master ports initiate the communication by issuing requests on an output request signals. Slave ports participate in the communication when requested, since their request signal is an input. There are modules that are both master and slaves, but not simultaneously. An example is a DMA controller. It is a slave to the host processor when it is configured with the start address and block size, but it assumes a master role when it accesses the main memory and the peripherals on the system bus.

The following section describes the specification method for any class I/O protocols in the module library.

3.3 The I/O Protocol Specification Method

As shown in Figure 3-1, ALOHA generates interface modules from a behavioral description. The behavior consists of the interface functionality and a set of system design constraints. The functionality is the high-level communication pattern between interacting system modules and is described by the IDL language presented in the next chapter. The design constraints are the low-level event sequencing and time constraints imposed by I/O protocols of the interacting modules.

They are already captured for the designer in the module library using the representation and specification methods presented below. It should be mentioned here that the design constraint designers usually think in terms of is the data transfer throughput. This constraint is actually determined by the protocol time constraints. If a source module uses a synchronous protocol with a 10MHz clock rate to transfer data to an asynchronous destination port, then the interface generated must fulfill this constraint, and must provide a 10MHz transfer rate also assuming that the destination port can operated at this rate.

3.3.1 Protocol Specification with Event Graphs

An I/O protocol is characterized by signal events and precedences between events, including timing relationships. Events can also occur concurrently. In Figure 3-4, the VMEbus write protocol has a high degree of concurrency, although the timing diagram does not effectively reveal it. When the DTACK_L falling edge occurs, the remainder of the handshake and changing of information values can occur concurrently before the next transfer. So, a protocol representation must model synchronous and asynchronous I/O events, precedences, concurrency and time constraints, and must also be suitable for simulation and synthesis. The directed cyclic graph meets these requirements, and a hybrid of the STG [Chu87a] and event graph [Borriello87] was chosen to model I/O protocols. Throughout this thesis, this hybrid will be called an "event graph" because it is a succinct term.

There is a one-to-one to correspondence between the timing diagram and the event graph representation. Figure 3-5 demonstrates this principle with a static RAM component. The I/O port structure and timing diagram that describes its write access protocol was obtained from a data sheet. A signal event, such as the chip select falling event (CS_L-), corresponds to a node in the graph. Event precedences correspond to a graph edge. For example, the address must become valid before chip select falls, and this set-up relationship between the two events is modeled with edge e4 in the graph. A time constraint, such as the 5ns minimum address set-up time, specifies an

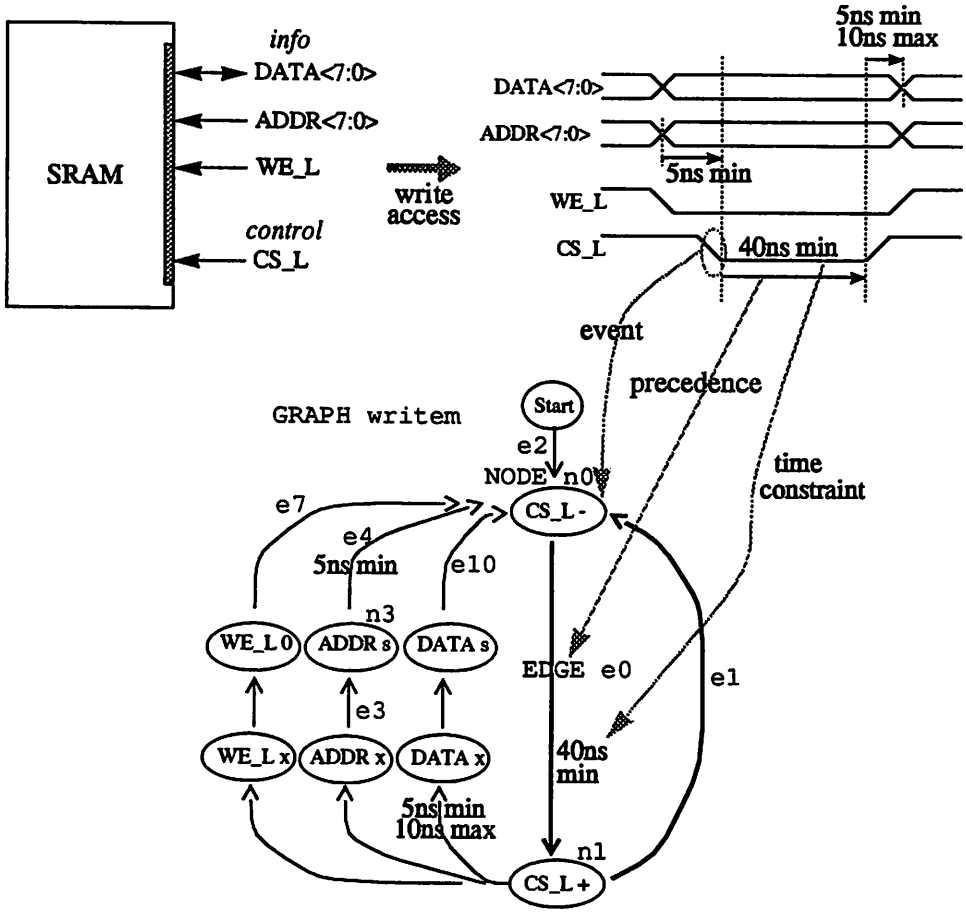


Figure 3-5 : I/O Protocol and Event Graph for SRAM Write Access

interval of time in which an event precedence can occur. The time constraint is represented as a weight associated the edge. In the event graph, multiple edges branching from a node model the concurrency between subsequent events. Tracing the cycles in the event graph, the write I/O protocol is described as follows: data, address and write information signals stabilize; then the chip select signal asserts on the falling edge; followed by its rising edge which completes the chip select active-low pulse; and finally the information signals become invalid.

A module port may use multiple protocols. The static RAM uses a different I/O event sequence for

the read access compared to the write access. This requires one event graph for the read protocol and another for the write protocol.

To specify an I/O protocol in the module library, a library developer can manually translate the timing diagram into an event graph, and then use the AFL graph description language to textually enter the graph into the library. The AFL language basically describes the structure of the graph with a list of nodes, edges and their connections and properties. This is analogous to a structural netlist. Besides a language, another method for specifying I/O protocols is entering them with a timing diagram editor, like WAVES [Borriello88b]. For IC components, this amounts to duplicating the timing diagrams as they appear in the manufacturers' data sheets. However, for ASICs and subsystems where the I/O protocol is not pre-defined, it is better to design new protocols using event graphs, because the representation can clearly and cleanly show sequential and concurrent behavior.

The following describes the event graph representation, including its node and edge elements, and corresponding AFL specification in more detail. The complete AFL syntax for event graphs is provided in Appendix A.

The Graph

To illustrate the AFL specification, the AFL description corresponding to the SRAM event graph (in Figure 3-5) is shown in Figure 3-6. Only a portion of the complete description is shown, but enough to give a concrete understanding of how the specification is used in the module library.

The graph is cyclic since the protocol repeats for each new RAM access. The special node labeled "start" points to the initial event of the protocol, marking the beginning of a transfer. In the AFL specification, the event graph is given a unique name that serves to identify the protocol it is representing, such as "writem" in the example. The RAM is a slave device, and the model_name "slavep" reflects this. If it were a master, then the "masterp" model_name would be used. The

```
/* I/O Protocol for SRAM write access */

(GRAPH (NAME writem)
   (MODEL ( (model_name slavep) ))
   (ARGUMENTS ( (port sram) (timeunit ns) ))
   (NODELIST
      (NODE
         (NAME n0))
         (ARGUMENTS (
            (signal CS_L)
            (value f)
            (direction in)
            (valid DATA)
            (valid ADDR)
            (valid WE_L)
         (IN_CONTROL (e1 e2 e4 e7 e10) )
         (OUT_CONTROL (e0) )
      )
      (NODE
         (NAME n3)
         (ARGUMENTS (
            (signal ADDR)
            (bitvectwidth 8)
            (bitvectbase 0)
            (value s)
            (direction in) ))
         (IN_CONTROL (e3) )
         (OUT_CONTROL (e4) )
      )
   )
   (CONTROLLIST
      (EDGE
         (NAME e0)
         (CLASS control)
         (ARGUMENTS ( (min 40) ))
         (IN_NODES (n0) )
         (OUT_NODES (n1) )
      )
      (EDGE
         (NAME e4)
         (CLASS ctrlinfo)
         (ARGUMENTS ( (min 5) ))
         (IN_NODES (n3) )
         (OUT_NODES (n0) )
      )
   )
)
```

Figure 3-6 : AFL Description for the SRAM Write Protocol

name of the module port that uses the described protocol and the time units that the time constraints are expressed in is given in the ARGUMENTS field of the specification.

The I/O Event

The node represents a signal event, whether the signal is of type information or control. In the AFL specification, the event node is given a name that uniquely identifies it among other nodes listed, like "n0" for the CS_L- event. A node description contains characteristics about that event, held in the arguments field. The important ones are the signal name on which the event occurs, the transition value, and direction of the signal, as shown for the chip select falling event. Events on bus information signals, such as DATA, will also have the bus width and least significant bit index specified. The default value is one bit and zero index, respectively.

Signals are allowed to take on the transition values listed in Table 3-2, adapted from the SCALD value system [McWilliams80]. Control signals may only take on the rising, falling and high-impedance values, while information signals are allowed to take on stable, unknown, high-impedance, or specific bit-vector values.

Table 3-2 : Signal Value System

Value	Meaning
+	rising, signal changing from low to high
-	falling, signal changing from high to low
s	stable, signal is stable at some bit vector value
bit vector	signal is stable at a specific bit vector value
x	unknown, signal value is unknown or don't care
z	high-impedance, signal is at high-impedance state

The optional characteristic, valid, of an event names the signals that are stable, or "valid", upon the

occurrence of the event. In the example, the chip select fall event indicates that address, data and the write enable signal to the memory are valid. So its node specification specifies this with the "valid" argument. There is also the invalid characteristic that is the counterpart of the valid argument.

Event Precedences and Time Constraints

Event precedence corresponds to a graph edge. For instance, edge e0 in Figure 3-5 shows that the CS_L+ event must follow the CS_L- event. In the AFL specification, an edge also has a unique name, like "e0" for this precedence relationship.

Time constraints or event circuit delays between events, represented as weights attached to edges, are specified in the edge ARGUMENTS, such as the 40 ns minimum CS_L active-low pulse width. Currently, the AFL specification supports the min-max-avg delay model where a time constraint is expressed with a minimum to maximum bound, and an optional average time between the bounds. If none are specified, then the bounds take on zero minimum and infinite maximum by default. More complex delay models can be used, such as statistical models, but they are not currently supported by design tools. Sometimes, when a module with an internal clock uses an asynchronous protocol, a time constraint or delay is a function of the clock period. In this case, it is specified as a string in edge ARGUMENTS. For example, instead of 'min 40ns', the constraint appears as 'min "3*clock"', where clock is the clock period.

The time constraints and delays shown so far were associated with precedence edges. Sometimes, the I/O protocol will specify a time constraint between two events that do not have a precedence relationship. To express this, the event graph representation provides a second type of edge called the timing edge. Figure 3-9 shows dashed timing edges.

Lastly, the structure of the event graph (connectivity between nodes and edges) is described by the IN_CONTROL and OUT_CONTROL field of the node specifications, and the IN_NODES and

OUT_NODES field of the edge specifications. In summary, the AFL specification emphasizes the structure of the event graph, and conveys behavioral properties through the ARGUMENTS field.

3.3.2 Examples of Event Graphs

The full event graph for the ASIC of Figure 3-3 is shown in Figure 3-7. The interesting feature is how periodicity in a synchronous protocol is specified. The clock control signal has a 50% duty cycle and a period of 100ns. This is implemented with minimum and maximum time constraints of equal value associated with the edges that directly connect the clock rising and falling events. The event graph also shows that the synchronous protocol can be thought of as a special case of a handshake protocol that uses only the request control signal with periodic rise and fall events.

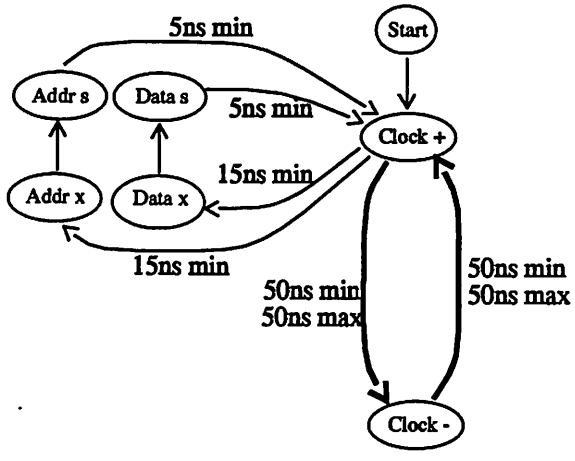


Figure 3-7 : Event Graph for a Synchronous ASIC Protocol

Figure 3-8 illustrates the complete event graph representing the VMEbus write protocol from Figure 3-4. It has a loop for the handshake between the data strobe DS control signal and the acknowledge DTACK signal, and another for the handshake between the address strobe AS and the DTACK. The graph also shows the high degree of concurrency between event after the acknowledge event, DTACK-, has occurs. This allows the remainder of the protocol to complete

quicker than if events occurred sequentially.

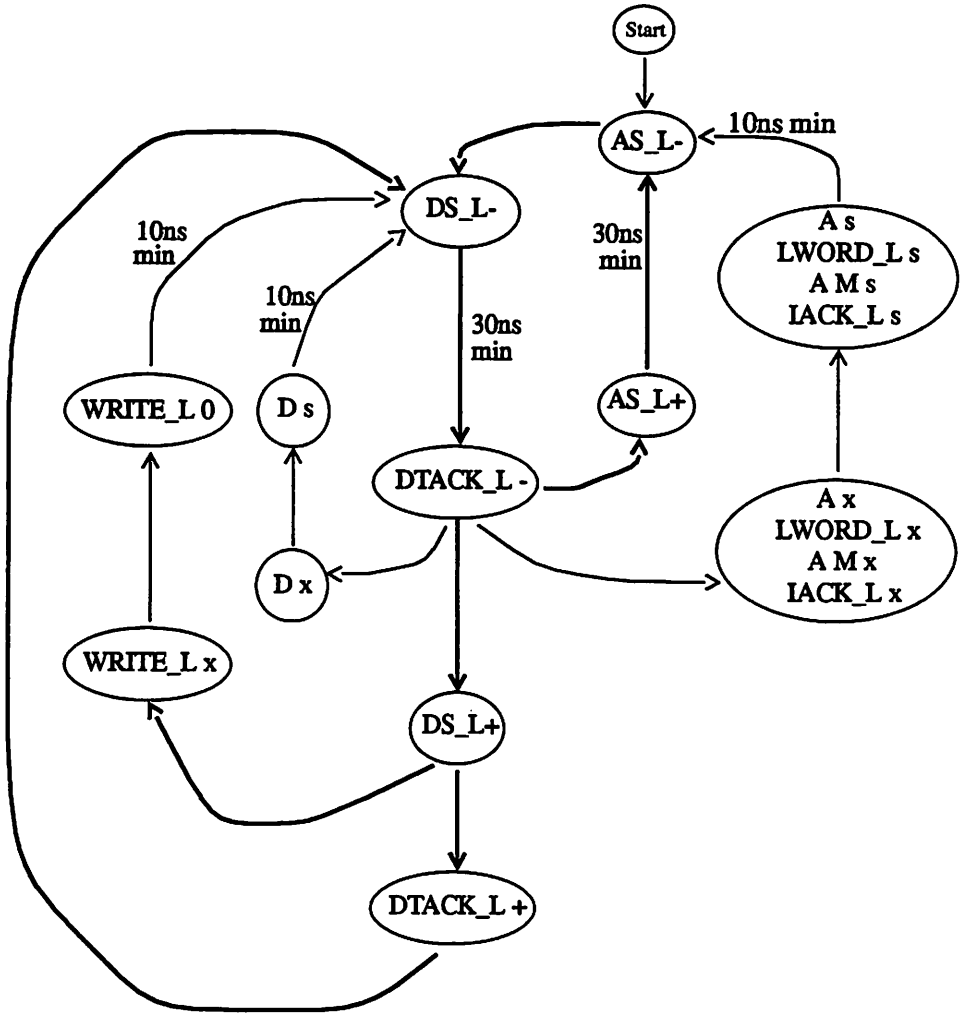


Figure 3-8 : Event Graph for the VMEbus Write Protocol

The VMEbus event graph shown is just one of 5 events graphs that cover the VMEbus data transfer, interrupt and arbitration protocols. All the others, except one, are equally or more complex than the one shown. Evidently, this graph illustrates that manual entry of the I/O protocols into the module library can be error-prone, motivating the need for consistency checking and simulation.

3.4 Consistency Checking and Simulation

When an AFL specification of an event graph is entered into the module library, the library user-interface also performs an automatic check of the event graph description for connectivity and consistency errors. In addition, protocol behavior can be simulated using the OEsim simulator [Amon91a]. Simulation provides a way to check the correctness of the I/O protocol specification. Unlike consistency checking, it allows the designer to evaluate the performance of an I/O protocol before the logic or physical hardware is designed.

Consistency Checking

Consistency checking catches syntax errors in the AFL specification entered by the library developer and also fatal behavioral errors evident from the event graph structure and properties.

Errors in the specified protocol behavior are detected in two ways by the user-interface. The first uses the graph structure to find errors, checking connectivity between nodes and edges in the event graph. Obvious errors are nodes that have no input edges or no output-edges, resulting in events that never happen or in dead-end. Edges must have exactly one input node and exactly one output node, and this is also checked.

The second check inspects node and edge properties to detect consistency errors. Signal names appearing in a node specification must have been declared in the I/O port structure. Time constraints specified with an edge must have increasing minimum, average, and maximum values. At the graph level, the time constraint between all pairs of events must not conflict. This means that the minimum time along any path between the two events must be less than the maximum time along any of the other paths, as illustrated in Figure 3-9.

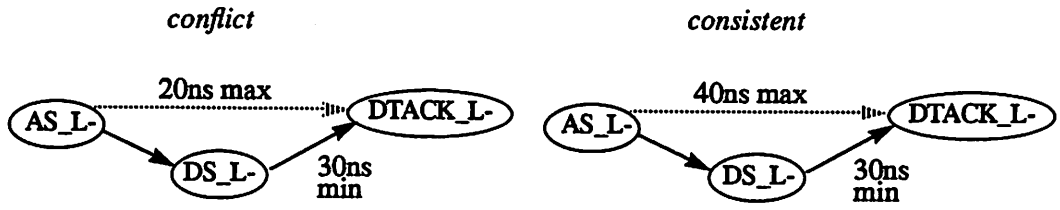


Figure 3-9 : Time Constraint Consistency

Simulation

Simulation of an I/O protocol validates its correctness and performance. For example, the library developer may enter the wrong time constraint value for the event graph of a off-shelf module. In another example, the I/O protocol being defined for a custom module in development is simulated for possible deadlock conditions and also to evaluate the potential performance. In either way, the I/O protocol is verified before it is entered into the module library. I/O timing behavior is simulated using the OEsim simulator [Amon91a]. The simulation model is based on the OE-graph representation, discussed in Chapter 2. Experience with OEsim has shown it to be a valuable tool for designing I/O protocols.

The model is directly generated from the event graph AFL specification. Currently, this is manually done, but it can be automated using the following rules. Illustrated in Figure 3-10, the OE-graph elements of interest for modeling I/O protocols are the event node, the operation node (graphically a box), and the precedence edge that represent dependence between an event and an operation node. Protocol behavior is modeled as operation nodes that are triggered by incoming events to produce outgoing events within the module's circuit delay. The circuit delay may not be part of the protocol specification, and is distinguished from a time constraint. Figure 3-10 shows how a simulation model is generated from the event graph for the SRAM write protocol in Figure 3-5. An event node in the event graph maps to an event node in the OE-graph. Its incoming

precedence edges collectively translate to a operation node in OE-graph that produces the event of interest when all the predecessor events have occurred. OE-graph edges are inserted to the model to connect the predecessor events nodes to the operation node and the event node under consideration. The operation node is an instance operation node model from a simulation library.

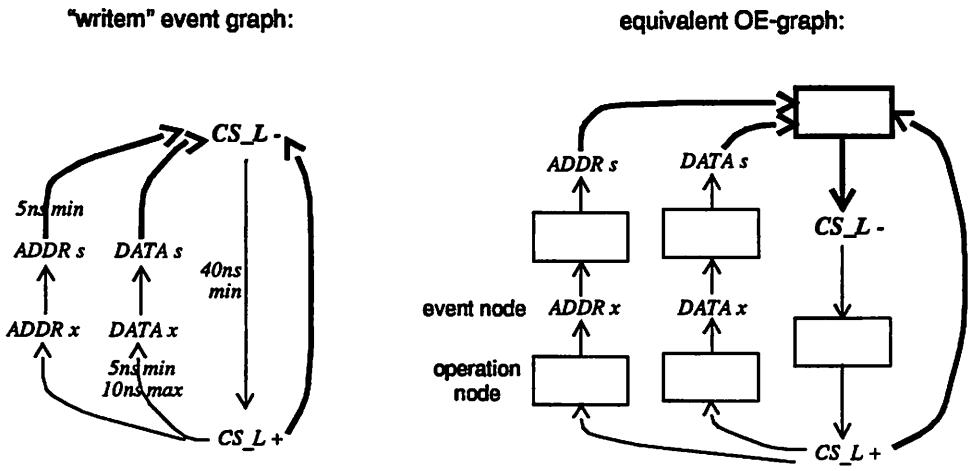


Figure 3-10 : OE-graph Model for SRAM Write Protocol

In addition to the constant delay model, OEsim can simulate delays based on the uniform distribution model, where delays are specified within a minimum and maximum bound. Unlike many simulators, OEsim has time constraint checking and reporting capabilities. Time constraints specified in the event graph are translated into the OE-graph format using a library of constraint functions that have been developed already. The functions include minimum, maximum, set-up and hold time constraints and the function parameters are the minimum and/or maximum time bounds. During simulation, OEsim will continuously monitor timing between events and flag any violations, inserting warning messages into the simulation results.

The user initiates simulation with an input signal event, such as CS_L+. During the simulation run, the user provides the input events while the model generates events occurring on output signals.

3.5 Extensions to the Module Library

The module library captures the low-level details of module protocol, structure and physical properties into a central database, hiding these details from the designer. It supports the principle of design reuse and modularity. This is the key impact that the module library has on system design.

As mentioned in Section 3.2, electrical characteristics are part of the module specification. This information can also be entered into the library, and then used to drive module generators including the interface generator; however, this has not been implemented so far. The electrical characteristics of a module vital to interface design are the I/O signal levels and capacitances. The first is concerned with logic levels, noise margins and tri-state or open-collector properties. Interface generation can use this information to determine the appropriate type of line driver or receiver for converting voltage levels and driving load capacitances.

In Section 3.1, the library approach described assumes that the modules have pre-defined protocols. This makes obvious sense for off-th e-shelf components, but other modules like ASICs, FPGAs and subsystems are parameterized to generate an instance when the module is called from the library. Conceptually, in this case, the I/O protocol does not need to be pre-defined and can be determined when the module is generated. To support this approach, a library of protocol templates can be added to the module library. This fourth type of library would contain a minimal yet comprehensive set of parameterizable I/O protocols from which the designer can choose for the particular application. Figure 3-11 illustrates this concept with a synchronous protocol. The information signals are Data and Addr, and the control signals are CLK and RDY. The RDY signal allows the I/O port to indicate which clock cycles transmit valid data. The parameters are the clock period, duty cycle, and the width of data and address.

Important issues in using protocol templates are, first, deciding which protocol to use and, second, what the parameter values should be. These decisions will be determined by communication

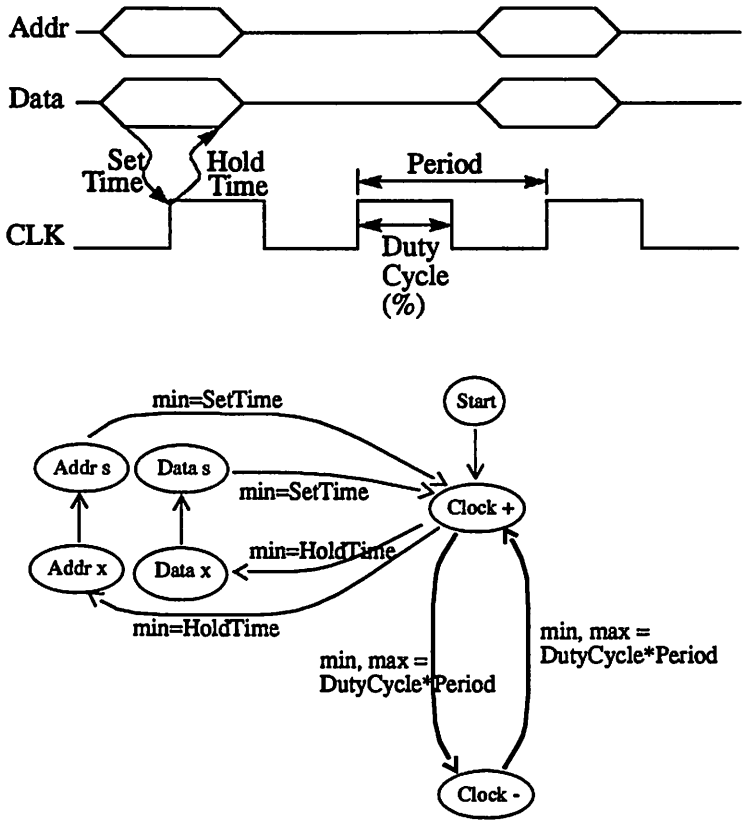


Figure 3-11 : Template for a Synchronous I/O Protocol

performance and cost requirements, such as the desired throughput, timing characteristics of the interacting modules, and the size of the I/O port. Protocols can be simulated with OEsim, as discussed in Section 3.4.

CHAPTER 4

HIGH-LEVEL SYSTEM SPECIFICATION

High-level design starts from a specification of the desired hardware behavior which consists of the functionality and the design constraints. For interface generation, the constraints are the event sequencing and time constraints imposed by the I/O protocols of interacting system modules. The functionality corresponds to the inter-module communication pattern, concentrating on the global data and control flow requirements. While the module library captures the detailed protocol constraints, the input language describes the desired functionality. The IDL language provides the designer an entry point into the ALOHA interface generator.

The goal of the specification method is to abstract system communication to a level that is independent of the module protocol and technology details, while being easy to use and expressive. At this level, what is known is the input and output signals of interacting modules and their functional relationship. There is no knowledge about the internal structure of the interface module that implements the communication. The key issue centers on "what the interface is doing." This chapter focuses on how the IDL language describes this to achieve a high-level design abstraction. As shown in Figure 1-7 of Chapter 1, the interface generator transforms the

high-level specification into an internal representation and then synthesizes the internal structure and execution sequence, which are the subject of subsequent chapters.

This chapter first presents the underlying model that defines the semantics of the input language. Then, an introductory example demonstrates the key features of the language. Afterwards, the language features are discussed in more detail, and applied to more examples. Finally, the chapter addresses simulation issues at the system level.

4.1 Specification Model

Describing system communications requires modeling the data and control flow behavior between interacting modules. Data flow refers to the routing of information from source modules to destination modules, including any transformations to be performed during the transfer. Control flow refers to the sequencing of transfers, such as exercising concurrent, sequential, conditional or looped transfers. The specification method models system communication as a network of modules that transmit and receive information through their ports. Information streams from the source ports are spatially and temporally mapped to information streams at the destination ports. The interface is the actual entity that implements the mapping; so, modeling the inter-module behavior amounts to specifying the interface function. This model is further described below.

4.1.1 I/O Transaction

As discussed in Chapter 3, a module port is a source or destination of information or both. Information can be data, address or any other value to be communicated. An *I/O transaction* is the transmission or reception of a single information value (or packet) between the module port and its environment, obeying the port protocol, of course. The time for one transaction to execute is considered the basic unit of time in a sequence of transactions. This is illustrated in Figure 4-1 for a processor initiating a sequence of read and write transactions. During read transactions, the

processor is the source of address and data values, while it is the source of address and the destination for data values during write transactions.

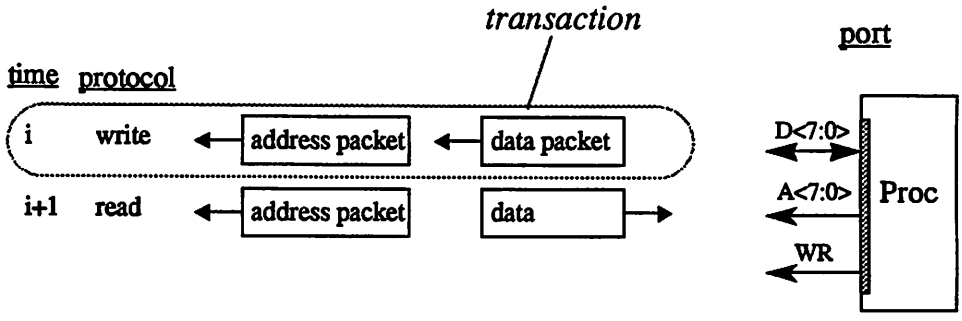


Figure 4-1 : Processor I/O Transactions

The transaction concept is adapted from the port interaction concept used in the SSCS specification model [Bochman83] and the notion of sample sequences in digital signal processing [Oppenheim89]. The SSCS model does not explicitly speak of I/O protocols, but it promotes the abstract notion of *interaction types*. This has been applied toward I/O protocols to meet the special needs of interface specification. The useful feature in the sample sequence model is allowing information to come from a current or past samples.

The I/O transaction requires identifying the module port, an information value, and the I/O protocol used. This means the name of the port, the name of the I/O signal that passes or accepts the information, and the name of the protocol. It does not need to know about the I/O events and timing details of the protocol being used.

4.1.2 Inter-module Transfer

The I/O transaction only models what happens at one end of the communication, but information transfers involve at least two ends. The specification model maps a stream of information values from source transactions onto a stream of values for destination transactions.

Each mapping to a destination value corresponds to a inter-module transfer. The interface implements the sequence of mappings. Figure 4-2 illustrates an example of an interface that transfers data from a source port to a destination port.

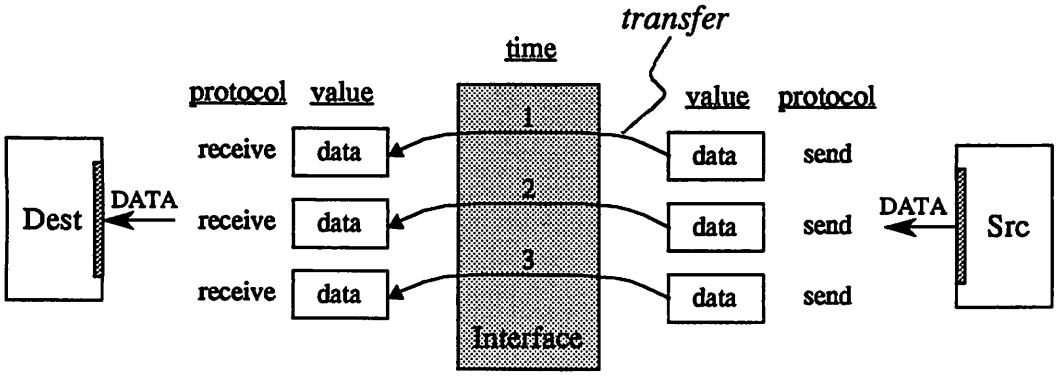


Figure 4-2 : Inter-module Transfers

Beyond the basic transfer, a transfer may involve a transformation of the source values, such as arithmetic or logic operations, before it is routed to the destination. Source values also can be from a current or past port sample. These comprise the data flow behavior. The interface can also map source sequences to destination sequences in a concurrent fashion, conditioned on some information value, or iteratively for a finite amount of times, implementing control flow behavior.

As the interface facilitates a sequence of transfers, it also executes a lower level sequence made up of interlocked protocol events for each transfer. The IDL specification models the higher level of behavior, while the event graph models the lower level of behavior. The protocol details are hidden from the designer by the module library.

4.2 An IDL Specification Example

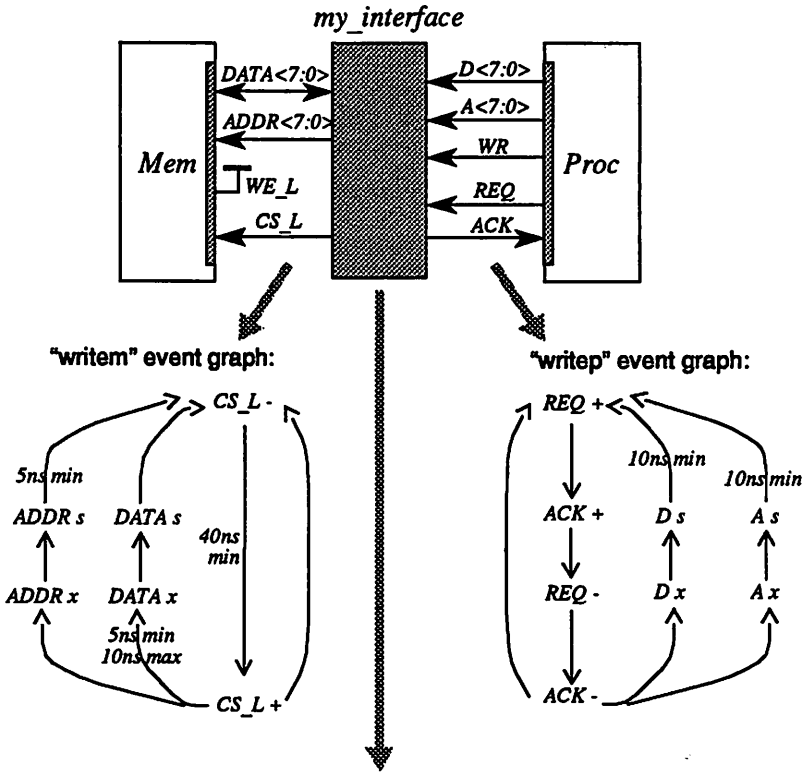
Since inter-module communications tends to be control dominated rather than data computation dominated, it is natural to use a procedural language as the input specification.

Accordingly, the high-level specification developed especially to describe inter-module communications is a procedural hardware description language, called IDL (interface description language).

IDL has many features that are found in other hardware description languages. The inputs and outputs of the described module are declared, followed by a description of the behavioral relationship between the outputs and inputs. The special feature of IDL is that the behavioral description is independent of the protocol details. Relevant languages investigated before developing IDL can not achieve this level of abstraction. The VHDL simulation language can express communication behavior at the desired level. However, for synthesis, VHDL is tedious to use and results in a much longer description due to more overhead compared to the equivalent IDL description, defeating the concise and easy to use goal. Also, since only a small subset of VHDL is relevant to describing interfaces and can be synthesized from, it would be necessary to define a VHDL subset which in effect is IDL. The simulation strategy is to translate the IDL specification into a VHDL model, discussed in Section 4.5.

This section provides an example to demonstrate the high-level specification method for interface behavior. It introduces the key IDL specification elements, which are the port declaration, the I/O transactions and the inter-module transfers. A complete treatment of the language is given in the following section.

Figure 4-3 shows illustrates a simple example of a processor that writes into a static memory module. To simplify the example, the memory write signal is hardwired to the logic one level. The protocol and time constraints that the interface must meet is shown in the memory and processor event graphs, which are captured in the module library. The IDL specification from the designer has two parts: the port declarations followed by the specification body. First, in the specification, all the modules involved in the communication are declared with the PORT statement. These modules are instances of modules chosen form the module library. For example, the memory *Mem*



IDL specification:

```
DESIGN my_interface
  PORT PROCESSOR Proc;
    SOURCE Proc.D<7:0>, Proc.A<7:0>, Proc.WR;
  ENDPORT;
  PORT SRAM Mem;
    DEST Mem.ADDR<7:0>;
    BIDIRECT Mem.DATA<7:0>;
  ENDPORT;
  ROUTINE main;
    IF (Proc.WR EQL '1') THEN
      Mem/writem.ADDR<7:0> = Proc/writep.A<7:0>;
      Mem/writem.DATA<7:0> = Proc/writep.D<7:0>;
    ENDIF;
  ENDROUTINE;
ENDDESIGN my_interface;
```

Figure 4-3 : IDL Specification for a Processor to Memory Write Interface

is an instance of the SRAM library primitive described in Figure 3-5 of Chapter 3. In addition, signals used in the specification body and their direction are declared within the PORT declaration. In general, multiple instances of a library module can be listed in one PORT declaration. This is useful for communication involving a bank of modules, such as a bank of static memories. In that case, the instances would be uniquely named, such as *Mem1*, *Mem2*, *Mem3*, etc.

In the specification body, denoted by ROUTINE, the interface functionality is described using the I/O transaction concept. The notation for a transaction is:

module_instance_name/protocol_name.signal_name

It states that the named module port has a word on the named signal terminal to be transmitted or received using the named protocol, for example, *Mem/writem.ADDR<7:0>*. The protocol name identifies the event graph from the module library that represents the I/O protocol. In the example, *writem* and *writep* refers to the library event graphs shown in Figure 4-3. In general, module ports may employ more than one type of protocol, and the specific protocol named in the IDL transaction will be chosen by the designer depending on the application requirements.

By using the transaction concept together with the underlying module library, the user does not need to specify the details of the protocol nor the control I/O signals of modules on which the signaling protocol occurs. In fact, in Figure 4-3, the control signals *Req*, *Ack* and *CS_L* in the block diagram do not appear in the input specification. The event graphs in the module library contain information about the protocols, time constraints and control signals that make up the low-level behavior of each module.

The I/O transaction only describes one end of the inter-module communication. An inter-module transfer is described with the assignment "=" notation, where the right side is the source transaction and the left side is the destination transaction. In the example, the first IF statement conditions any write transfer on the status of the processor write signal, *WR*. If the condition is met, there is a byte address and byte data transfer, where the processor uses its *writep* protocol and

the memory uses its *writem* protocol. In effect, the interlocking of protocol events that synchronize any transfer is implied in the assignment statement rather than explicitly stated. The communication routine implicitly returns to the beginning once it reaches the end, repeating ad infinitum.

In addition to the key features presented above, the specification also allows transfers to use data operations. Source bit vectors can be put through a logic or addition function, appearing on the right side of the assignment statement. Then the result is transferred to the destination port named on the left side. Besides the conditional IF feature, transfers can be embedded in concurrent, sequential or iterative control statements. These are typical of hardware description languages. In Figure 4-3, the "{" and "}" brackets surround the concurrent data and address transfers.

The input language described allows the designer to concisely express communication behavior, since the knowledge of protocol details comes from the underlying library and is transparent to the designer. The model presented works well for describing a wide range of interface functions. Further, the language and module library minimizes the redesign effort if a module experiences a change in its I/O specification. For instance, an upgraded memory with a shorter set-up time constraint and faster access time may replace the older version in the module library. This change is reflected in the module library, while the original high-level specification is still applicable. In keeping with a system design approach, the combined use of the input language and module library provide the proper level of abstraction and support modular design.

4.3 The IDL Specification

This section formally surveys the main features of the IDL specification and gives examples to illustrate how they are used. Currently, the language provides the minimum set of constructs that allow the designer to easily and briefly describe the interface behavior. Although it was developed

as a synthesis language, all its constructs can also be simulated. The IDL grammar and its parser is implemented with the Lex and Yacc facilities. Appendix B describes the syntax in detail.

4.3.1 Design Declarations

As shown in Figure 4-4, the entire IDL description is surrounded by the DESIGN declaration, where the designer can choose an arbitrary name to identify the interface. The language allows symbolic names for constant bit vectors or non-negative integers, and any of these are declared next. Examples where this feature is useful is a communication that requires incoming address to be compared to the destination address (specified as a constant), or a communication that requires the source data to be logically ANDed with a mask (specified as a constant) before it is sent to the destination.

```
DESIGN design_name
    CONSTANT constant_name = value;

    PORT library_module_name instance1, instance2, ... ;'
        SOURCE signal1, signal2, ...;
        DEST signal1, signal2, ...;
        BIDIRECT signal1, signal2, ...;
    ENDPORT;

    Block Declarations
ENDDESIGN;
```

Figure 4-4 : Design, Constant and Port Declarations

After the CONSTANT declarations in Figure 4-4, all the module ports participating in the communication are listed along with the information signals involved, as shown in the figure. In the PORT statement, the library name is first given followed by the instance names. The direction of the I/O signal is either source, destination or bidirectional. Sources (declared with SOURCE)

are inputs into the interface. Destinations (declared with DEST) are outputs of the interface. Bidirectional signals (declared with BIDIRECT) carry signals that flow in either direction.

Finally, within the DESIGN declaration are the block declarations which contain the specification body. Blocks are described below.

4.3.2 Block Declarations

An interface can be made up of independent and concurrently running blocks. This is similar to the process notion in distributed systems or the process notion in VHDL [Lipsett89]. The actual behavior of the interface is described within the BLOCK declaration, shown in Figure 4-5. The block can be thought of as an external infinite loop. Once the sequence of statements described inside the block has completed execution, behavior automatically returns to the top statement.

```
BLOCK block_name
    ROUTINE routine_name;
        statements
    ENDROUTINE;

    FUNCTION function_name<return-bit-width>(parameter1, parameter2, ...);
        BDS "file_name";
        or statements
    ENDFUNCTION;

    PROCEDURE procedure_name;
        statements
    ENDPROCEDURE;

    RESETPROC reset_name;
        statements;
    ENDRESETPROC;
ENDBLOCK;
```

Figure 4-5 : Block, Routine, Function and Procedure Declarations

Inside the BLOCK, behavior is partitioned into four possible subgroups. The first is the ROUTINE. This declaration contains the description of inter-module transfers, including data and control flow. To cleanly organize a complex behavioral description, IDL provides the FUNCTION and PROCEDURE declarations.

The function is a subprogram to the main routine, and it only describes combinational logic using either IDL combinational logic operators or the name of a BDS formatted file. It returns the value of a bit vector using the RETURN statement. BDS is a language for specifying combinational logic and is compatible with the MIS logic synthesis system [Segal88]. For example, the function is very useful for describing interfaces that decode addresses in a memory mapped system. The address map can be described in the FUNCTION declaration and is kept separate from the main routine. The main routine calls the decoding function at the appropriate time in its program, and the function is instantaneously evaluated. During synthesis, the actual evaluation time will be scheduled.

The PROCEDURE declaration is a subprogram to the main routine that describes a subsequence of inter-module transfers. It does not return a value to the main routine. Its body is described with any IDL data operation or control statement. It is best used when system communications involves repeating a subsequence at various points in the specified behavior.

The fourth subblock is the reset procedure declaration, RESETPROC. It is a special function that runs concurrently with the main routine. It monitors input signals, such as those indicating status, and whenever it detects a specified condition, it interrupts the control flow in the main routine and returns control to the first statement. In this way, the interface is reset.

The following subsections describe how behavior is specified in the ROUTINE declaration.

4.3.3 Transactions and Transfers

As described in the previous sections, the I/O transaction is a single information value passing through a port implying the signaling protocol. Specifying a transaction does not mean the interface instantaneously samples the information signal. Instead it is interpreted as the interface samples the signal when a control event indicates that information is valid on that signal. The designer does not need to know which control event, or the time constraint governing the sampling set-up or hold times. That information is captured in the module library and will be extracted automatically during synthesis. The I/O transaction is specified as:

module_instance/protocol.signal<msb:lsb>

and its semantics were explained in the previous section. The *module_instance* and *signal* names must be specified. Usually, the signal is information rather than control, since signals like data and address are the actual information being transferred between modules. If no *protocol* is named, then this literally means the instantaneous value of the signal, independent of the any control event. Also, if the most significant and least significant bit index, *msb* and *lsb*, are not specified, then the signal is one bit wide. When only a portion of the total signal width is specified in the routine, the outside bits are regarded as "don't care" at that moment in time.

The inter-module transfer is a mapping of a source transaction to a destination transaction. The most basic transfer an interface can carry out is directly sending a packet of information from one source to one destination port, and this behavior is expressed with the assignment statement, "=". Examples are the address and data transfers from processor to memory in Figure 4-3. It is important to emphasize again that the assignment statement represents a complete transfer cycle, where the data transfer and event-level synchronization occur jointly and concurrently. Protocol events such as clock or handshaking is implied and transparent to the writer of the IDL description.

In general, an inter-module transfer can involve multiple source ports and transformations. The general form of a transfer statement is:

Dest_transaction = Function_of(Src_transaction1, Src_transaction2, ...);

The next two sections describe IDL data and control operations that extend the transfer statement beyond the strict one-to-one mapping behavior.

4.3.4 Data Operation Expressions

Data operations manipulate values from input transactions and produce a new value. They consists of the very simple constant and bit selection expressions, delay operation, logical and arithmetic operations, and relational expressions.

Constant and Bit Selection Expressions

The constant expression is a constant bit vector or an integer. For example, these three statements assign the same binary value to the destination signal:

```
Mem/writem.DATA<7:0> = '00010001';
Mem/writem.DATA<7:0> = 17;
Mem/writem.DATA<7:0> = MyConstant;
```

Integer expressions in an assignment statement are translated to the equivalent binary representation. *MyConstant* is the symbolic name of the '00010001' bit vector declared before the PORT definitions.

Bit selection is simply choosing a bit slice of a vectored signal. So,

```
Proc/writep.DATA<7:4>;
Proc/writep.DATA<7>;
```

selects the upper half and the most significant bit of the DATA byte. Complex interface functions, such as byte swapping and concatenation, are expressed by composite bit selection operations. For example, in the following statements, the memory receives a byte formed by swapping the upper and lower half of the processor byte.

```
{ Mem/writem.DATA<7:4> = Proc/writep.D<3:0>;
  Mem/writem.DATA<3:0> = Proc/writep.D<7:4>; }
```

The bit selection feature can also demonstrate how communication involving multiple sources and destinations is described. In this example, the full memory byte is formed by concatenating half bytes from two processors, Proc1 and Proc2.

```
PORT PROCESSOR Proc1, Proc2;
ROUTINE main;
   { Mem/writem.DATA<7:4> = Proc1/writep.D<3:0>;
     Mem/writem.DATA<3:0> = Proc2/writep.D<3:0>; }
ENDROUTINE;
```

The description also shows that although two instances of the processor module are declared, both processor transactions name the writep protocol. This is because protocols are not instantiated, but used by reference.

Delay Operation

Values from past signal samples can be referenced in a statement by using the sample delay operation, "@N". N is a positive integer. So, the statements

```
{ Mem/writem.DATA<7:4> = Proc/writep.D<7:4>;
  Mem/writem.DATA<3:0> = Proc/writep.D<3:0>@1; }
```

indicate that the upper half of the memory byte comes from the current processor byte, while the lower half comes from the previous sample. The delay operation is the only data operation that has state. Its use implies memory hardware in the interface to be synthesized, since the interface will need to hold the past value in temporary storage.

Logical and Arithmetic Operations

Logical operations include the logical invert NOT, logical AND, OR, NAND, NOR and XOR expressions. All accept two inputs except for the complement which works with one input, and all can accept transaction values or constants. An example is the following statement that masks the source word as it is sent to the destination.

```
Mem/writem.DATA<7:0> = Proc/writep.D<7:0> AND '11110000';
```

Arbitrary and complex combinational logic manipulations can be described within the FUNCTION declaration, discussed in a previous section. In the routine body, the function is invoked with the function call expression. For example, this statement calls a function that contains the combinational logic description of a check sum function.

Mem/writem.DATA<7:0> = Check(Proc/writep.D<7:0>);

The current IDL language implementation only supports the addition "+" and increment "++ constant" arithmetic operations. These are sufficient, because inter-module communications practically only use these types. Usually, they are used in applications that require address generation such as in direct memory access.

Relational Expressions

The relational expressions, EQL and NEQ, test for equality or inequality between two variables or a variable and a constant. An example of this is:

IF (*Proc/writep.WR* EQL *'1'*) THEN ...

Here, a positive result from the equality test causes the flow of control to branch.

4.3.5 Control Flow Statements

The IDL language provides four ways of expressing control behavior. These are the conditional statement, concurrent construct, sequential statement and the loop statements. Complex control over the sequencing of transfer is achieved by appropriately combining these statements.

Conditional Specification

The first control flow operation is the conditional IF statement. It was introduced earlier in Figure 4-3. It has the general form:

IF *condition-expr* THEN
    statements

```
ELSE
    statements
ENDIF;
```

The *condition_expr* is a boolean control function. When it evaluates to true, it causes control to branch to the first set of statements. Otherwise, control branches to the set of statements following the ELSE clause. The ELSE clause is optional, and if it is omitted and the condition_expr evaluates false, control will exit the IF statement continuing with the following statement.

Concurrent Specification

Concurrent actions are specified by surrounding them with the "{" and "}" brackets. It too was previously introduced in Figure 4-3. Specifying concurrent operations shows the maximum concurrency that can be achieved. Some sequential behavior may emerge because of data dependencies between certain operations.

Behavior consists of many concurrency levels. The event graph represents fine-grained concurrency between I/O events. While the BLOCK declaration described above captures course-grained concurrency between independently operating hardware. The degree of concurrency discussed here falls between the event and block extremes. It describes concurrent transfers between related modules, rather than parallel communications among independent modules as in the BLOCK case.

Sequential Specification

Sequential behavior is the execution of an action following the completion of another. Specifying sequential behavior forces control precedence even if consecutive transfers have no data dependencies. It is expressed by separating statements with the statement:

NEXT(*(src_port1), (src_port2),* ...);

The NEXT keyword implies the precedence. However, on its own, it can cause ambiguities about how consecutive transfers are executed. To illustrate this, consider the sequential behavior

specified as:

Mem/writem.DATA<7:4> = Proc/writep.D<7:4>;
NEXT();
Mem/writem.DATA<3:0> = Proc/writep.D<3:0>;

In the first transfer, the memory receives data from the processor. On the next transfer, it is not clear whether the processor has actually produced new data for the second transfer to memory or if the processor port remains at the same state as before. To clarify this issue, the *src_port* arguments in the NEXT statement allow the designer to specify the source ports that have a new data sample. In the IDL implementation, if the name of the source port is absent in a NEXT statement, then it is assumed that the port remains at its old value, as in the example above. In this case, the interface to be synthesized may require a memory element to capture the old sample from the processor. To specify that the processor port has a new sample for the next transfer to memory, its name is listed in the NEXT statement as:

Mem/writem.DATA<7:4> = Proc/writep.D<7:4>;
NEXT(*(Proc)*);
Mem/writem.DATA<3:0> = Proc/writep.D<3:0>;

In the behavioral description above, multiple assignments are made to the same destination signal, DATA. The IDL model allows multiple assignments over a sequence of transfers. It does not allow multiple assignments within a concurrent set of statements, since that would result in conflicting values being sent to a destination. Also the end of a ROUTINE implies a NEXT statement where all declared source ports produce a new value as the behavior returns to the beginning of the routine.

Loop Specification

Loop behavior is captured by the ITERATE or WHILE statements. Interfaces that perform block transfers or multiplex parallel data onto a serial line can be conveniently described with these features. Examples are given in the following section.

The first statement specifies a fixed number of times a piece of behavior is repeated. It has the form:

ITERATE *index* FROM *constant1* To *constant2* DO
    statements
ENDITERATE;

The index variable is local to the iteration loop and does not need to be pre-declared. The *constant* is an integer constant or a symbolic constant name. The ITERATE statement is effectively a short-hand notation for a repeated set of statements separated by the NEXT statement.

When behavior is iterated over a number of times that is data dependent, the WHILE statement is used. Here the number of iterations is known only at run time, whereas it was known at compile time in the ITERATE case. The statement is specified as:

WHILE *expr* DO
    statement
ENDWHILE;

As long as the expression *expr* is true, the loop repeats.

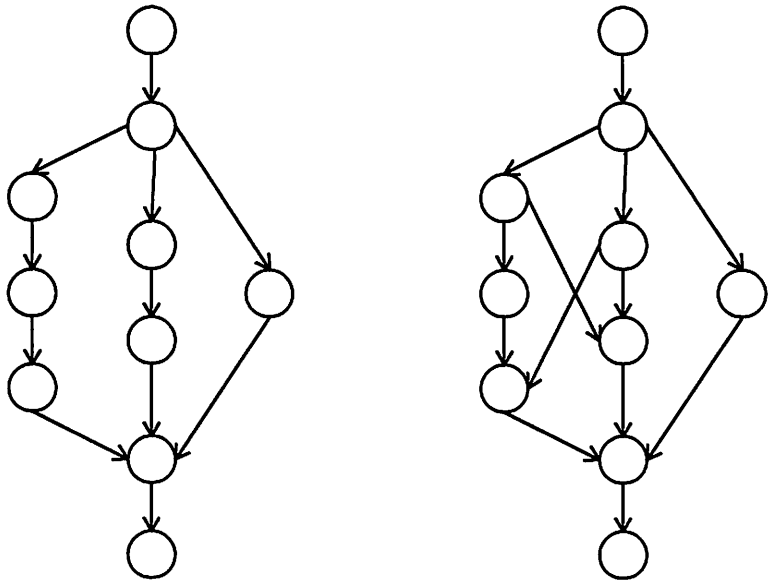
4.3.6 Limitations

The IDL specification method has two main limitations. The first is related to non-deterministic behavior and the second is concerned with concurrent/sequential control flow.

Currently, the IDL language only describes deterministic behavior. As the sequence of statements progresses, the next action is exactly known. Even in conditional statements, as the branch taken is determined at run-time, the set of branches is known, and the behavior in each of its branches is predictable. However, sometimes inter-module communications involves arbitration where several modules contend for a shared resource, such as shared memory or a bus, and only one wins in the arbitration process. Strictly speaking, this behavior is non-deterministic, since the outcome of arbitration is unpredictable. In practice, arbitration is usually implemented by an algorithm or

policy that is deterministic but makes the result look fair yet random [Guibaly89]. Examples are priority-based or round robin algorithms, and they can be logically expressed in the IDL language or with the BDS language.

The method of specifying concurrent actions allows subgroups of concurrent operations to be nested within a concurrent construct, "{ ... }". Also, the sequential NEXT statement can specify precedence between subgroups of concurrent operations. This method of specifying concurrent and sequential behavior models control flow as a series of parallel branches. This is shown in the left-hand control flow graph of Figure 4-6, where the circles represent a transfer or data operation and edges represent precedence. However, concurrent and sequential control can have a more general flow like that shown on the right side of the figure. In fact, an event graph models this general form of control precedence. So, the current IDL language can only model a subset of all possible control flows. In practice, interface behavior can be adequately modeled with the current control flow features, and the set is not so restrictive as the previous statement may indicate. For unrestricted control flow, the language needs to be enhanced with fork/join statements, which is supported by a few other hardware description languages such the AHPL language developed in the early days of HDLs [Hill78].



IDL Control Flow

General Control Flow

Figure 4-6 : Control Flow Examples

4.4 IDL Examples

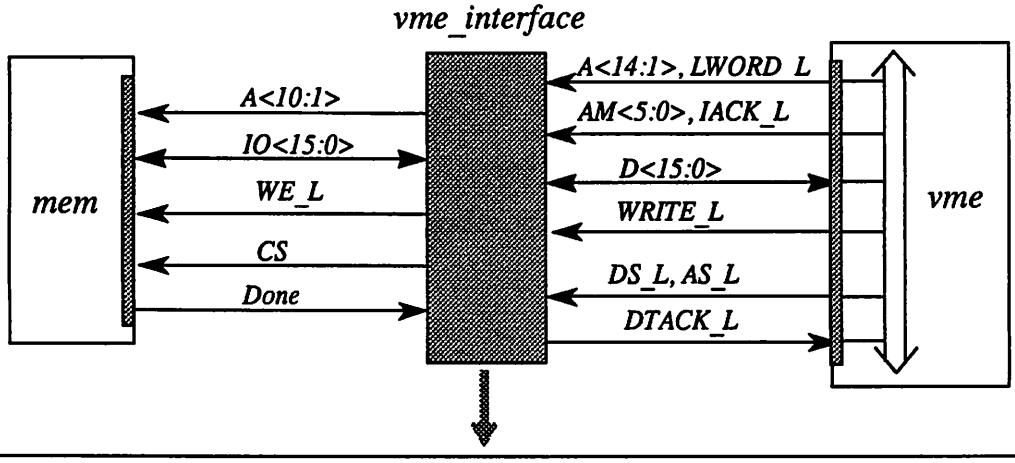
The previous sections have described the main IDL features. In this section, they are applied to four representative but more complex interface examples compared to the one from Figure 4-3. These examples demonstrate how the language is used to describe particular types of behavior. So, the given IDL descriptions show the specification body, leaving out the port declarations. The full descriptions are given in Appendix B.

VME System Bus Interface

Figure 4-7 (on two pages) illustrates the inter-module communication between the VME system bus and a static RAM module with built in acknowledgment. The interface provides read and write access from bus to the memory. The VME protocol and timing are described with an event graph for the read cycle and another for the write cycle (on second page of Figure 4-7). Similarly, the RAM has a separate event graph for its read and write cycles. In the IDL specification, the *dtb_read*, *dtb_write*, *readm*, and *writem* protocol names refer to these event graphs, respectively.

The IDL description also highlights how the FUNCTION declaration and the conditional IF statement are used to specify the interface's decoding function. The outer IF statement conditions any read or write access on successful decoding, invoked by the function call, *decode*. Decoding for VME memory mapped modules is quite complex, so the decoding function is described in a separate BDS file to achieve a cleaner IDL specification. The file is named in the FUNCTION declaration with the BDS statement.

The inner IF statement selects the read access transfers or the write access transfers depending on the status of the vme *WRITE_L* signal, which is considered an information type signal. The write access consists of three concurrent transfers, and the same holds for the read access. In the first transfer, the interface generates a constant bit value that is sent to the memory *WE_L* signal. The other two transfers occur directly between the VMEbus and the memory.




```
BLOCK vme_interface

    ROUTINE main;
        ! comment: if memory module selected
        IF ( decode(vme/all.A<14:1>,LWORD_L,AM<5:0>,IACK_L) EQL '1' ) THEN
            IF ( vme/all.WRITE_L EQL '0' ) THEN       ! write access
                {mem/writem.WE_L = '0';
                mem/writem.A<10:1> = vme/dtb_write.A<10:1>;
                mem/writem.IO<15:0> = vme/dtb_write.D<15:0>; }
            ELSE                                      ! read access
                {mem/readm.WE_L = '1';
                mem/readm.A<10:1> = vme/dtb_read.A<10:1>;
                vme/dtb_read.D<7:0> = mem/readm.IO<7:0>; }
            ENDIF;
        ENDIF;
    ENDROUTINE;

    FUNCTION decode<0>(w<3:0>, x, y<5:0>, z);
        BDS "decoder.bds";
    ENDFUNCTION;

ENDBLOCK;
```

Figure 4-7 : IDL Description for VMEbus Interface

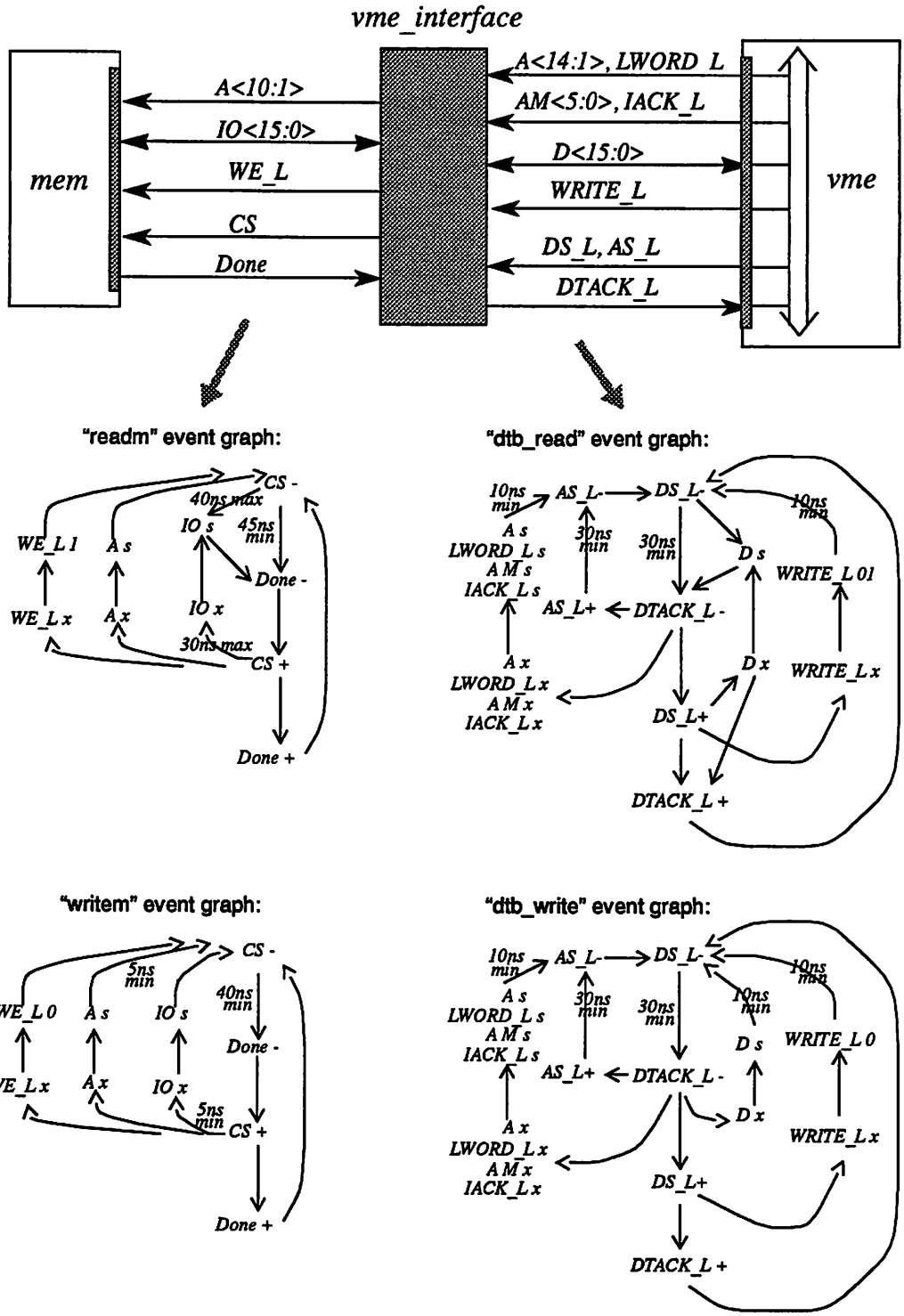


Figure 4-7: Event Graphs for VMEbus Interface

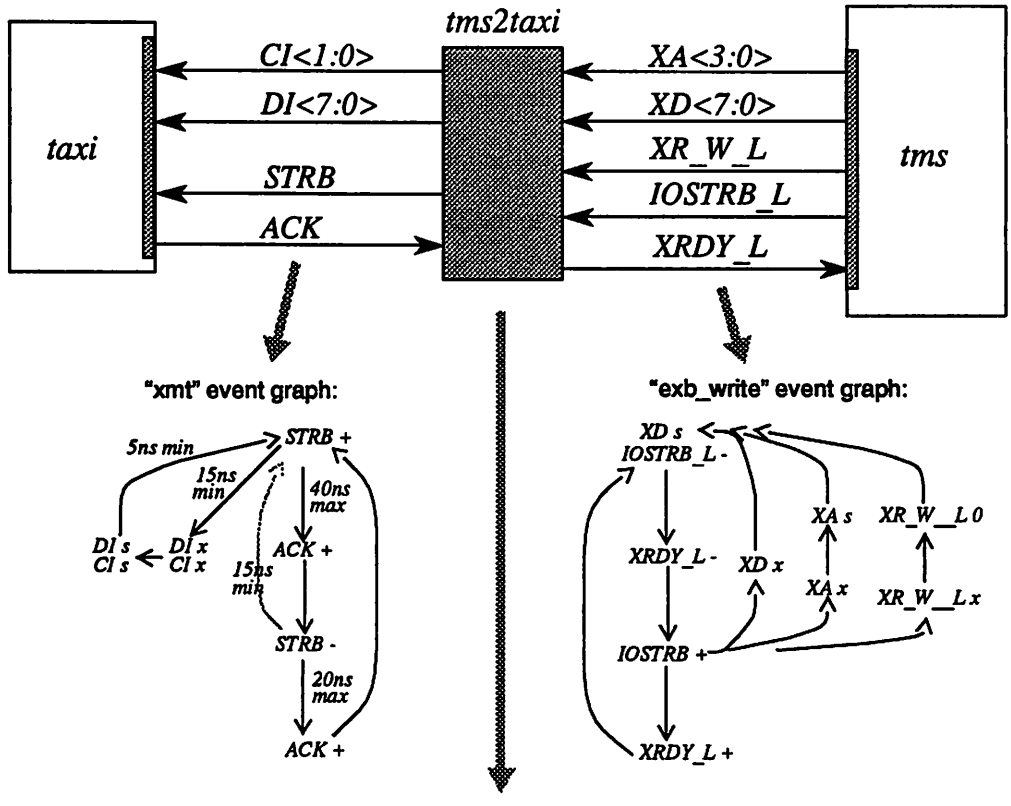
In the block diagram, the VME control signals (*AS_L, DS_L* and *DTACK_L*) and the memory control signals (*CS_L* and *DONE_L*) are used for protocol signaling. They do not appear in the high-level input specification, since they are hidden in the module library as shown by the event graphs on the second page of Figure 4-7.

TMS320 to Optical Link Interface

The interface from Figure 1-3 in Chapter 1 provides a good example of how multiplexed communication is described with the IDL language. The specification is shown in Figure 4-8, and it references the shown event graphs from the module library.

The TMS320 based uni-processor presents address and data and parallel busses. The TAXI port must accept these words on its *DI* signal bus over two separate and consecutive transactions. A TAXI word on the *DI* bus must also be accompanied by a header on the *CI* signal bus,

In the IDL description, the first set of concurrent transfers consist of the TMS address and header transfer. The header '00' is generated by the interface. The TMS data is sent on the second set of transfers, as specified by the NEXT statement. This statement is the key in describing multiplexed behavior. Notice that the TMS source port is omitted from the NEXT argument list, specifying that source data from the previous transfer is used for the second transfer.



IDL specification:

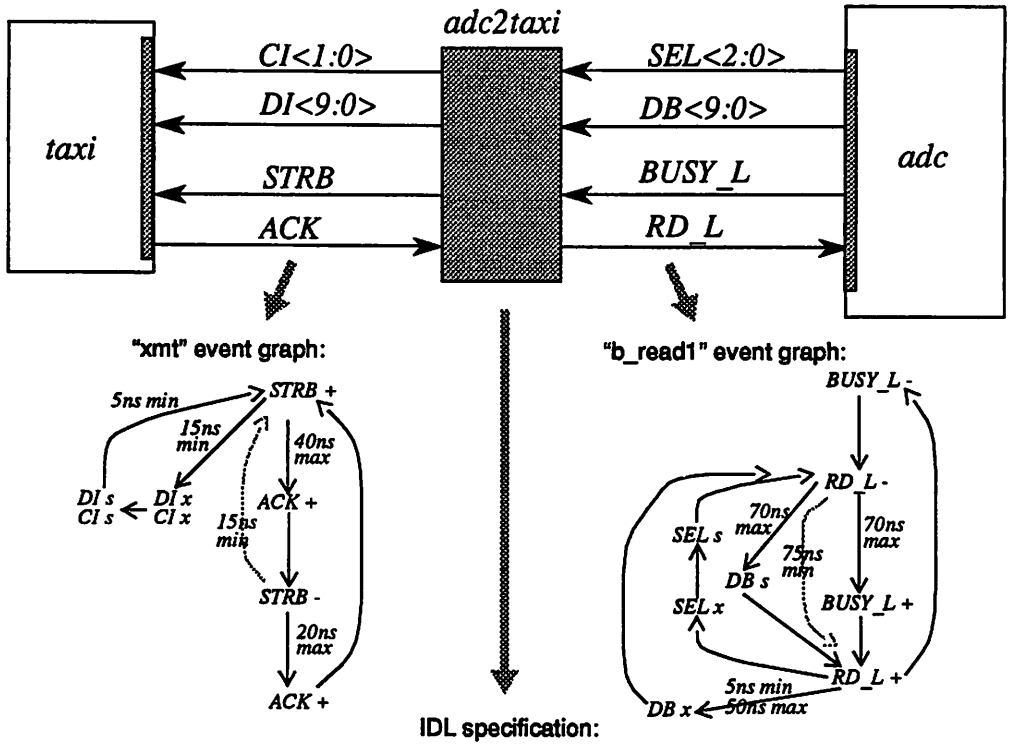
```
BLOCK tms2taxi

    ROUTINE main;
        IF ( tms/all.XR_W_L EQL '0') THEN
            { taxi/xmt.DI<3:0> = tms/exb_write.XA<3:0>;
            taxi/xmt.CI<1:0> = '00'; }
            NEXT();
            { taxi/xmt.DI<7:0> = tms/exb_write.XD<7:0>;
            taxi/xmt.CI<1:0> = '01'; }
        ENDIF;
    ENDROUTINE;

ENDBLOCK;
```

Figure 4-8 : IDL Description for TMS320 to Optical Link Interface

A/D Module to Optical Link Interface

Figure 4-9 demonstrates how block transfers are specified and how the PROCEDURE declaration is used. The source is an A/D module consisting of six individual converters sharing an I/O port, and the destination is the TAXI port. The interface first transmits a header, and then sequentially transfers six consecutive A/D words to the TAXI port.

In the IDL description, the header and A/D transfers are separated by the first NEXT statement. The statement "*xmit();*" is a procedure call to the procedure named *xmit* which contains the header transfer. Although it is a trivial application of the procedure facility, this example does show how use of the procedure organizes the behavioral description, keeping it clean. Following the first NEXT statement is an ITERATE statement specifying six iterations of two concurrent actions. First, the interface generates an address value, *j*, and transfers it to the A/D module address lines, *SEL<2:0>*, using the *b_read1* protocol (shown in the event graph of Figure 4-9). The integer value j represents the equivalent 3-bit binary vector. The second action transmits the data provided by the selected A/D converter to the TAXI port, which uses the *xmt* protocol shown in the figure. Since the two transfers are concurrent, the *b_read1* I/O protocol is cycled only once even though it appears in two statements. The NEXT statement at the end of the ITERATE body specifies that the A/D module advances to the next transaction for the following transfer to the TAXI port.




```
BLOCK adc2taxi

    ROUTINE main;
        xmit();                                  ! procedure call
        NEXT();
        ITERATE j FROM 0 TO 5 DO
            { adc/b_read1.SEL<2:0> = j;          ! generate address for A/D module
            taxi/xmt.DI<9:0> = adc/b_read1.DB<9:0>; }  ! send A/D word to taxi
            NEXT( (adc) );
        ENDITERATE;
    ENDROUTINE;

    PROCEDURE xmit;
        { taxi/xmt.CI<1:0> = '10'; }             ! transmit header
    ENDPROCEDURE;

ENDBLOCK;
```

Figure 4-9 : IDL Description for A/D Module to Optical Link Interface

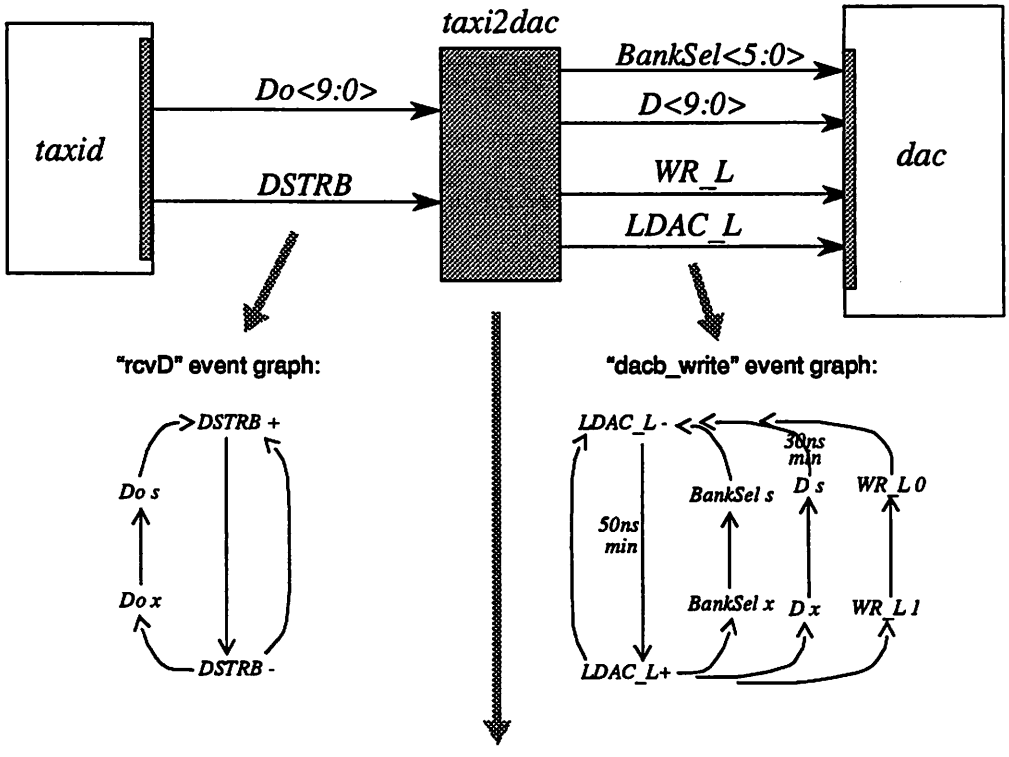
Optical Link to D/A Module Interface

Demultiplexed transfers between a TAXI optical receiver and a D/A module are described in Figure 4-10. In this example, words from two consecutive TAXI transactions are collected and sent to the D/A module; the first word serves as the address to the module and the second is the data to be converted. It illustrates a transfer that involves the @ delay and function call expression.

The specification body begins with the NEXT statement which has the argument "*taxid*". This argument is the instance name of the TAXI module. Accordingly, by the end of sequential statement, one transaction from the TAXI port has occurred. The following concurrent construct contains two transfers. In the first transfer, the input is a 10-bit word from the previous TAXI transaction, as shown with the"*taxid/rcvD.Do<9:0>@1*" expression. The decode function logically transforms the word into an address which is assigned to the D/A *BankSel* destination signal. In the second but concurrent transfer, the input is a 10-bit word from the current TAXI transaction, and it is assigned to the D/A data signal. In summary, the NEXT statement essentially forces the interface being described to transfer two consecutive words from the TAXI port when the second word has arrived.

Summary

The presented four examples demonstrate how the input language can describe system communications at a high level of abstraction. Compared to the event graphs, the IDL specification method is brief and relatively easy to write. It captures the data and control flow particular to the interface application, which is the essence of the communication. The details of the I/O protocols are captured by event graphs in the module library. This saves the designer from re-entering the protocol specification as modules are reused in different applications.




```
BLOCK taxi2dac

    ROUTINE main;
        NEXT( (taxid) );
            { dac/dacb_write.BankSel<5:0> = decode(taxid/rcvD.Do<9:0>@1);
              dac/dacb_write.D<9:0> = taxid/rcvD.Do<9:0>; }
    ENDROUTINE;

    FUNCTION decode<5:0>(x<9:0>);
        BDS "decoder.bds";
    ENDFUNCTION;

ENDBLOCK;
```

Figure 4-10 : IDL Description for Optical Link to D/A Interface

4.5 Simulation

To achieve a successful implementation, a specification method is much more meaningful with the support of a simulator. The primary goal is to confirm that the described inter-module communication behavior is actually what the designer had intended and to evaluate the expected communication performance. At the system level, simulation avoids the details of the internal interface structure and implementation.

Another purpose of system level simulation is to gather statistics about the message traffic through the interface. This information can be applied to determine the optimum depth of buffers (also called queue, FIFO or pipeline memory) inserted into the communication path (the interface) to meet throughput requirements. Buffers are most useful for supporting burst transfers between a fast and a slow module. In effect, each module communicates with the buffer at its own speed and does not see the other module. The buffer decouples the response time of the slow module from the fast one. An optimum depth, or range of depths, is such that the buffer is full during communications (neither empty nor overflowing). This parameter is difficult to determine because detailed statistics about the system traffic are usually undefined early in the design. So, the determined buffer depth is only as good as the model for port delays and transaction burstiness. Besides simulation, there are also other analytical methods for sizing buffers [Amon91b]. But, the quality of their results also relies on the quality of the traffic model.

Generating a Simulation Model

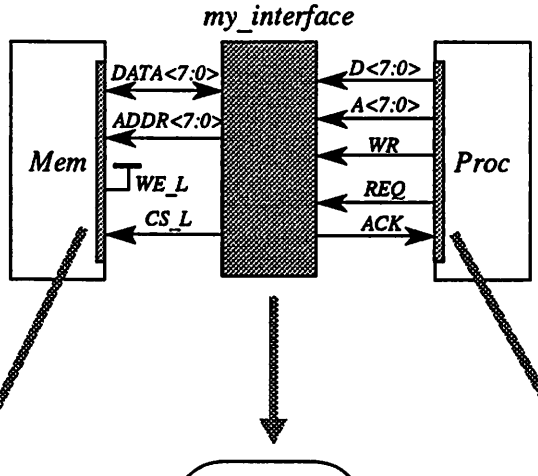
The IDL language was developed for synthesis and does not have an accompanying simulator. Instead, the VHDL simulation language is used to model system level communication. The mapping from IDL to VHDL is complex and results in a much more lengthy simulation model than the original synthesis specification, as outlined below and illustrated in Figure 4-11.

Each *module port* in the IDL specification (not the same as VHDL "ports") corresponds to a VHDL process which is thought of as a I/O process. The I/O process provides communications for the module port with its environment through VHDL ports made up of information signals and a two-phase handshake pair, using a ready and a done signal, that emulates synchronization. The handshake is strictly used as a synchronization mechanism in simulation and is unrelated to the actual module protocol. The details of the actual protocol are not modeled to keep simulation at a high abstraction level. The *interface* maps to one or more VHDL processes, depending on the number of concurrent constructs in the IDL description. In 4-11, the concurrent operation corresponds to one VHDL process which is by definition a concurrent operation. The interface process has a port (in the VHDL sense) which corresponds to each I/O process. The port consists of information signals and a two-phase handshake pair of ready and done signals.

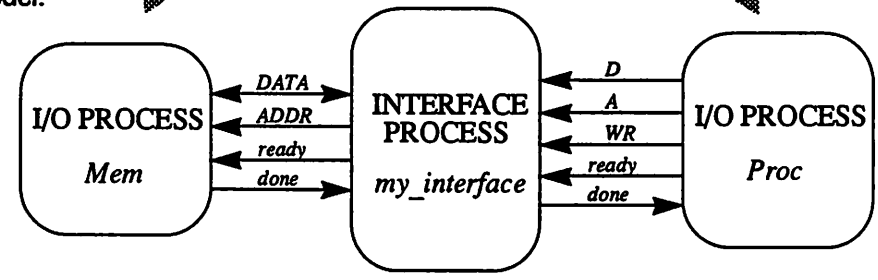
In the I/O process model, the delay between the handshake ready and done event reflect the cycle time specified by the module I/O protocol. In the interface process model, delays between its ready and done events can be the estimated delay of the interface circuit to be synthesized. To keep simulation at a high abstraction level, individual time constraints such as set-up and hold times are not modeled but reserved for simulation after synthesis.

The data flow within the IDL concurrent operation directly maps to the VHDL syntactic equivalents within the process. For example, the IDL transfer (an assignment statement) corresponds to the VHDL signal assignment "<=". The IDL function and procedure is the same as the VHDL function and procedure, respectively. IDL logic operations, such as "AND", map to their respective VHDL logic operation, such as "and". The IDL sample delay "@N", where the integer N is the delay amount, is the only data flow construct which does not have a direct VHDL mapping. Although not the only way, this operation can be modeled in VHDL with a static array variable that is local to the process (actually, VHDL does not even allow global variables). The array acts as a FIFO buffer, an its size is the same as the delay amount.

Block Diagram:



VHDL Model:



IDL specification:

```
DESIGN my_interface
  PORT PROCESSOR Proc;
    SOURCE Proc.D<7:0>, Proc.A<7:0>, Proc.WR;
  ENDPORT;
  PORT SRAM Mem;
    DEST Mem.ADDR<7:0>;
    BIDIRECT Mem.DATA<7:0>;
  ENDPORT;
  ROUTINE main;
    IF (Proc.WR EQL '1') THEN
      {Mem/writem.ADDR<7:0> = Proc/writep.A<7:0>;
      Mem/writem.DATA<7:0> = Proc/writep.D<7:0>;}
    ENDIF;
  ENDROUTINE;
ENDDESIGN my_interface;
```

Figure 4-11 : VHDL model of Module Ports and the Interface

Returning to the IDL concurrent operation, if it were nested in a conditional (IF) or loop (ITERATE and WHILE) statement, then these control statements also carry into the VHDL process and surrounding the VHDL data flow statements. The IF construct maps to the VHDL "if" statement, the ITERATE construct becomes the VHDL "for ... loop" statement and the WHILE construct corresponds to the VHDL "while ... loop". In a different scenario, if there was a second IDL concurrent operation following the one in the example (the two are separated with a NEXT statement), then another VHDL process is created for the second concurrent operation. The first process produces a token upon completion of execution, and this token is passed to the second process to activated it. With the token passing scheme, simulation emulates sequential behavior in the IDL specification. It is important to emphasize here that the NEXT behavior is different from the concept of sequential execution within a VHDL process.

This section has described what is involved in generating the VHDL model from the IDL specification. The main complexities are modeling the concurrent and sequential behavior and the sample delay operation. The key mapping techniques are using VHDL processes to model module ports and the interface, using a two-phase handshake to model synchronization, and token passing to model sequential behavior. At present, the VHDL model is manually generated. To support automatic generation, each module port should have a corresponding VHDL model in the module library. The models have not yet been installed into the library. The generation process starts by parsing the IDL specification, then translating it into the flow graph design representation, which is described in the next chapter. Finally, the VHDL model is created from the flow graph. The library models and automatic generation tool remain as future enhancements to the ALOHA system.

CHAPTER 5

DESIGN REPRESENTATION

In the larger task of integrating system hardware, behavioral synthesis automatically generates the structural implementation of an interface module from an abstract behavioral specification while meeting I/O protocol constraints. The behavior defines the inter-module communication function, and structure specifies a network of register-transfer-level logic components. From here, low-level design takes over to produce the actual physical implementation. Because it starts from an abstract level, behavioral synthesis is also called high-level synthesis.

The first step of behavioral synthesis is to translate the input specification into a design representation useful for synthesis, as shown in Figure 5-1. The behavior that needs to be represented consists of system-level data and control flow, and also I/O event sequencing and timing. A key feature of the design methodology presented in this work is separating the design representations for the two behavior levels. The design representation for system-level behavior is based on the control/data flow graph. The event graph represents the lower level of behavior, and is generated according to the data flow described in the flow graph and the I/O protocols captured in the module library. By describing behavior in two separate representations, the synthesis

process is simplified. In contrast, other representations have been developed that singly capture both levels of behavior in one and the same representation. They permit iterative application of some of the synthesis procedures, but make the control of the synthesis process more difficult [Nestor86][Whitcomb92][Borriello88a].

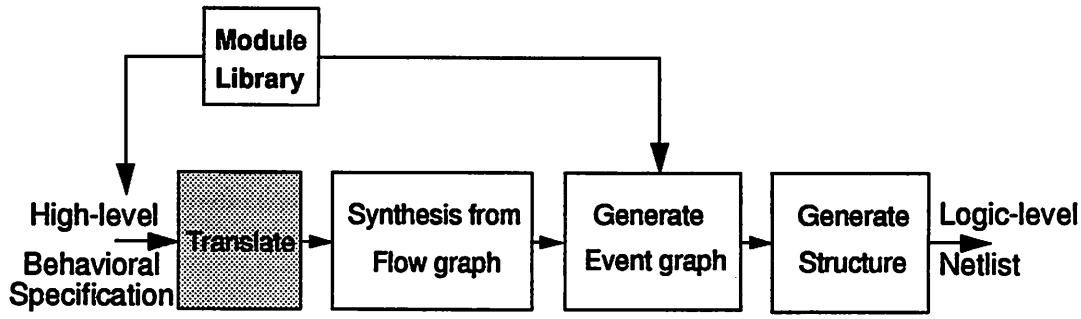


Figure 5-1 : Translation Step in Behavioral Synthesis

This chapter describes the flow graph design representation that drives behavioral synthesis in ALOHA. This includes the basic representation elements, and describes how a high-level IDL specification is translated into a flow graph. Examples of flow graphs are also provided. Synthesis from flow graphs and event graph generation are discussed in subsequent chapters.

5.1 The Flow Graph Representation

The flow graph is currently the most widely used representation for behavioral synthesis [McFarland90], although parse trees are also used [DeMicheli88]. The graph contains both the data flow and control flow implied by the input specification. Specifically, this means the data operations, data dependencies, control operations and control dependencies. Examples of flow graphs from other synthesis systems include the CMUDA Value Trace [McFarland78], the ADAM DDS [Knapp84] and the HYPER CDFG [Chu89] among others [Treleaven82] [Davis82] [Orailaglu86]. The common features between all these different flow graphs are nodes that

represented data and control operations, and directed edges that represented data and control dependencies. The ALOHA flow graph has these same features in addition to a special extension for abstractly representing I/O protocol synchronization, which was not addressed by the other flow graph representations.

This section uses simple examples to illustrate how the design representation captures data flow behavior and then how it captures control flow behavior. It should be kept in mind that flow graphs generated from typical interface specifications are much more complex than the examples shown. The next section gives more details about deriving the flow graphs from the IDL specification.

5.1.1 Data Flow

Figure 5-2 shows the equivalent graph for data flow in the processor to memory write example of Figure 4-3 in Chapter 4. The graph has primitive nodes to represent assignment and data operations such as constant, delay, logical and arithmetic operations. So, both the address and data transfers in the example translate to separate assignment nodes in the flow graph. Data dependency edges represent the inputs and output of a data flow node. A data flow node is fired when inputs are available on its input edges, and it produces a result on the output edge within the computation time of that node. To maintain abstraction, the computation time is assumed to be one unit of time. In the figure, the first assignment node is driven by an input edge corresponding to the processor address, and the assignment node in turn drives the output edge corresponding to the memory address. Input and output edges of data flow nodes do not carry any information about I/O protocols implied in the input specification.

For system level transfers, source and destination ports are synchronized by interlocking the I/O protocol events. This behavior is represented by a special type of node, called the *sync* node. As shown in Figure 5-2, a sync node is *associated* to a data flow node. It is essentially an abstraction that represents an I/O synchronization operation that is jointly and concurrently executed with a data operation. Attached to the sync node, the protocol dependency edge identifies the I/O protocol

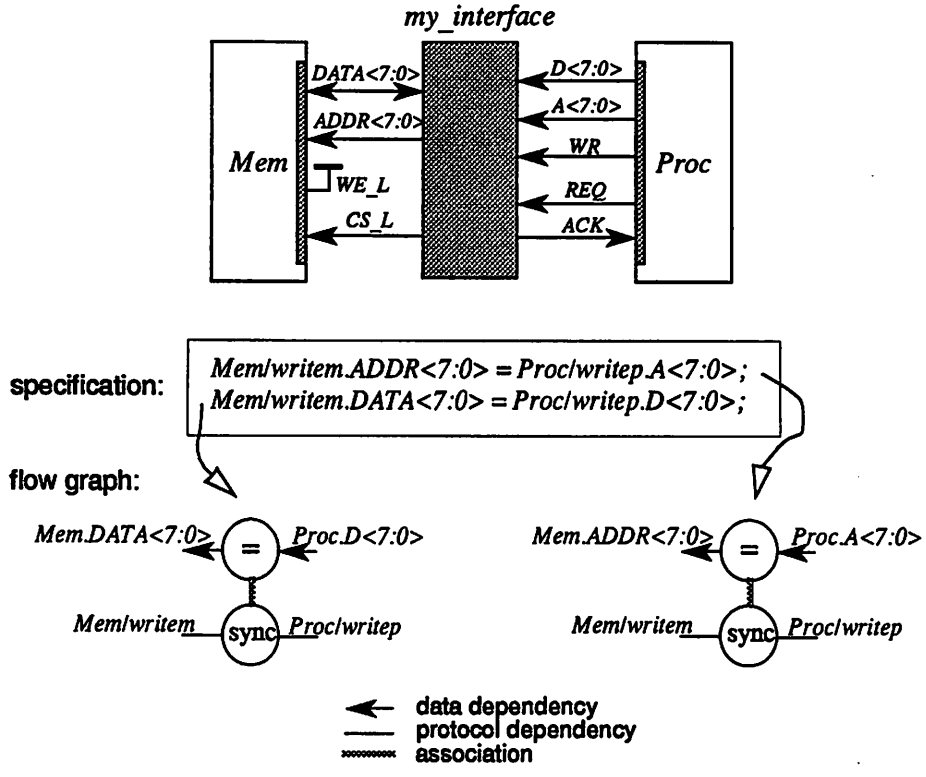


Figure 5-2 : Data Flow Graph for a Processor to Memory Write Interface

(captured in the module library) used by the operation's source or destination ports. It provides the link between behavior in the flow graph and behavior represented in the event graph. In general, there may be multiple sources and one destination port per transfer, and the sync node will have an edge for each unique source protocol and an edge for the destination protocol. Comparing the transfer statement to the flow graph equivalent, the behavior modeled by an I/O transaction is split across the data flow node and the sync node.

Figure 5-3 provides another example of a data flow graph that has complex operation. The AND operation provides output directly to the Memory, and, for this reason, an assignment node at the output is considered redundant and is omitted. A constant is viewed as a data flow node that continuously generates a constant value. Both the constant and delay nodes have a parameter that

specifies their constant and delay values, respectively. A sync node is associated with the AND node and with the delay node. A data operation with inputs and outputs that are internal to the flow graph does not have an associated sync node.

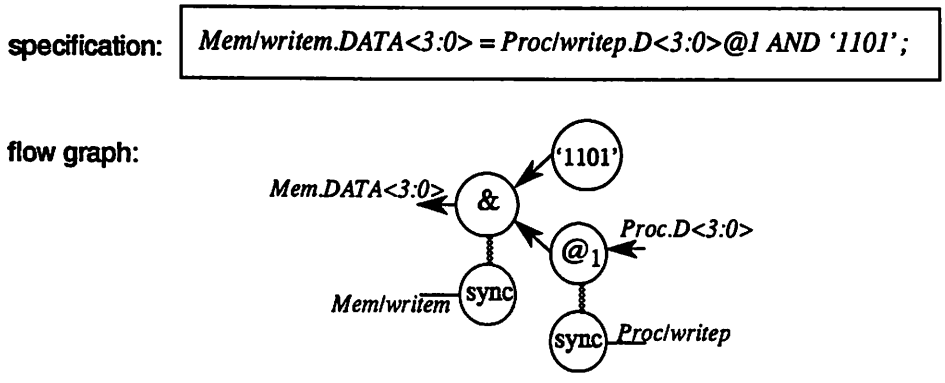


Figure 5-3 : Data Flow Graph with a Complex Operation

Although a flow graph represents behavior, its data flow nodes and dependency edges do imply some hardware but say nothing about the particular implementation. In Figure 5-2, the result of synthesis will include wires that actually transmit the address and data from the processor to the memory and logic that converts between the two I/O protocols. However, the final solution may or may not include latches to buffer the transfer, and usually there are multiple ways to interlock the I/O protocol events. The flow graph does not inherently specify such design parameters. These are imposed by the synthesis process.

5.1.2 Control Flow

The flow graph representation uses control nodes to capture concurrent, conditional and loop behavior. Control dependency edges represent sequential behavior. Control nodes are hierarchical, containing a subgraph for the data and control flow within their scope.

Figure 5-4 illustrates conditional control in the flow graph for the processor to memory write

example of Figure 4-3. The IF node is hierarchical, while the other nodes are primitives. The address and data nodes are contained within the IF control node, and will be executed concurrently only when the input condition (or guard) to the IF node is true. The condition "Proc.WR EQL 1" corresponds to the condition-expression of the IF statement. It is actually represented with the parse tree data structure within the *guard* node. For the sake of classification, the guard node is considered a data flow node that generates the condition status on a data edge to the IF node.

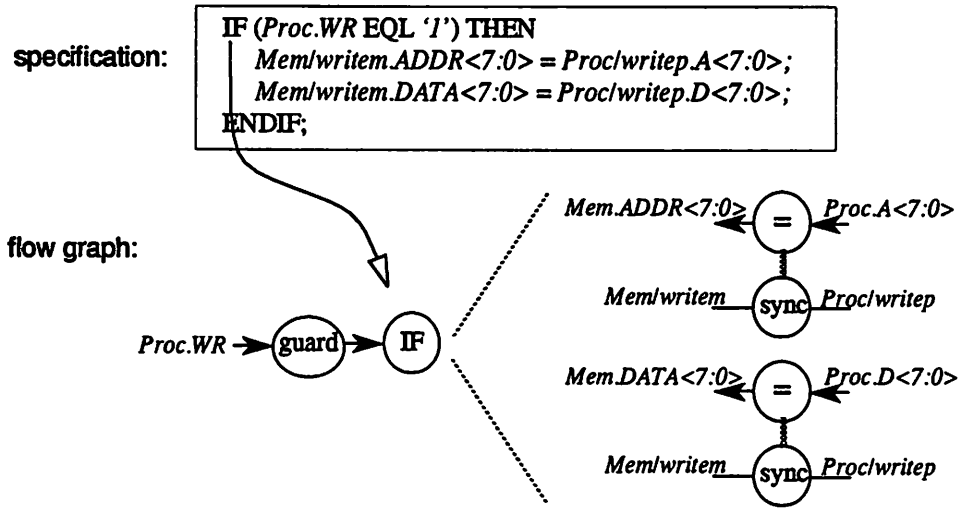


Figure 5-4 : Control/Data Flow Graph for Processor to Memory Interface

ITERATE and WHILE loop behavior also corresponds to hierarchical nodes in the flow graph. For example, if the data transfers of the previous example were embedded in a loop rather than a conditional statement, the equivalent flow graph will look like the one shown in Figure 5-5. An ITERATE control node has three parameters corresponding to the index variable, the minimum index value and the and maximum index value. In the example the parameter values are "i", 0 and 3, respectively. The data flow nodes contained in these nodes are allowed to execute concurrently unless data dependencies or an explicit control dependency is implied.

Concurrent behavior has the simplest control node representation. It too corresponds to a

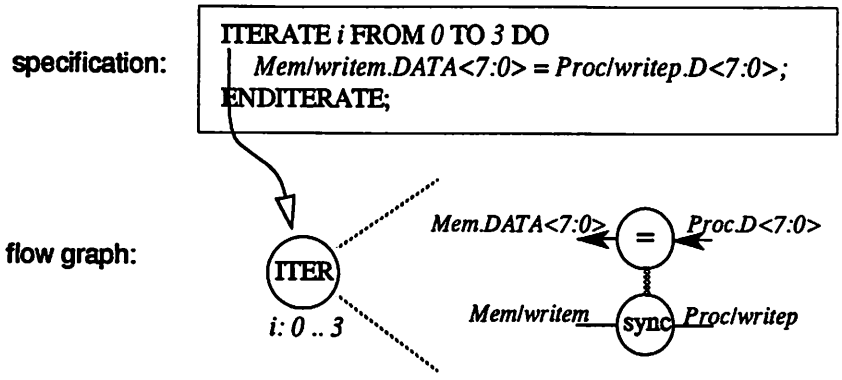


Figure 5-5 : Flow Graph with Iterate Control Node

hierarchical node in the flow graph, but it has neither conditional inputs nor parameters. It implies that the data operations it contains execute concurrently except when restricted by data dependencies. The concurrent version of the processor to memory write example is illustrated in Figure 5-6.

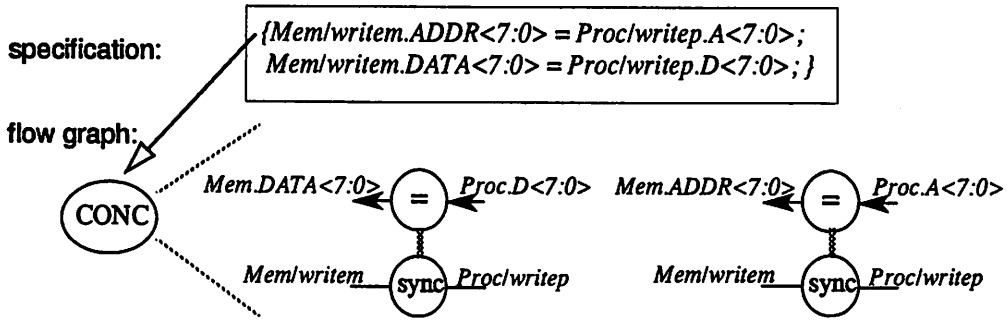


Figure 5-6 : Flow Graph with Concurrent Control Node

Finally, a control precedence edge represents sequential behavior, corresponding to a NEXT statement in the input specification. An edge directed from the first node to a second node specifies that the first node must be executed to completion before the second is fired. It is analogous to the precedence edge in event graphs. In Figure 5-7, a control edge places the two data transfers in a linear flow.

Created as a convenience for synthesis, the start node and its output control edge in Figure 5-7 mark the first data or control action implied in the input specification. All the flow graphs for the previous examples have a start node, but they were omitted to simplify the figures.

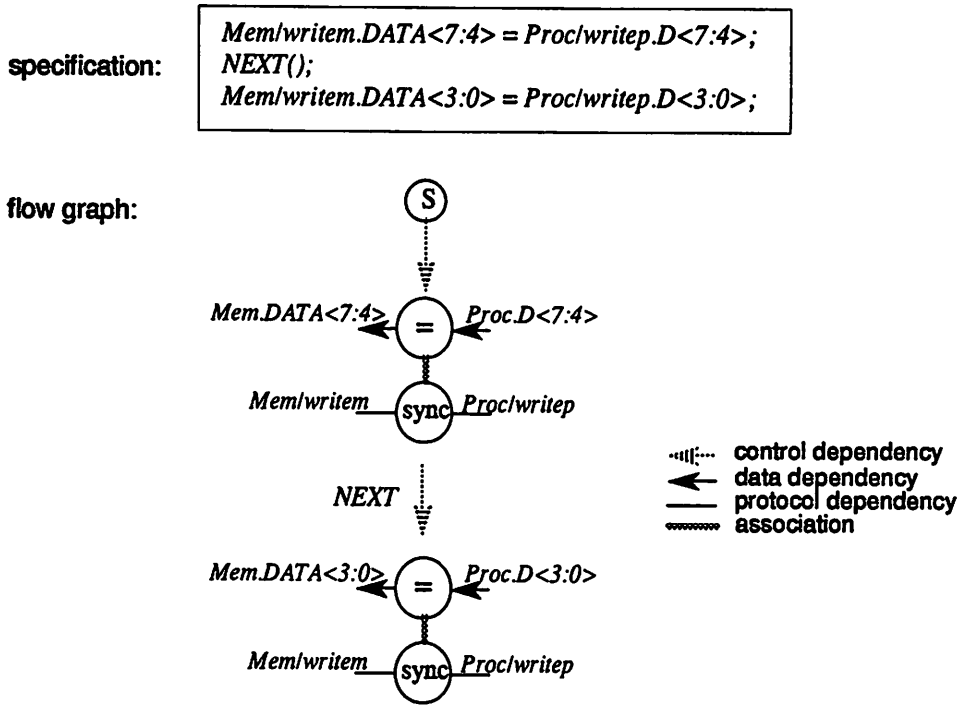


Figure 5-7 : Flow Graph with Control Precedence Edge

5.2 Translating from Specification to Flow Graph

The translation step brings the high-level input specification into the equivalent flow graph representation. It provides the bridge between the designer's input and the synthesis tools. Translation is a multi-step process itself, starting with a direct mapping between IDL data and control flow constructs to flow graph node and edge elements. This is followed by several passes that transform the initial flow graph into a form suitable for subsequent synthesis tools to take over the design.

5.2.1 Basic Construction

All the data flow constructs in the IDL language have an equivalent flow graph primitive node. Table 5-1 lists the different data flow nodes in the behavioral node library. It also shows the IDL counterpart of a node, the number of input and output edges and the names of parameters associated with the node. All the nodes have one or two inputs and exactly one output, except for the sync, function and guard node. The sync node is listed among the data flow nodes, although it represents an I/O synchronization operation rather than actual data manipulation. It has an unlimited number of edges.

Table 5-1 : Data Flow Nodes

Description	Node Name	IDL Construct	Inputs/Outputs	Parameters
assignment	"="	"="	In / Out	none
constant	constant	'binary_value'	None / Out	value
delay	"@"	"@"	In / Out	delay
NOT	"!"	NOT	In / Out	none
AND	"&"	AND	In1 In2 / Out	none
OR	"|"	OR	In1 In2 / Out	none
NAND	"!&	NAND	In1 In2 / Out	none
NOR	"!|"	NOR	In1 In2 / Out	none
XOR	"^"	XOR	In1 In2 / Out	none
add	"+"	"+"	In1 In2 / Out	none
increment	"++"	"++"	In / Out	none
equal relation	"=="	EQL	In1 In2 / Out	none
not eql relation	"!="	NEQ	In1 In2 / Out	none
function	comb	FUNCTION	* / Out	BDS file name
guard	guard	none	* / Out	none
synchronize	sync	none	*	none

The function node contains the parse tree representing the combinational logic described in the IDL function declaration, and has an input edge corresponding to each argument in the function call. Sometimes, the logic function is described in a BDS file name, and the function node captures that as a parameter value rather than as a parse tree. The guard node also contains the parse tree equivalent of an IDL description. The node generates a token when its input values meet a particular condition. The token is placed on a data edge that fires a conditional node.

With the exception of the NEXT and RESTART statement, the IDL control flow constructs map onto a hierarchical node. Sequential behavior specified by the NEXT statement corresponds to a control edge, and the RESTART construct corresponds to a primitive node. Table 5-2 shows the types of control flow nodes in the behavioral node library, their IDL counterpart and any associated parameters. A loop or concurrent statement, as well as the procedure or reset-procedure

Table 5-2 : Control Flow Nodes

Description	Node Name	IDL Construct	Parameters
if condition	IF	IF	none
else condition	ELSE	ELSE	none
concurrency	CONC	"{ }"	none
iteration	ITER	ITERATE	index, min, max
while	WHILE	WHILE	none
procedure	procedure name	PROCEDURE	none
restart	restart	RESTART	none
start	S	none	none

declaration, map to one hierarchical node. A start node is inserted into the flow graph. It has an output control edge that points to the first nodes in the control flow to be executed, as specified in the input specification. When a reset-procedure is specified, the start node is driven by a guard node containing the reset-procedure function.

The IF statement actually translates to two possible hierarchical nodes; one for the body of the THEN clause, and another for the body of the ELSE clause. Both are driven by the same input condition produced by a guard node that corresponds to the condition-expression in the IF statement. The ELSE node fires when the input condition evaluates to false. Like the example specification in Figure 4-7, an IDL description can have IF statements nested within another IF or ELSE clause. This configuration translates to an IF node driven by a guard node that contains the composite condition-expressions of the individual IF statements.

So far, it may seem that the data and control flow in the flow graph have been treated as disjoint entities. In actuality, the design representation views data flow as primitive behavior that takes place in time according to a specified control behavior. So, all data flow nodes are contained within hierarchical control nodes. For example, an IF node allows the data flow nodes to fire when a true token arrives on its input condition edge, and an iterate node sequences the data flow within its scope through the specified number of times. The procedure node allows its data and control flow to fire when it is called in the main flow graph. Even a single data transfer is contained within a concurrent control node.

5.2.2 Flow Graph Passes

Once the basic structure of a flow graph is constructed from the input specification, it is brought through transformations that alter or refine its structure in a manner that is independent of the input specification but useful toward synthesis. These transformations are described in these next two subsections.

The first pass eliminates redundancies in the derived flow graph. Since ELSE nodes execute when their input condition have a false token, they are transformed int IF nodes that are driven by the complement of the original input condition. This simple transformation helps to simplify the flow graph representation while maintaining the original information.

The second pass expands hierarchical procedure nodes. When a procedure is called in the control flow of the main flow graph, the subgraph contained in the procedure node is promoted to the top level and replaces the procedure node. After this pass, no procedure nodes remain in the flow graph.

The third pass unrolls the iterate loops. A hierarchical iterate node in the flow graph is expanded as in the procedure case. Its subgraph is replicated the specified number of times. Then the subgraphs are placed in the main flow graph and ordered linearly using control edges.

Strictly speaking, the passes described here are optional. They are not necessary for successful synthesis, but they do help to simplify the synthesis process because they simplify the flow graph structure. The synthesis techniques can focus on creating the logic from the represented behavior rather than on what kind of behavior is hidden in the flow graph structure. Of course, possible disadvantages are higher hardware costs and longer critical path delays in the final implementation compared to the results based on a more sophisticated flow graph.

5.2.3 Clustering

The passes described above manipulate the control flow in the flow graph. On the other hand, the *cluster* transformation works at the data flow level. Basically, it logically groups data flow nodes that compose a complete inter-module transfer.

In the flow graph of Figure 5-4, the address and data transfers are represented as two separate operations. Nevertheless, they are actually part of a larger transfer involving synchronization between the processor and memory module. Clustering looks for this situation in the flow graph. Specifically, it identifies concurrent data operations with input and output signals from common source or destination ports. To do this, as shown in Figure 5-8, all hierarchical control nodes are inspected for data flow nodes whose associated sync node depends on the common I/O protocols. Their sync nodes are then merged into one sync node and associated with each data flow node,

forming a clustered set of data flow nodes. The example shows the address and data assignment are associated to one sync node, implying that the individual transfers occur jointly with a interlocked sequence of I/O protocol events.

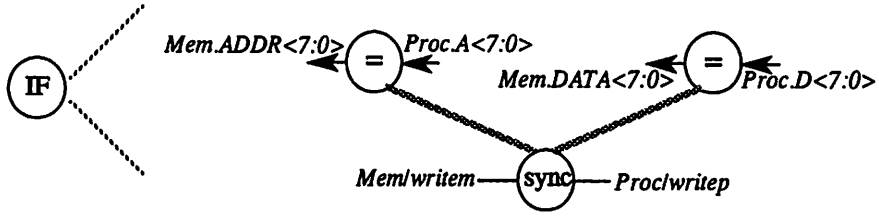


Figure 5-8 : Clustered Flow Graph for Processor to Memory Interface

In the previous example, data flow nodes that are data-independent were clustered because they have a protocol dependency. Sometimes, data flow nodes that have a data dependency but unrelated protocols are also clustered. For example, the flow graph from Figure 5-3 shows the delay @ node providing data to the logical AND node. Each of these nodes is associated with separate sync nodes linked to different protocols. If these data dependent nodes are coupled into one complex operation, then they can be clustered, as shown in Figure 5-9. The new flow graph shows that the processor port is synchronized to the memory port when the data operations are executed within one transfer cycle. In this example, the cluster transformation is only responsible for merging sync nodes and associating them with a clustered set of data operations. The actual decision to cluster data flow nodes that are protocol-independent is made during a later synthesis step.

Clustering merges sync nodes of related data flow nodes, transforming the original graph into a form more suitable for implementation. In this sense, it is can be considered a core synthesis step.

specification: *Mem/writem.DATA<3:0> = Proc/writep.D<3:0>@1 AND '1101';*

clustered flow graph:

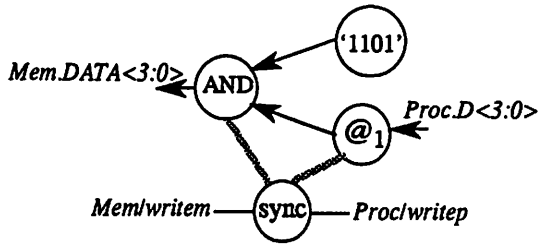


Figure 5-9 : Clustering Data Flow Nodes Into a Complex Operation

5.3 Flow Graph Examples

Using simple examples, the previous sections have described the design representation and the translation of an IDL specification into a flow graph. In this section, the representation method is applied to the first three examples presented in Section 4.4 of Chapter 4. The flow graph representations are more detailed in construction and appearance compared to their IDL specifications. This is expected, since synthesis refines the design specification by definition, and translation is the first step. These examples demonstrate how behavioral synthesis raises the level of abstraction at which a designer works by providing the front-end translator.

VME System Bus Interface

Figure 5-10 illustrates the flow graph that describes the inter-module communication between the VME system bus and a static RAM module. The corresponding IDL specification is shown in Figure 4-7 of Chapter 4.

There is an IF node for the write and the read access. The read IF node was an ELSE node before a flow graph pass was performed. Each IF contains three data flow nodes. The first is a constant node that models the transfer of a constant bit vector to the memory *WE_L* signal. Notice that a constant transfer does not map to a constant node that drives an assignment node, because the assignment node is considered redundant. The second and third data flow nodes are assignment nodes that

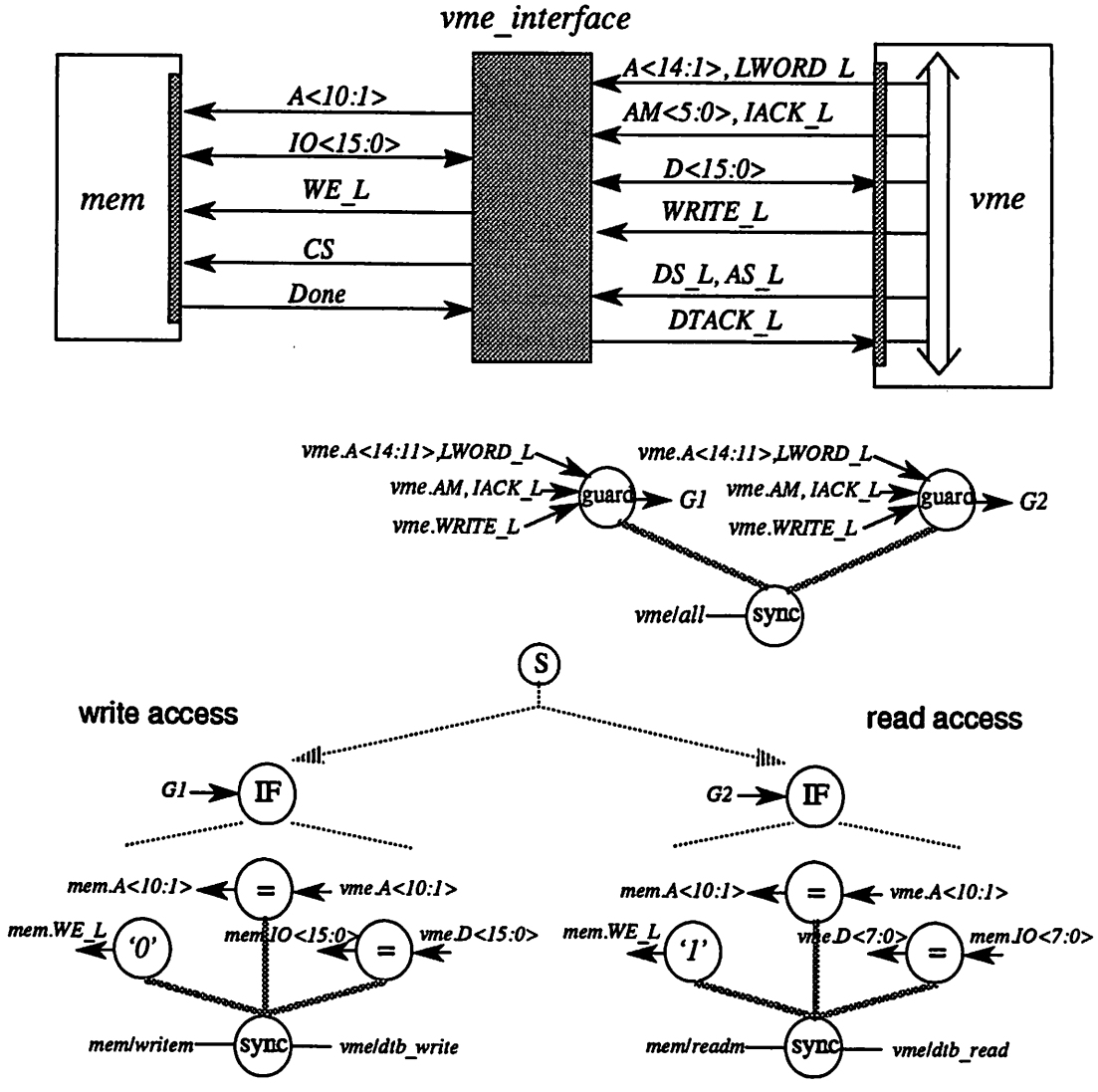


Figure 5-10 : Flow Graph for VMEbus Interface

model the address and data transfer. These three nodes are associated to one sync node representing the synchronization between the VMEbus and the memory obeying their write access protocols. The initial flow graph, constructed from direct translation, actually associates separate sync nodes with each of the three data flow nodes. The clustering transformation merges them.

In the flow graph, behavior represented on parallel control flow branches must be mutually

exclusive. So in the example, the two IF nodes can never be fired at the same time.

The decode function in the input specification is part of the condition expression within the IF statement. Upon translation into a flow graph, that behavior is contained within a guard node that fires the IF node.

In the block diagram, the VME control signals (*AS_L, DS_L* and *DTACK_L*) and the memory control signals (*CS_L* and *DONE_L*) are used for protocol signaling. These signals and the protocol event graphs do not appear in the flow graph, since they are hidden in the module library.

TMS320 to Optical Link Interface

The flow graph representing multiplexed communication between the TMS320 uni-processor and the TAXI optical transmitter (see Figure 4-8 in Chapter 4) is shown in Figure 5-11. The concurrent address and header transfer is captured in the top IF node, while the concurrent data and header transfer is contained in the bottom concurrent node. The sequential execution from the address to the data transfer, specified by the NEXT statement, is represented by the control edge connecting the top IF node to the concurrent node. Within the control nodes, the data flow nodes are clustered and associated with a sync node representing the synchronization between the TMS source and the TAXI destination.

In the IDL specification, both concurrent constructs are nested in the IF statement and occur only when the condition-expression evaluates to true. In contrast, the flow graph represents the first concurrent construct with an IF node, and the second concurrent operation with a concurrent node. It would seem that the second one could be captured as an IF node. However, this is unnecessary and redundant. In the flow graph, the IF node fires only when the input condition evaluates to true. Then, only after the concurrent transfers within the IF node have completed, the next concurrent transfers can execute. Since the control precedence implies that the second set is conditioned on the first, the flow graph behavior is equivalent to the specified behavior in the input description.

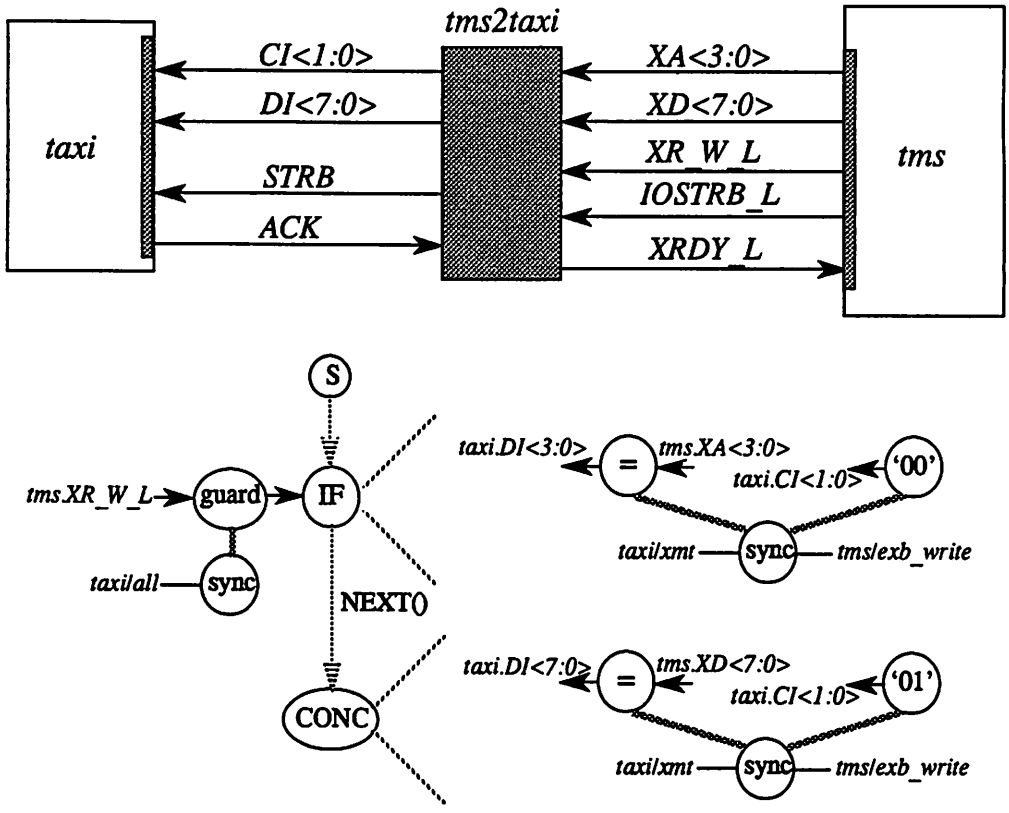


Figure 5-11 : Flow Graph for TMS320 to Optical Link Interface

A/D Module to Optical Link Interface

Figure 5-12 demonstrates how looped behavior is represented in the flow graph. It corresponds to the IDL specification shown in Figure 4-9 of Chapter 4 describing the communication between an A/D module and the TAXI optical link.

Translation from specification to flow graph first substitutes the contents of the procedure in Figure 4-9 into the specification body where the procedure call is made. The procedure content is just the transfer of a constant value. This transfer translates to a constant node contained in a concurrent node, as shown in the left flow graph of Figure 5-12. The following NEXT statement maps to a control edge, implying precedence between the first transfer and a block of transfers. The

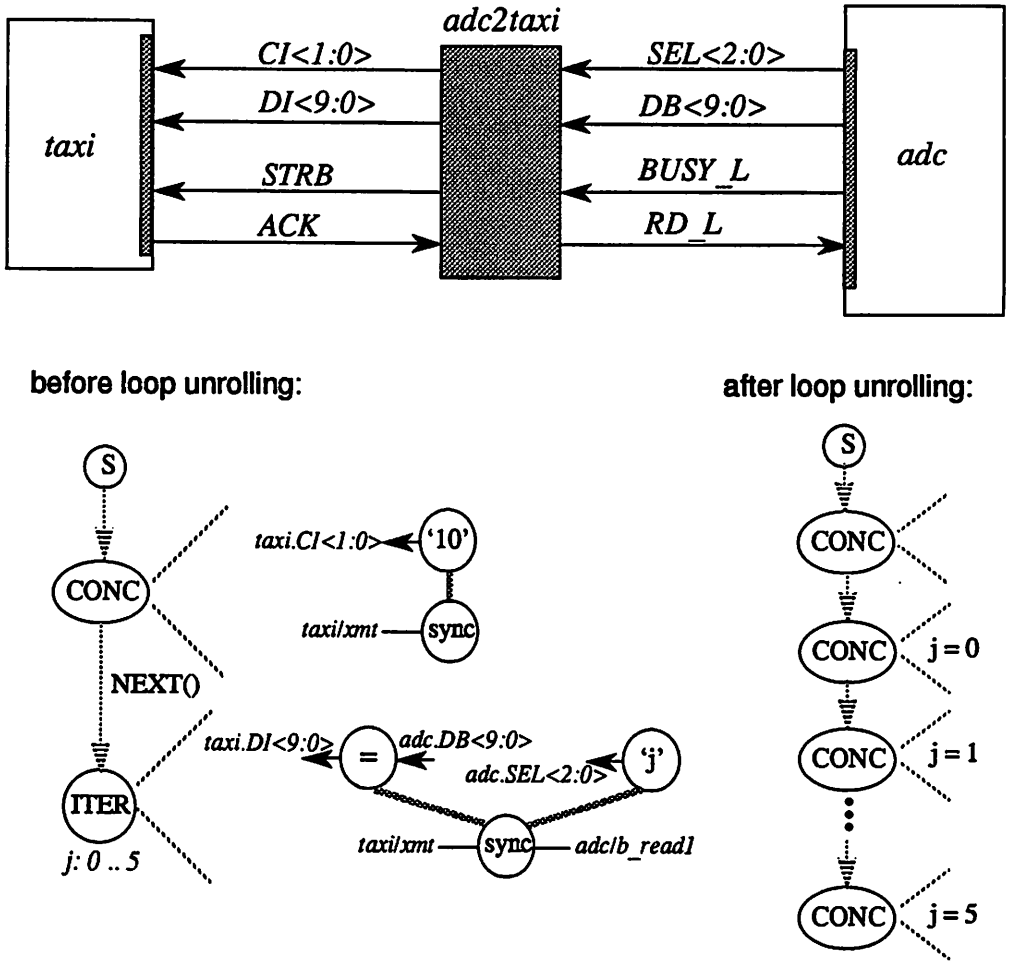


Figure 5-12 : Flow Graph for A/D Module to Optical Link Interface

hierarchical ITERATE node captures the address and data word transfer. When the loop node is unrolled, the assignment node for the address transfer is transformed into a constant node that takes on the actual value of the loop index variable. The left flow graph in the figure illustrates the flow graph after the unrolling pass. Unrolling creates a linear flow of six "concurrent" control nodes, containing the subgraph from the original ITERATE node.

5.4 Summary and Implementation Issues

The flow graph design representation is implemented using the HYPER flowgraph policy [Rabaey90]. It can also be stored in the OCT database. In fact, the translator outputs a textual description of the constructed flow graph in HYPER's AFL format.

The translator constructs a hierarchical flow graph from the behavior contained in the ROUTINE declaration of the input specification. At the top level, control edges originate at the start node and flow from control node to control node. Figure 5-13 shows an example of a top level flow graph. Control nodes represent conditional, concurrent and loop behavior while the control edges

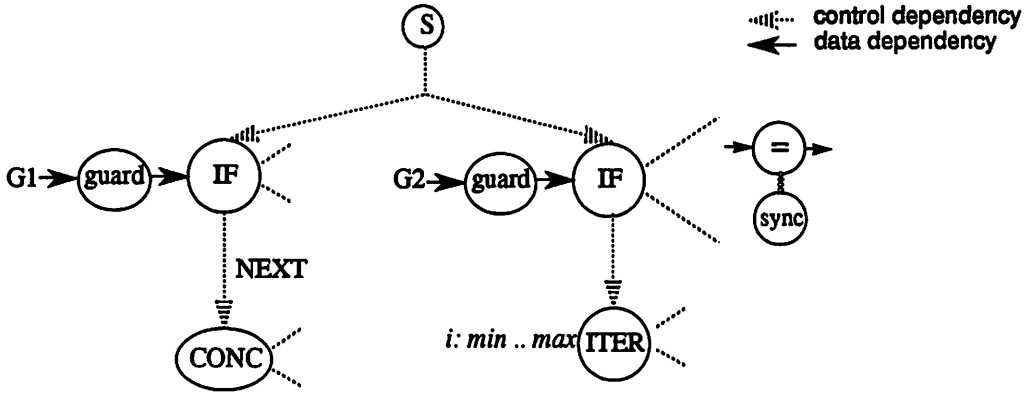


Figure 5-13 : Example of Control and Data Flow Hierarchy

represent precedence. The start and conditional control nodes are fired by a guard node that generates a token based on input conditions. When fired, the data and control flow within the scope of the control node are allowed to execute. The end of the control flow occurs at a control node that has no output control edges. So, the combined use of control nodes and edges at the top level represents when communication actions can happen. The data flow nodes and edges within the hierarchical nodes represent how the actions happen.

The design representation described is flexible and describes a wide variety of behavior. However,

the current implementation of the synthesis software imposes a few restrictions on the flow graph structure. First, only one level of hierarchy is accepted in the flow graph. The main flow graph contains hierarchical control nodes and precedence edges. Within the control nodes are the primitive data flow nodes. No more levels of hierarchy are allowed. So, if the behavior in the input specification results in a flow graph with multiple levels of hierarchical nodes, the translator will have to flattened the original graph until one level of hierarchy is created. Actually, practically all behavioral synthesis systems in current use require flattened flow graphs [Walker91]. The second restriction is related to control precedence. Behavior represented on parallel branches in the control flow must be mutually exclusive. So, in Figure 5-13, the input conditions labeled G1 and G2 are never true at the same time. This assures that nodes on both branches do not execute simultaneously. Both these restriction make the flow graph compatible with the current state of the synthesis software, and are not limitations of the design representation.

Besides performing a one-to-one mapping from the input specification to the flow graph, the translator performs elementary checks for inconsistencies and basic optimizing transformations. Examples are type checking, converting integer types to binary vectors, in-line expansion of procedures and loop unrolling. However, the current translator implementation lacks sophisticated checks and optimizations, such as checking for multiple assignment within a concurrent construct, dead code elimination and common sub-expression elimination. Inclusion of such abilities into the translator implementation will promote it into a compiler. For this reason, the term "translate" was chosen over "compile" to describe the front-end of the synthesis tools.

CHAPTER 6

SYNTHESIS FROM FLOW GRAPHS

Starting from the system communication behavior of an interface module and a set of I/O protocol constraints, the core steps of behavioral synthesis find a register-transfer structure that implements the intended behavior while meeting the constraints. The flow graph representation describes the communication behavior, while the event graph captures the protocol constraints in the module library.

Synthesis is a three phase process that refines abstract behavior into a structural implementation, as illustrated in Figure 6-1. First, the initial flow graph is transformed by performing clustering, scheduling and allocation. These techniques adjust the flow graph characteristics to create or optimize the system-level data and control flow implementation, while preserving the specified behavior. Second, event graphs corresponding to the I/O protocols of interacting modules are interlocked to create event sequences for protocol synchronization. The interlocked event graphs are generated according to the data flow described in the transformed flow graph. Finally, from the final flow graph and the interlocked events graphs, the behavior is mapped into a structure. The next three chapters present these core synthesis steps.

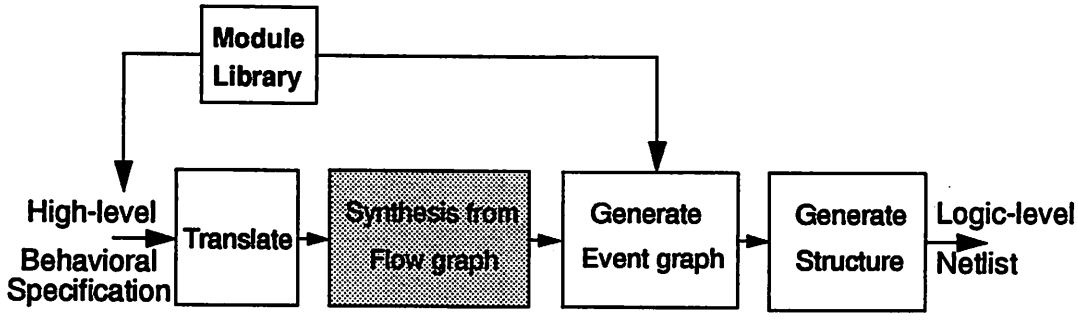


Figure 6-1 : Synthesis from Flow Graph and Interface Generation

This chapter focuses on the flow graph transformations. Among them, clustering was described in the previous chapter. Scheduling and allocation in ALOHA are the main topics here. The input to the synthesis transformations is a flow graph translated from the input specification. The output of synthesis is a final flow graph ready for mapping onto structure. The first section explores the key problems and objectives of scheduling and allocation. The next two sections presents the scheduling and allocation techniques. The last section provides examples of transformed flow graphs.

6.1 Synthesis Issues

At this point of interface generation, the flow graph represents the communication behavior to be implemented in hardware. The behavior consists of data flow and control flow from the input specification. For example, Figure 6-2 illustrates the interface presented in Section 4.4 that demultiplexes two words, arriving sequentially from a TAXI optical receiver module, to the destination module which is a bank of D/A conversion units. As shown in the flow graph, the first word contains the address of a specific bank, and it is delayed by one transaction cycle. The second word contains the actual data to be written into a D/A unit. After the address is delayed (control flow), it is decoded into the individual select lines using the "decode" function, and both the data

and select words can be sent to the D/A module (data flow). To simplify the illustration, the hierarchical control nodes have been flattened and the sync nodes omitted.

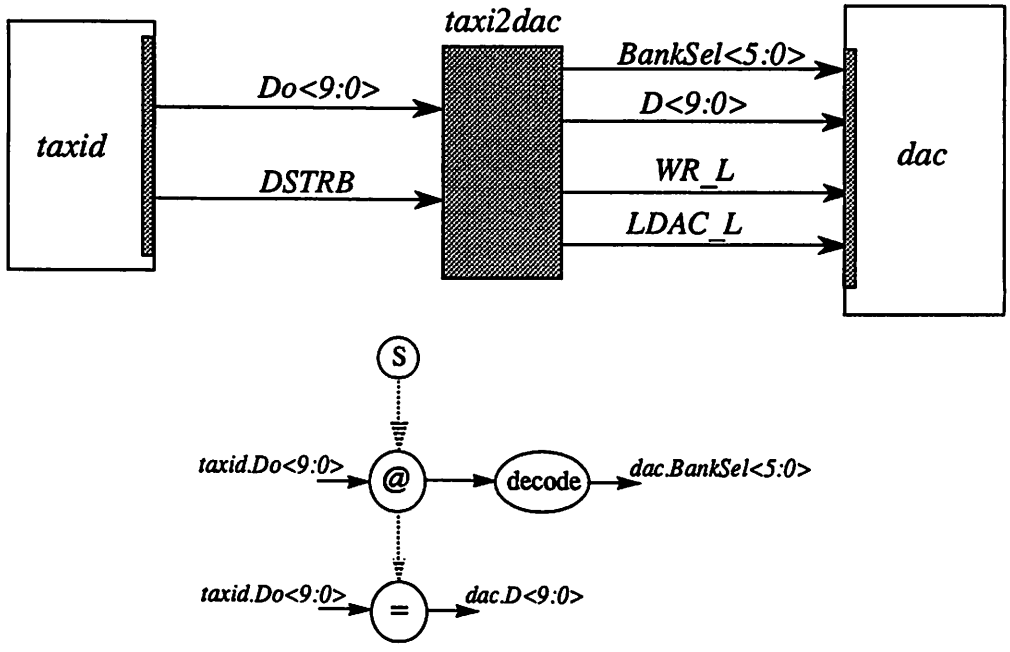


Figure 6-2 : Flow Graph for Demultiplexed Transfers

The data and control flow captured in the initial flow graph are just the minimum high-level constraints that the final implementation must fulfill. The exact time at which an operation is executed and the exact hardware element which executes the behavior are still open. To illustrate this, Figure 6-3 shows two possible solutions for the above example. The left-side solution is obvious. In the first time step, the address word is delayed by storing it in a register. In the second time step, the stored address is decoded by combinational logic and sent with the arriving data word to the destination on parallel busses. The precedence from the delay to the decode operation is imposed by data dependency, and the precedence from the delay to the data assignment is derived from explicit control dependency in the initial flow graph. The right-side solution is less obvious. The decode function is executed first and its result is delayed by storing it in a register. During the second time step, the result in storage is exported to the destination along with the data

word on parallel busses. Both the solutions achieve the original behavior, although they swap the execution time for decoding and use registers of different widths.

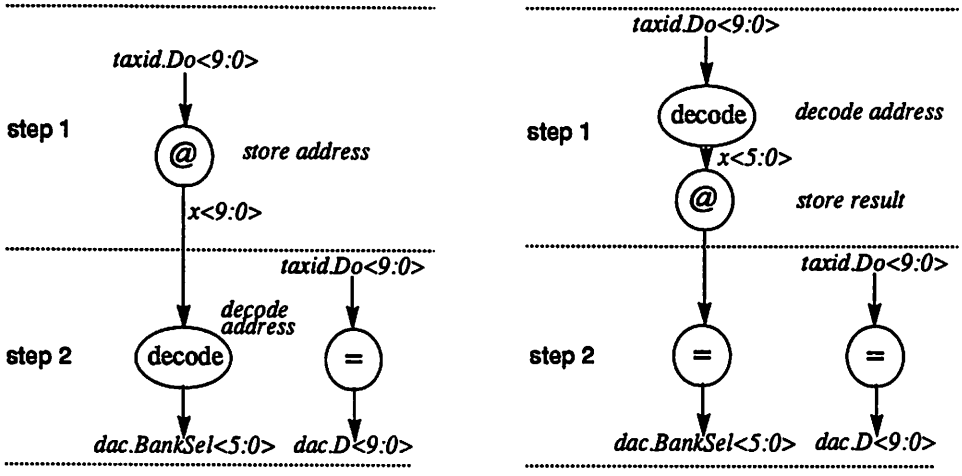


Figure 6-3 : Steps in Flow Graph for Demultiplexed Transfers

The example given demonstrates the key issues at this point in synthesis. To actually implement the behavior in hardware, synthesis needs to specify the time and the hardware resource that executes the operation. Scheduling and allocation are flow graph transformations that address these two problems, respectively. Since there is usually a solution space that satisfies the high-level constraints, synthesis guides the transformations according to an objective and a model of the hardware structure. These are discussed below.

6.1.1 Objectives

Scheduling puts each flow graph operation into a *control step* for execution. A control step is the basic unit of time in a sequence of operations. For example, the time steps shown in Figure 6-3 are control steps. In the execution model used by ALOHA, a control step corresponds to one transaction cycle, and the beginning of a cycle is marked by a protocol event. The duration of the cycle can vary from one to the next. In comparison, control steps for synchronous systems

correspond to a clock cycle. This transformation produces a *schedule* which is just the sequence of control steps and the operations within them, such as the two shown in Figure 6-3.

Resource allocation determines the number and type of hardware elements required to implement the data flow. Hardware elements include functional units such as adders and decoders for data operations, storage units such as registers and latches for signal values, and wires and busses for transfers. The accompanying problem of assigning a specific hardware instance to a data flow operation is known as resource assignment or module binding. In this chapter, allocation will be used to collectively refer to both problems. Allocation is a simple process for interface applications because they tend to be control-oriented rather than computation-oriented, unlike DSP applications which tend to have the opposite characteristics.

Scheduling and allocation are closely interrelated, making the problem complex. A short schedule will likely require more hardware resources than a long schedule where few resources are shared. Likewise, allocation places constraints on scheduling, causing a circular dependency. Few behavioral synthesis systems perform scheduling and allocation in only two separate phases. Most switch between the two tasks iteratively until an optimal schedule is produced [McFarland90]. Behavioral synthesis from flow graphs in ALOHA takes a similar strategy, and it is broken into two main steps:

a. Generate an initial schedule that satisfies all data dependency and control precedence constraints in the original flow graph. Then allocate hardware units for the initial schedule.
b. Optimize the resource allocation and schedule producing the final schedule.

The technique for generating the initial schedule has been developed specifically for interface applications, while the scheduling and allocation optimizations are performed with existing techniques developed by the behavioral synthesis research community.

Constraints are just conditions that the generated schedule must meet. Optimizations seek to minimize or maximize some measure of quality of the schedule. For example, the goal can be to

minimize the implementation area or the critical path, or it can be to maximize throughput and hardware utilization. Many existing synthesis systems seek to minimize area given a throughput constraint [McFarland90]. Interface applications tend to have datapaths that store and route information rather than compute results from the information. Accordingly, the datapaths are dominated by storage and interconnect elements, such as busses and multiplexers, with some functional units like adders and arbitrary combinational logic. In comparison, DSP applications are dominated by register files, adders, multipliers and shifters. Because interfaces facilitate inter-module transfers, the synthesis method used in ALOHA seeks to minimizes the critical path through storage and interconnect elements.

There are three other tasks that support scheduling and allocation. The first is partitioning large design specifications among multiple interface modules. This is difficult at the system level, because more than one type of technology can be used, such as various FPGAs, TTL and other technologies especially for system interconnects. The second task is estimating the throughput, area (cost) or hardware utilization of design alternatives. The final task is supplying information on the technology library of available cells. These three are not discussed in detail, since this chapter concentrates on the overall synthesis task.

6.1.2 Interface Template

The overall objective of scheduling and allocation is to bring behavior closer to a structural implementation in the space of possible solutions. In addition to the flow graph execution model, the two tasks are directed by a register-transfer model of the target structure. The structural or architectural model is an interface template consisting of a datapath and *two* controllers implemented with asynchronous sequential machines (FSMs), as shown in Figure 6-4. The datapath provides the physical link for allocated inter-module transfers, and it executes data operations. The interface FSM controls the schedule of transfers and internal operations, while the protocol FSM executes I/O protocols to synchronize the transfers. The three blocks operate

concurrently, but the interface controller FSM configures and initiates any action in the datapath or protocol controller FSM. In contrast, the structural model used in traditional synthesis techniques contains a datapath and one synchronous FSM. The main difference between the interface template and the traditional model is the additional FSM for protocol control.

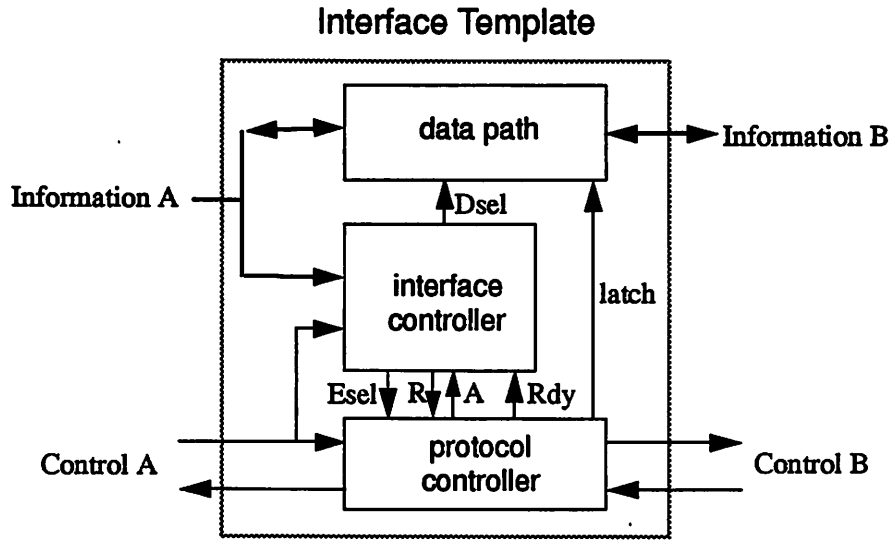


Figure 6-4 : The Target Structure

Side A of the interface links the master ports to the slave ports on side B. The template supports multiple source and destination ports. Details of the template blocks, the interaction between the blocks, and the relationship of scheduling/allocation to the interface template are described below.

Datapath

The data path block implements data flow behavior. It contains the register-transfer units that actually compute and move information between source ports and destination ports. Assignment and data operation nodes in the flow graph can be mapped to parameterizable functional, storage and interconnect units. Functional units include adder elements and combinational logic units. Parameters would be the width of the adder or the name of a BDS file for a combinational logic unit. Storage units consist of registers or latches with separate input and output terminals. The

parameter describes the width of the register or latch. The protocol controller derives clocks for latching information into storage elements from asynchronous events. Currently, register files or multi-port registers are not supported. Interconnect units are tri-state buffers, multiplexors, wires and busses. The last two can be uni-directional or bi-directional.

Allocation is the first step toward creating a datapath structure. It determines the number of hardware units that will make up the datapath, and in this sense it determines the complexity and speed of the datapath. As the interface controller sequences through the schedule of inter-module transfers and data operations, the datapath is configured on each control step with the Dsel configuration word, as shown in Figure 6-4.

Protocol Controller

The protocol controller sequences and times protocol events to synchronize clustered transfers. Specifically, it contains the logic that executes behavior specified in the event graphs, including time constraints. The protocol controller links the control signals of source and destination ports, whereas the data path links the information signals. Each sync node in the flow graph has a corresponding event graph to represent the protocol synchronization. When the datapath exectutes transfers associated with a sync node, the protocol controller realizes the event graph behavior. An event graph is enabled by the Esel select word issued by the interface controller, as illustrated in Figure 6-4. The Esel word is accompanied by the 4-phase handshake signals, R and A. The next chapter describes how event graphs are generated from the sync nodes.

Scheduling is the only high-level transformation that is related to the protocol controller. As explained in Chapter 5, Section 5.2.3, a set of data flow operations in the flow graph that are data dependent but protocol independent may be clustered. This type of clustering is performed during scheduling. Effectively, scheduling has determined that the set of data operations are put into one control step rather than over several. This may or may not simplify the protocol controller logic, but it will reduce the latency of the corresponding datapath.

Interface Controller

Scheduling is the first step toward creating the interface controller FSM. The central controller implements the schedule and initiates the joint action of the data path and protocol controller. Synthesis produces a hardwired FSM, and a control step in the schedule corresponds to a single state of the FSM. On each control step, the controller issues the Dsel word to configure the data path, and it issues the Esel word to select the associated event graph in the protocol controller. The FSM uses information signals, such as address and write status, to compute input conditions (guards) that determine which of several mutually exclusive states to branch to. These states correspond to the IF behavior.

In the interface template, the beginning of each control step, or FSM state, is marked by an asynchronous *local clock* event, rather than a synchronous clock event. The controller generates the local clock from two signals. The first is the Rdy signal issued by the protocol controller to indicate that it is done executing the current event graph. The second is a completion signal, not shown in the figure, that indicates the combinational logic internal to the FSM is done computing the next state and the Dsel and Esel configuration words. This scheme is similar to the local clock scheme used in [Hayes81].

6.2 Generating a Schedule

The initial scheduling phase takes a flow graph translated from the input specification, and constructs a minimal schedule that meets all the data dependency and control precedence constraints in the original specification. Upon completion, the original flow graph has been transformed into a flow graph with a sequence of control steps that govern control flow and that contain data flow nodes. The resulting schedule may require more than the minimum amount of hardware resources to achieve a given throughput, but the schedule is improved during a later optimization phase.

6.2.1 Creating Control Steps

Scheduling starts by creating control steps in the flow graph to place the control and data flow nodes. This is straight-forward because it is directly supported by the flow graph design representation described in Chapter 5. The flow graph consists of hierarchical conditional (IF) and concurrent (CONC) type nodes that are arranged in branches and sequences linked by control precedence edges. For example, the flow graph in Figure 6-5 shows two branches of control flow. The corresponding IDL description is also shown, but the sync nodes are omitted to simplify the illustration.

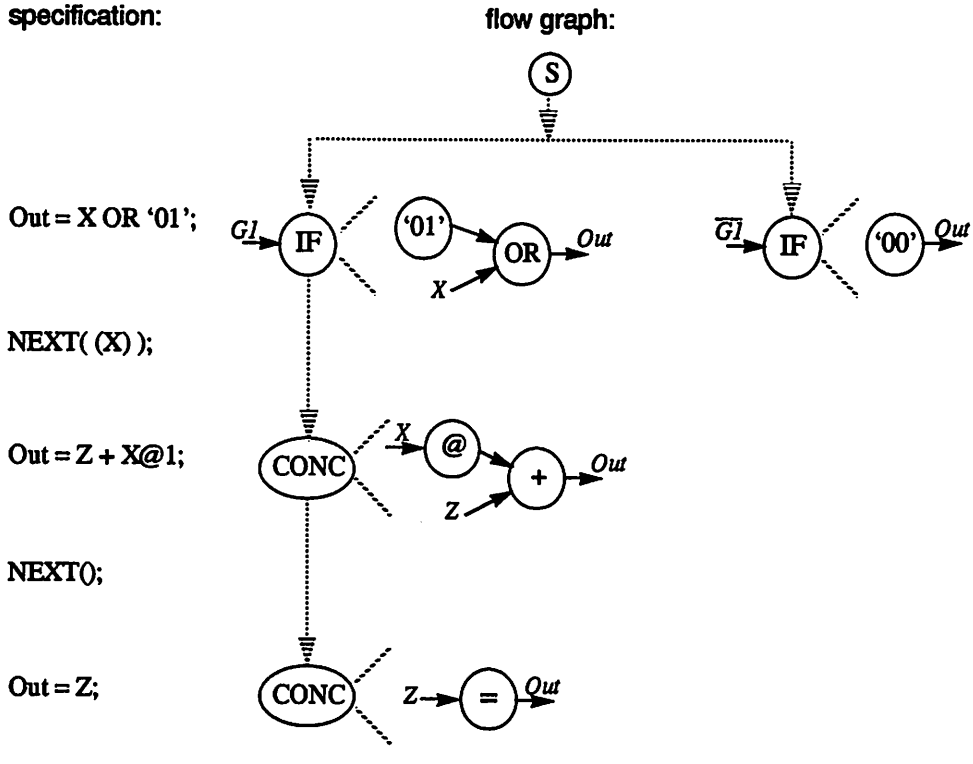


Figure 6-5 : Initial Flow Graph

A hierarchical node only contains data flow operations, which can be executed in one control step. So, as the flow graph is traversed from the start node down the precedence edges of each branch, a control step is created for each hierarchical control node, and the data flow operations within are

assigned to that control step. This is shown in Figure 6-6. Control step 1 has two possible branches which are mutually exclusive. In effect, the control nodes and precedence edges in the flow graph represent a default schedule that is maximally concurrent.

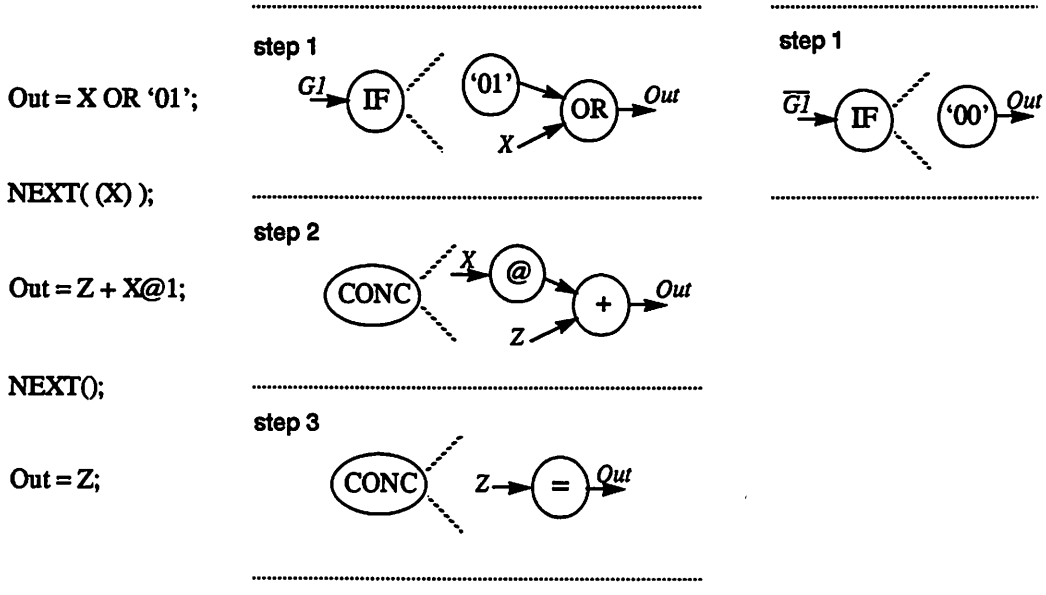


Figure 6-6 : Scheduling of Control and Data Flow Operations

6.2.2 Scheduling Inputs and Outputs

The previous step scheduled the control and data flow *operations*. The next step is scheduling when *inputs* to data flow operations are accepted by the interface and when the *outputs* are exported. This is necessary because inputs from source ports can arrive earlier than the execution time of the operation, and outputs to destination ports can happen at a later time than the data operation.

Determining from which control step an input value originates is more complex than creating control steps or scheduling outputs. Starting from the flow graph produced in the previous step, input scheduling visits every data flow operation, ignoring the delay operations. At each operation,

it considers the input edges directly driven by source ports or the input edges driven by a delayed version of source information. At this point in the process, input scheduling can encounter one of three cases, as illustrated in Figures 6-6.

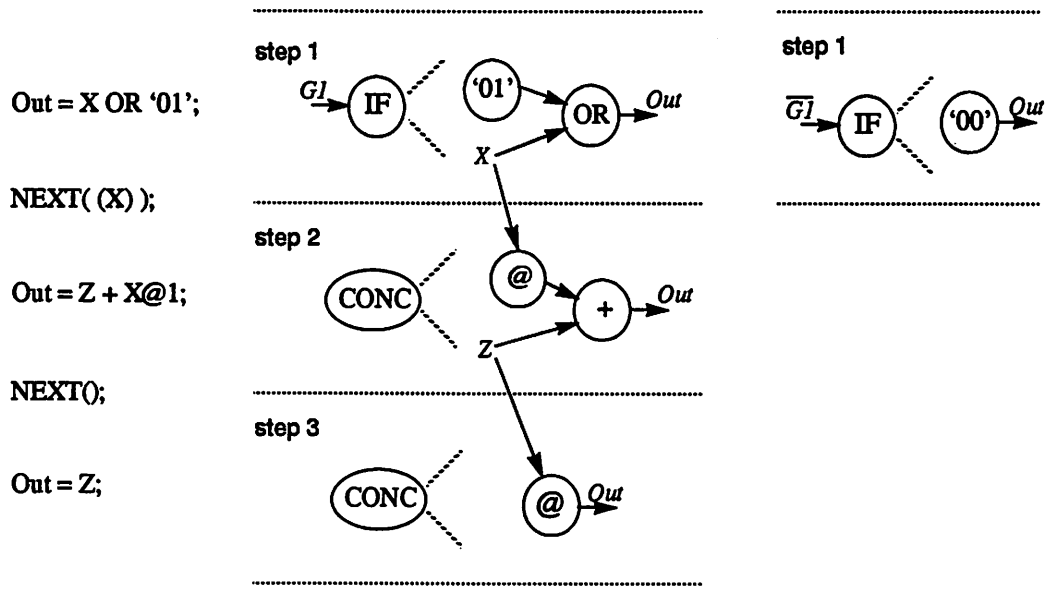


Figure 6-7 : Scheduling of Inputs and Outputs

In the first case, an input value arrives during the same control step that the operation it drives is scheduled into. Such inputs are scheduled into the same control step. As shown in Figure 6-7, scheduling accepts the *X* input value to the OR node during the same control step. The input from the *Z* transaction is scheduled into the same control step that the addition operation occurs. Comparing Figure 6-6 to Figure 6-7, the scheduling done in this case is obvious and does not alter the flow graph schedule.

In the next two cases, a source port generates an input value during an earlier control step than the operation is scheduled into. In Figure 6-6, the *Z* input value to the assignment node in the third control step is actually from a transaction occurring in the second control step, and this demonstrates the second case. Because of the delay node, the input value to the addition operation

is generated during a prior *X* transaction, and this illustrates the third case. The only difference between the two is that the latter uses a delay node.

In either case, the inputs are scheduled into the appropriate control step, as shown in Figure 6-7. In the flow graph, inputs originating from prior transactions are scheduled with a delay node. The delay node is placed in the same control step as the data flow operation of interest. The input to the delay node is the source transaction from the prior control step, and the output drives the operation node. The difference in control steps between the source transaction and the operation node is the parameter value of the delay node. In effect, the delay node represents a delayed version of a input value scheduled into a prior control step. In Figure 6-6, both the *X* value to the addition and the *Z* value to the assignment originate from prior transactions. Accordingly, scheduling delays the input values by one control step, as illustrated in Figure 6-7. In the latter case, a delay node is created and substituted for the assignment node, and the input edge to the delay node is connected to the appropriate prior control step. In the former case, the delay node already existed, so only the input edge to the delay node is connected according to the input schedule.

Scheduling an output value is straight-forward. An output value of a data flow operation is always scheduled for export in the same control step that the operation is in. So, output values are exported as soon as possible. For example, in Figure 6-7, the OR, addition, and assignment operations send their output values to the destination port *Out* during the same control step that they are executed. There is no danger of contention over a destination signal, because the input specification does not allow multiple assignment to a destination within a concurrent construct. Scheduling outputs is a trivial task, and the flow graph remains unaltered.

6.2.3 Initial Allocation

The initial scheduling is followed by an initial resource allocation step. Using the scheduled flow graph, data flow operations are assigned to hardware units. Nodes representing arithmetic and combinational logical operations are assigned to functional units. These include adders for

arithmetic nodes, elementary logic gates for logic nodes, and a complex logic unit for each function node. A property is attached to the data flow node indicating the type of functional unit realizing the operation.

As described above, an input value generated by a source transaction during a control step may be used by an operation in a subsequent control step. Allocation provides a storage element to temporarily capture that input value. For example in Figure 6-7, the *Z* input feeds a delay node in control step 3, and the *X* input drives a delay node in control step 2. Since the delay node represents the need for temporary storage, it is *moved* to the control step that contains its source transaction, illustrated in Figure 6-8. The output of the delay node is still the input to the subsequent data flow operation in the original control step. If necessary, an assignment node is used to export values stored from a prior control step, such as the one shown in control step 3. The parameter value of the delay node represents the lifetime of the stored value.

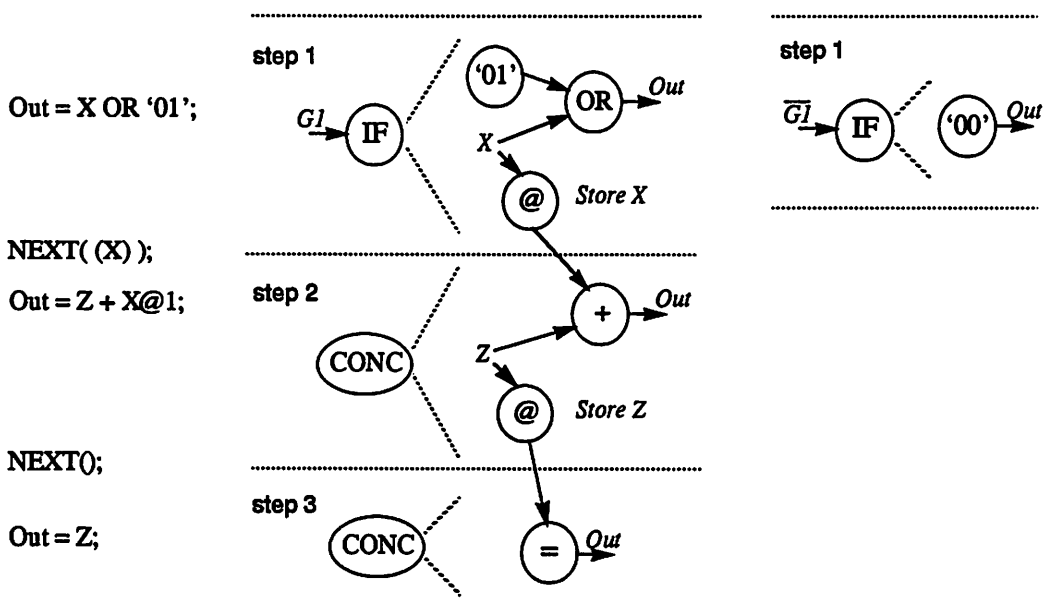


Figure 6-8 : Initial Schedule and Allocation

Interconnect elements realize the assignment and constant operations. Assignment nodes represent transfers and are assigned a wire or bus that connects the input value to the output value. In contrast, constants are assigned a wire or bus that is hardwired to the appropriate combination of logic high or low levels. Other interconnect elements include multiplexors and tri-state buffers. Output signals that can be driven by multiple input signals require a multiplexer to resolve the contention over the output bus. Some signals may be bi-directional or set to a high-impedance value. The data edge in a flow graph depicts signals as strictly input or output, but it carries a property indicating the bi-directional or high-impedance condition, and tri-state buffers are assigned to signal busses. Multiplexors and tri-state buffers are actually allocated at the final step of behavioral synthesis, described in Chapter 8.

Although relational or guard nodes represent data operations, they are not implemented in the datapath of the interface template. Instead, the behavior is realized in the interface controller. The sync node is actually realized by the protocol controller. Since allocation assigns datapath hardware units, these three nodes are ignored and considered during later synthesis steps.

The initial allocation process provides sufficient hardware units to implement the initial schedule, but not the minimum number. Functional and storage units can be shared among different data flow operations. For example, a single ALU can execute the logical OR and the addition operations in Figure 6-8, even though allocation assigns separate hardware. A storage element that temporarily stores a data value can be reused for another data value as long as the lifetimes of both values do not overlap. These are optimizations made in the next synthesis phase. What is important about the initial allocation is that it provides an upper bound on the number of hardware units required to implement the initial or optimized schedule. The measure can be used to guide scheduling optimizations.

These steps conclude the initial scheduling and allocation phase. The initial schedule meets all the data and control flow requirements specified in the original flow graph and is maximally

concurrent. Although it is not optimal, the schedule and resource allocation can be mapped onto register-transfer logic in the interface template. For an improved implementation, optimizations to the schedule are made before mapping to hardware.

6.3 Scheduling and Allocation Optimizations

Initial scheduling puts each data flow operation into a control step. Initial allocation assigns each data flow operation a dedicated hardware unit, such as an adder, latch or bus. Schedule and allocation optimizations reduce the length of the schedule and minimize the critical path through the datapath to be implemented. This measure of quality is chosen over minimizing area, because interface applications emphasize inter-module transfers over internal data computation. This characteristic causes datapaths to be dominated by storage and interconnect elements with some functional units. Although functional units tend to occupy more area than storage or interconnect elements, there are few of them, and even the initial schedule and allocation will yield a satisfactory implementation in terms of area.

6.3.1 Allocation issues

Allocation aims at minimizing the number of hardware components used to implement the given schedule while minimizing the critical path. The upper bound on the number of each type of hardware element was found during the initial allocation phase. These numbers can be used as a starting point. With the upper bounds, the schedule is optimized (described below). If rescheduling is successful, then the numbers can be reduced and a new schedule produced. This process of allocation followed by scheduling is iterated until the operations can no longer be rescheduled within the given resource limits.

The most important goal is to achieve a satisfactory compromise between the number of hardware elements and the critical path delay, because the two aims are conflicting. For example, ALU units

can be shared among arithmetic and logical operations that occur in different control steps, reducing the number of functional units. Similarly, input values with non-overlapping lifetimes can be assigned to the same register, reducing the number of storage elements. However, the savings in area complicates the interconnection paths. The functional and storage elements require multiplexors at their inputs since they are shared among multiple sources. Their outputs may fan out to multiple destinations, increasing the capacitance on wires and busses. The cumulative effect is increased delay on the critical path.

6.3.2 Scheduling Issues

The schedule is optimized by moving (rescheduling) data flow operations among control steps. The earliest time an operation can execute is the control step in which the last input arrives. The latest an operation can execute is the control step in which the output must be produced. So, in general, there is a window of consecutive control steps that the operation can be scheduled into while satisfying data dependency constraints. This is illustrated by the flow graph on the right side of Figure 6-9. The inputs to the add operation come from prior control steps and are captured by the delay nodes. The operation can execute in either the second or third control step. In Figure 6-8, all the data flow operations have only one possible control step they can execute in while satisfying the given data dependency requirements.

The initial schedule places each operation in the same control step as its output. In effect, operations have been scheduled for execution as late as possible (ALAP). Rescheduling the data operation nodes can use any classical scheduling technique developed for conventional flow graphs. These techniques include as soon as possible (ASAP) scheduling [Thomas83][Trickey87], list scheduling [Kramer90] [McFarland86] [Pangrle87] [Parker86] [Thomas90], freedom-based scheduling [Parker86], and force-directed scheduling [Cloutier90][Paulin89], listed from simplest to most complex [McFarland90].

Currently, ALOHA uses ASAP scheduling. This algorithm improves the schedule by taking

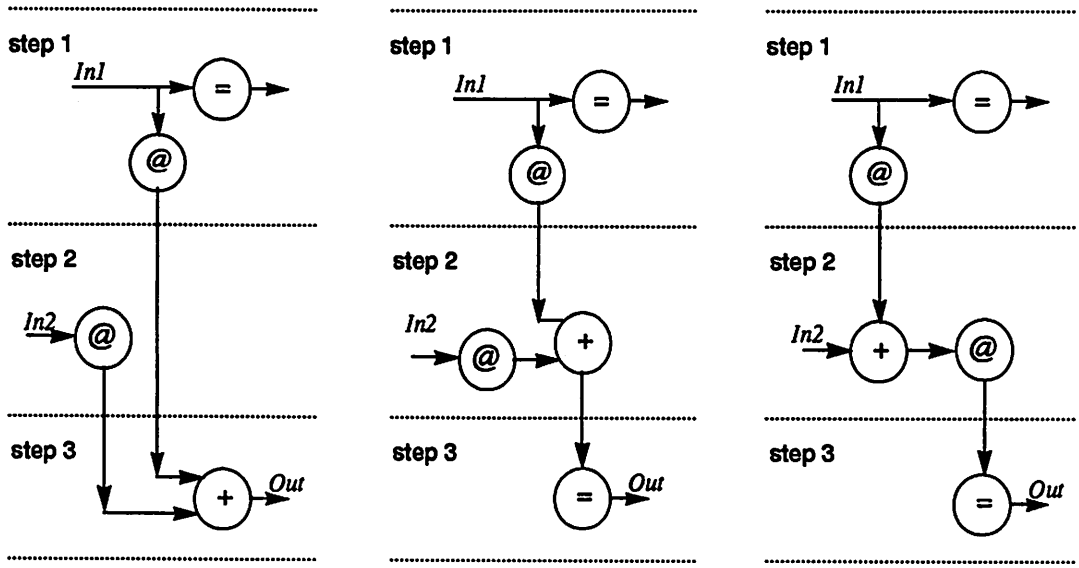


Figure 6-9 : Scheduling Optimizations

operations one at a time, and placing them into the earliest control step possible, within the window of control steps and under resource allocation limits. For example, in Figure 6-9 the addition is moved from the third control step to the second step, as shown in the middle flow graph. At this point, an optional transformation based on retiming decides if the In2 input to the add operation is captured in temporary storage or if, equivalently, the result of the addition is stored. If it is the latter, the delay node that captures the In2 value can be moved to the output of the addition operation, as shown in the last flow graph of Figure 6-9. Retiming maintains the equivalent input/output behavior as the original flow graph. Rescheduling an operation also requires house-keeping tasks such as inserting an assignment node in the third control step to export the addition result. In Figure 6-8, the data flow operations can only be executed in the control step that they are initially scheduled, so the flow graph remains unchanged. From experience it has been found that ASAP scheduling produces reasonably efficient results for interface applications.

List scheduling in which operation nodes are prioritized will produce even more optimal results than the ASAP algorithm, but it is more complex to implement [McFarland90]. Interface applications that are more compute intensive, such as application-specific I/O processors, will benefit most from list scheduling. ASAP scheduling easily meets the design requirements of interfaces as complex as DMA controllers.

After the allocation and scheduling optimizations, the original flow graph has been transformed into a final flow graph that can be mapped to register-transfer logic based on the interface template. The final schedule defines states of the interface controller FSM, while allocation defines the number and types of hardware units in the datapath.

6.4 Examples of Scheduling and Allocation

This section presents four examples of scheduling and allocation, and shows how the two tasks brings behavior toward a structural implementation. The first example is from Figure 6-2. The other three are based on the flow graphs, translated from the input specifications, in Section 5.3. The scheduled and allocated flow graphs should be compared to the original ones. It is important to keep in mind two points. First, the flow graphs represent the system communication behavior, which is the interface behavior. Second, the flow graph is implemented with the interface template, which is the internal structure of the interface block shown in the following block diagrams.

Optical Link to D/A Module Interface

The initial flow graph for a module that interfaces the TAXI optical link to a bank of D/A converters was shown in Figure 6-2. The TAXI link has a single bus for address and data words, and the D/A has a parallel address and data bus. The flow graph describes two sequential transfers which demultiplex address and data words from source to destination. The corresponding scheduled and allocated flow graph is illustrated in Figure 6-10. The combinational decode function has been scheduled into the first control step, which is the earliest possible time it can be

executed. The address value to the decode function is captured in temporary storage to maintain the integrity of the decode result, which is exported in the following control step. The other alternative is to store the decode result instead of the decode input. The first was selected so that the source port is released as soon as possible. The data word transfer is scheduled into the second control step. The schedule consists of two control steps, so the interface controller FSM sequences through two states.

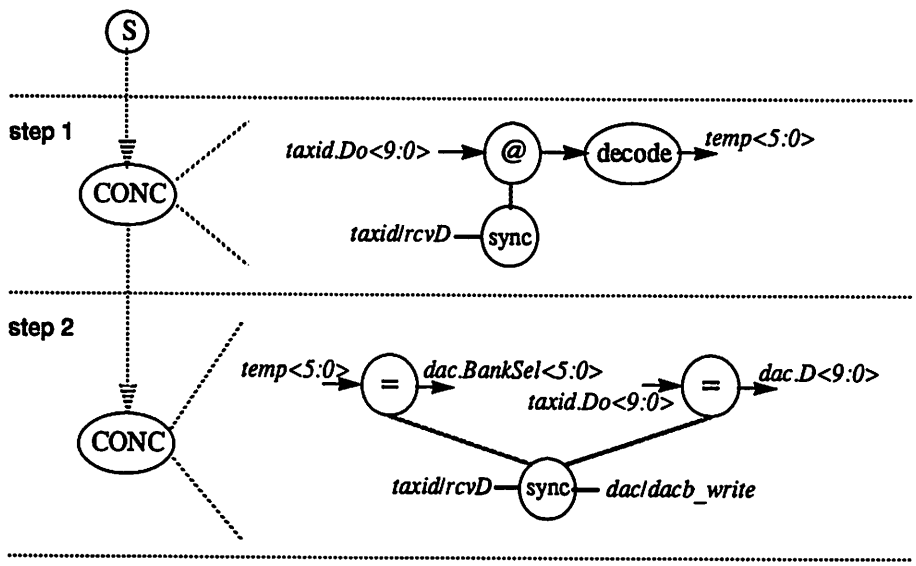


Figure 6-10 : Schedule for the Optical Link to D/A Interface

VME System Bus Interface

Figure 6-11 illustrates the scheduled flow graph for inter-module communication between the VME system bus and a static RAM module. There is really no difference between the original clustered flow graph and the scheduled one, other than the formality of control steps. Both the read and write access, contained in an hierarchical IF node, are scheduled into separate control steps. Unlike the previous example, these control steps are mutually exclusive rather than sequential. The interface controller monitors the address and write status inputs, and it generates the guard signals

which choose the control step to branch. Because there is no sequential control in this schedule, optimizations actually reduced the FSM to a purely combinational controller.

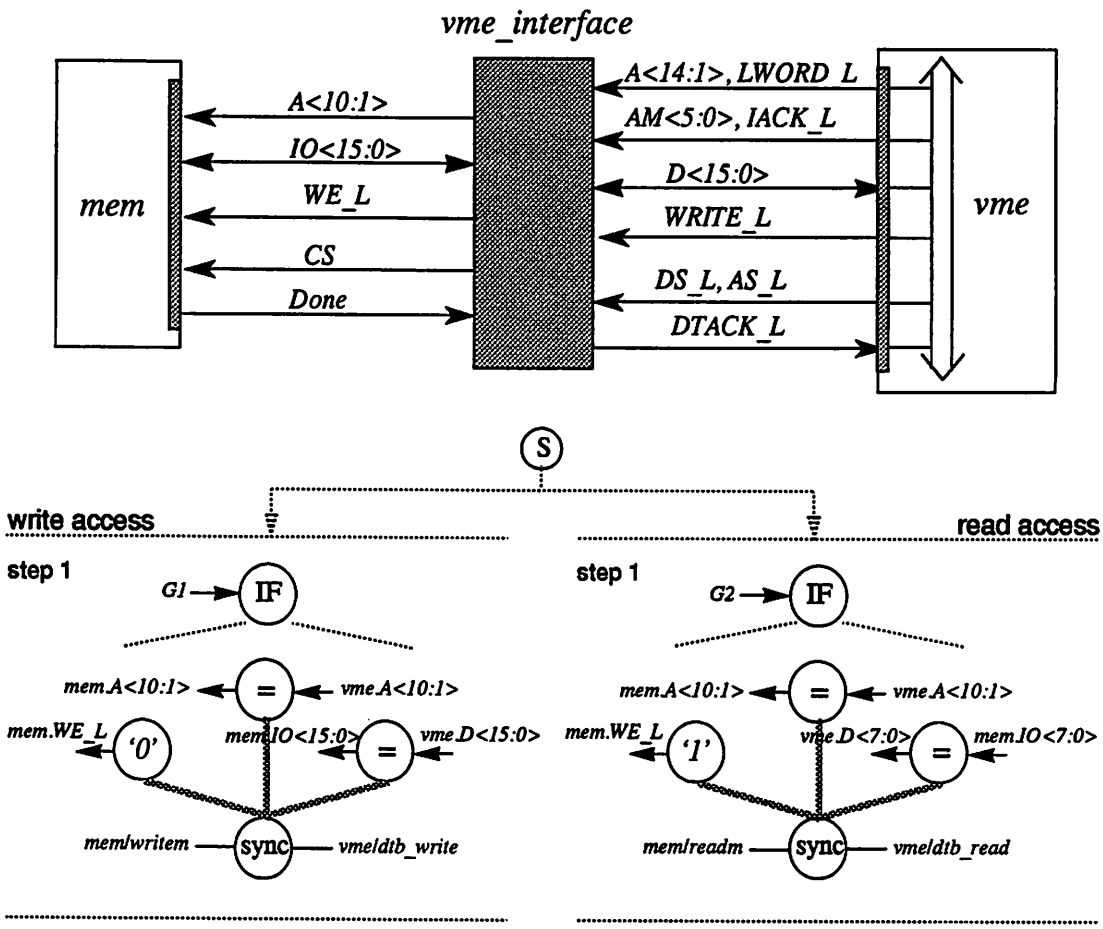


Figure 6-11 : Schedule for VMEbus Interface

TMS320 to Optical Link Interface

The scheduled flow graph representing multiplexed communication between the TMS320 uni-processor and the TAXI optical transmitter is shown in Figure 6-12. The IF control operation and its guard node is scheduled into the first control step, and the concurrent control operation is placed into a second control step. The schedule contributes to two control states in the interface controller implementation. As shown in the original flow graph of Figure 5-11, the input to the data word

transfer actually arrives during a prior transaction. So, the TMS XD input is scheduled into the first control step. It is also allocated a storage element that holds the input for the data transfer in the subsequent control step.

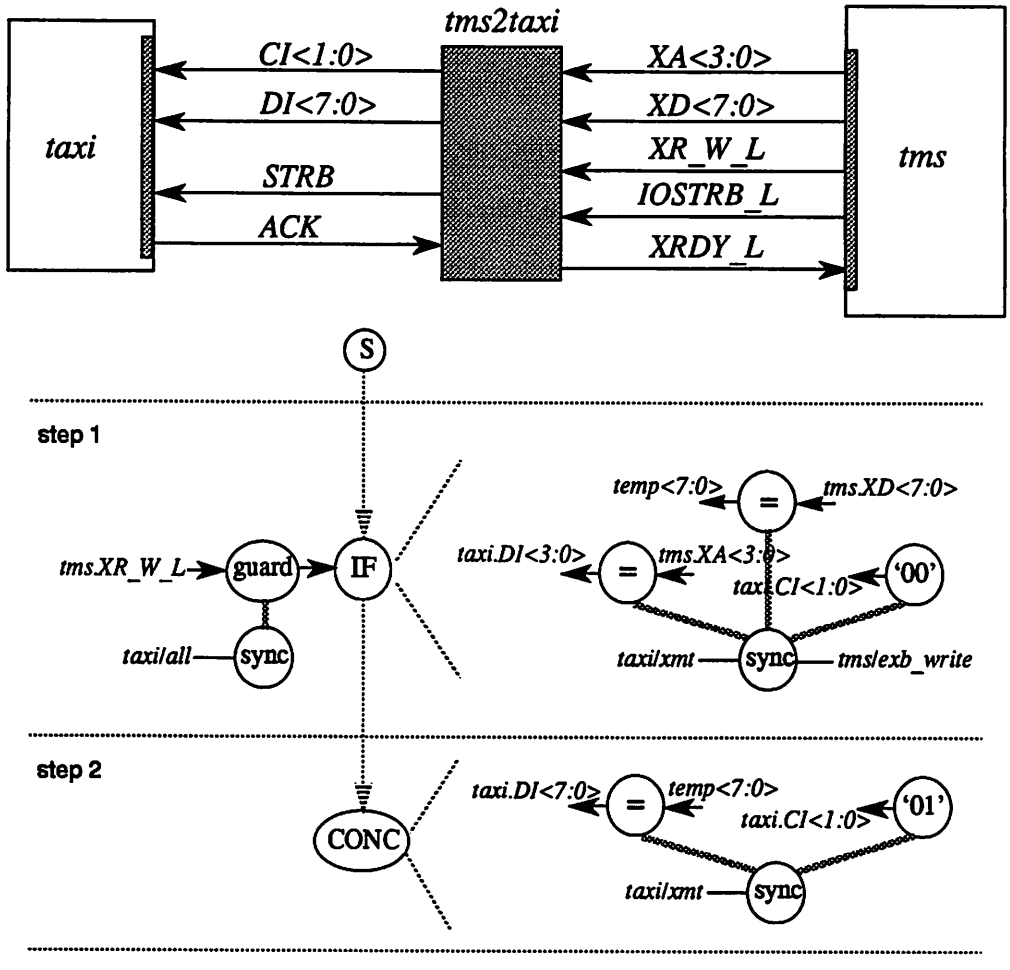


Figure 6-12 : Schedule for TMS320 to Optical Link Interface

Scheduling also transformed the sync node associated with the data word transfer. Originally, the sync node linked two I/O protocols, which were the TAXI and the TMS protocols. Because the TMS input was scheduled in a prior control step, the input to the data transfer in the new flow graph is from an internal flow graph edge. So, the data transfer in the scheduled flow graph is dependent only on one I/O protocol.

A/D Module to Optical Link Interface

Figure 6-13 illustrates the scheduled flow graph representing communication between an A/D module and the TAXI optical link. The example demonstrates how iterated behavior is transformed to a series of control steps. After loop unrolling, as shown in Figure 5-12, the concurrent control operations are scheduled into a linear flow of seven control steps. The last six steps correspond to iterated behavior, one step for each iteration. Accordingly, the interface controller FSM cycles through seven states. In the first state, the FSM configures the datapath for a constant transfer and sets up the protocol controller to execute the destination protocol. In the following states, the FSM configures the data path for a constant transfer and an inter-module transfer. It also sets the protocol controller up to convert between the source and destination protocols.

6.5 Summary

Of the three primary functions of an interface module discussed in Section 1.1, the synthesis techniques presented in this chapter specifically addresses the communication datapath and transfer control needs. Scheduling and allocation techniques transform an initial flow graph into one that is ready for mapping to RTL units in the target interface template. Scheduling orders inter-module transfers and data operations into control steps, assigning them a time to be executed by the final interface hardware. Allocation determines the appropriate type and number of hardware resources required to execute the transfer and data operations. The synthesis strategy first generates an initial schedule and allocation from the flow graph and optimizes the initial solution into a final flow graph. The first part uses specially developed techniques, while the second part uses available algorithms [McFarland90]. Together, the two techniques bring behavior closer to a structural realization.

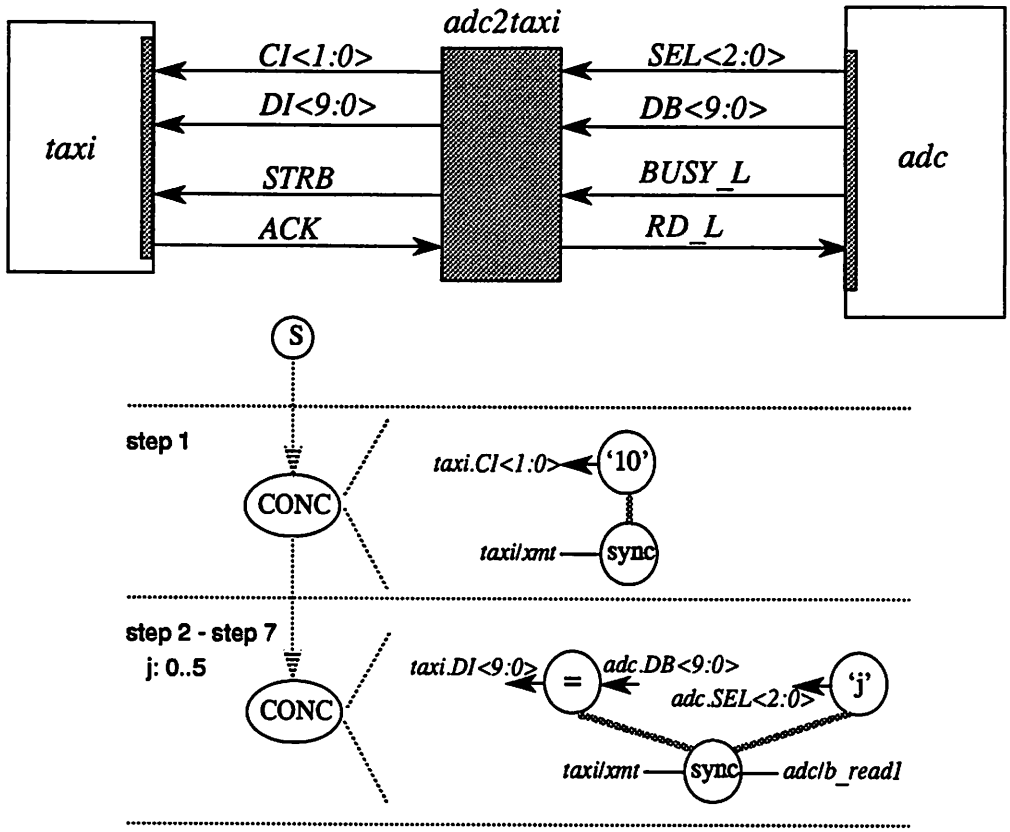


Figure 6-13 : Schedule for A/D Module to Optical Link Interface

CHAPTER 7

GENERATION OF EVENT GRAPHS

Comprehensive synthesis of interface modules includes a treatment of the functional requirements and a treatment of the design constraints. The previous three chapters have covered the inter-module communication behavior, which is the functionality, and synthesis techniques based on the flow graph representation. This chapter focuses on synchronizing communicating modules and synthesis techniques that account for I/O protocols and time constraints, which are the design constraints. Dealing with synchronization requires moving from the flow graph to the event graph level.

The next synthesis phase takes a flow graph of scheduled transfers and operations, and generates event graphs that represent protocol synchronization, as shown in Figure 7-1. The basic technique uses the data dependencies of an inter-module transfer to interlock event graphs corresponding to I/O protocols in the module library. The result is an overall event graph specifying the sequencing and time constraints that correctly synchronize the transfer. This phase links high-level behavior captured in the flow graph to the low-level behavior captured in the event graph. From both these behavioral specifications, the complete interface structure is synthesized.

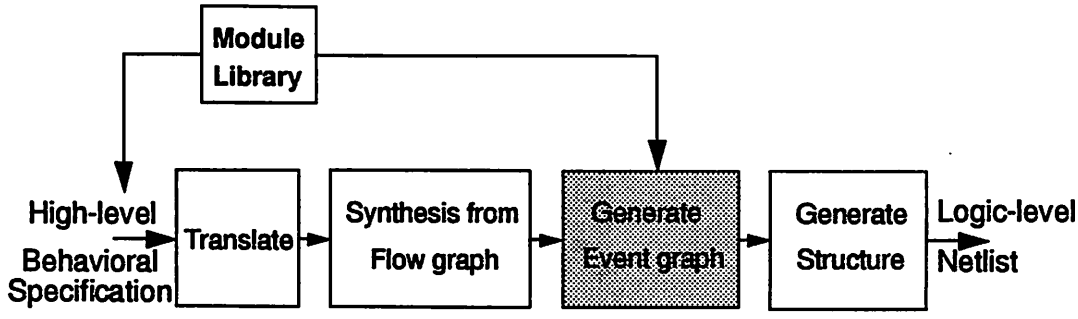


Figure 7-1 : Event Graph Generation and the Design Flow

The first two sections of this chapter present the techniques for generating and optimizing interlocked event graphs. The next section gives a summary, and the last section applies the techniques to four examples.

7.1 From Flow Graph to Event Graph

The flow graph captures inter-module communication behavior using data operation nodes, such as assignment, delay or sync nodes. The sync node is really a system-level abstraction for I/O synchronization between source and destination modules. To synchronize a transfer, protocol events are interlocked to meet the sequencing and time constraints of the individual modules. The protocols are named by the sync node and captured in the module library with event graphs.

For example, Figure 7-2 illustrates the processor to memory interface from Figure 5-2. The block diagram, the input specification and clustered flow graph are repeated here for convenience. The processor transfers address and data information to the memory on a write cycle. The sync node associated with the write transfer specifies that the processor and memory is synchronized by interlocking events from the processor protocol called "writep" and the memory protocol called "writem". Each protocol event graph in the module library has a unique name to identify it.

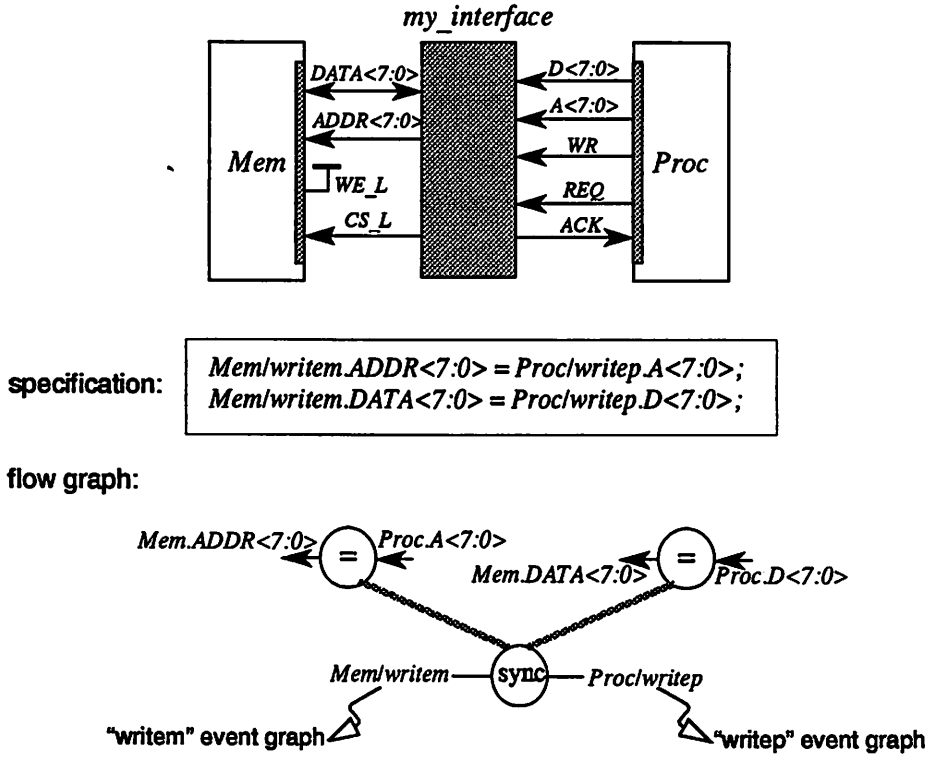


Figure 7-2 : Flow Graph for Processor to Memory Interface

So, the synchronization problem is how to interlock the individual graphs, generating an overall event graph that describes the synchronization procedure. The interlocking is determined by the data dependencies and the performance requirements of the data flow associated with a sync node. The basic technique for interlocking event graphs is presented below. It is applied to each sync node in the flow graph.

7.1.1 Generating the Initial Event Graph

To synchronize an inter-module transfer, the related protocol event graphs are interlocked into an initial and overall event graph using the data dependency requirements of the transfer. This is the first step in the event graph generation phase. The processor to memory write example illustrates this step, as shown in Figure 7-3.

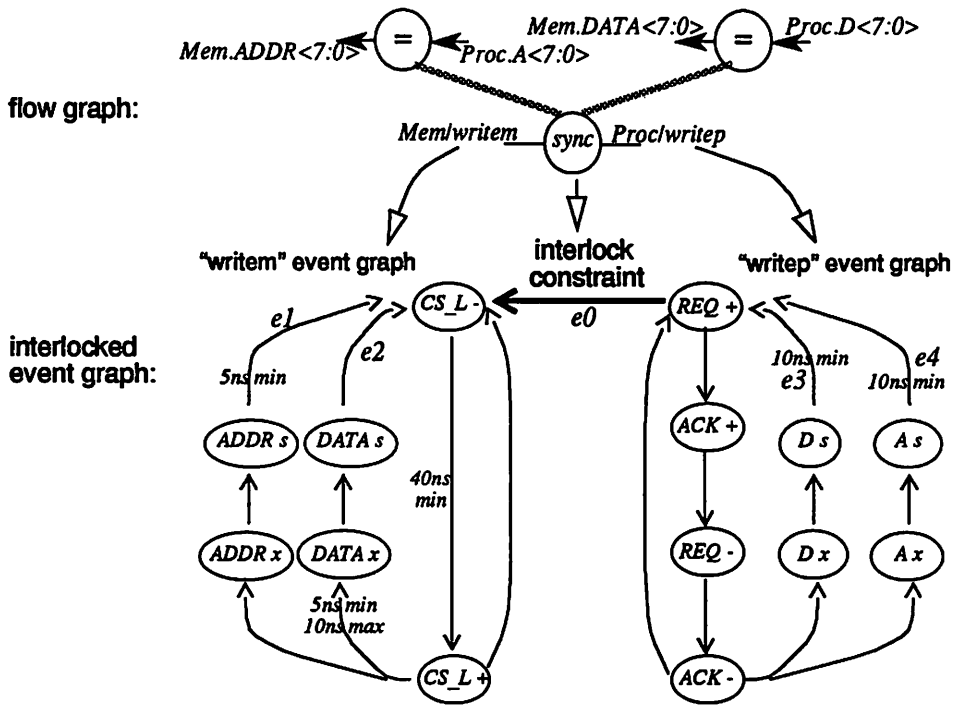


Figure 7-3 : Interlocked Event Graph for Processor to Memory Interface

For the sync node associated with the write transfer, synthesis starts by loading the named event graphs for the source and destination ports from the module library. In the example, this consists of the processor "writep" graph and the memory "writem" graph, shown on the right and left side of Figure 7-3, respectively. Interlocking the source and destination event graphs uses the data dependency principle: *an event that indicates that information is available from the source must precede the event that indicates that information has arrived at the destination.* In Figure 7-3, the *REQ+* event indicates that the processor is presenting valid address and data, indicated by edges *e3* and *e4*, while the *CS_L-* event signals that valid information is present at the memory port, indicated by edges *e1* and *e2*. To interlock the event graphs, an event precedence edge is thus inserted from the source event to the destination event. In the example, the edge *e0* forces the *REQ+* event to trigger the *CS_L-* event, interlocking the "writep" and "writem" graphs into an

overall event graph that governs the synchronization procedure. Such an edge is called an *interlock constraint.*

The key to the above synthesis step is the interlock constraint. In essence, the constraint provides a synchronization point for the communication. When there are multiple sources that concurrently communicate with a destination, protocol events from the sources are interlocked with a common destination event. Figure 7-4 illustrates this. The two processors transfer data to the memory during one write cycle. Synthesis loads a copy of the memory "writem" event graph from the module library, and also a copy of the "writep" event graph for each processor. The first processor provides the upper half of the data byte, so an interlock constraint is inserted from event *Proc1.REQ+* to event *Mem.CS_L-* (edge *e0*). The second processor provides the lower half of the data byte, generating the interlock constraint from *Proc2.REQ+* to *Mem.CS_L-* (edge *e1*). The interlocked event graph is formed from three individual event graphs. Likewise, when a source port concurrently communicates with multiple destinations, a common source event is interlocked with a protocol event at each destination. This is shown in Figure 7-5, where a single processor writes two memory devices during the same write cycle. In either case, the interlock constraints provide multiple synchronization points for the communication.

Comparing Figure 7-2 to Figure 7-3, the input specification does not mention any time constraints, such as the *40ns min* time constraint from *CS_L-* to *CS_L+*, or any control I/O signals used for protocol signaling, such as *CS_L* and *REQ*. Synthesis introduces these details into the behavioral representation during event graph generation. This phase and the module library support design from a system abstraction level. Interlock constraints are equivalent to the interconnect edges used in Janus [Borriello88b] to interconnect event graphs before synthesizing the control logic. Also, the constraints can be formally expressed using guarded commands [Martin86], which was applied in [Meng89] as an input specification for synthesis of handshake circuits as described in Chapter 2. For example, the interlock constraint in Figure 7-3 is expressed as:

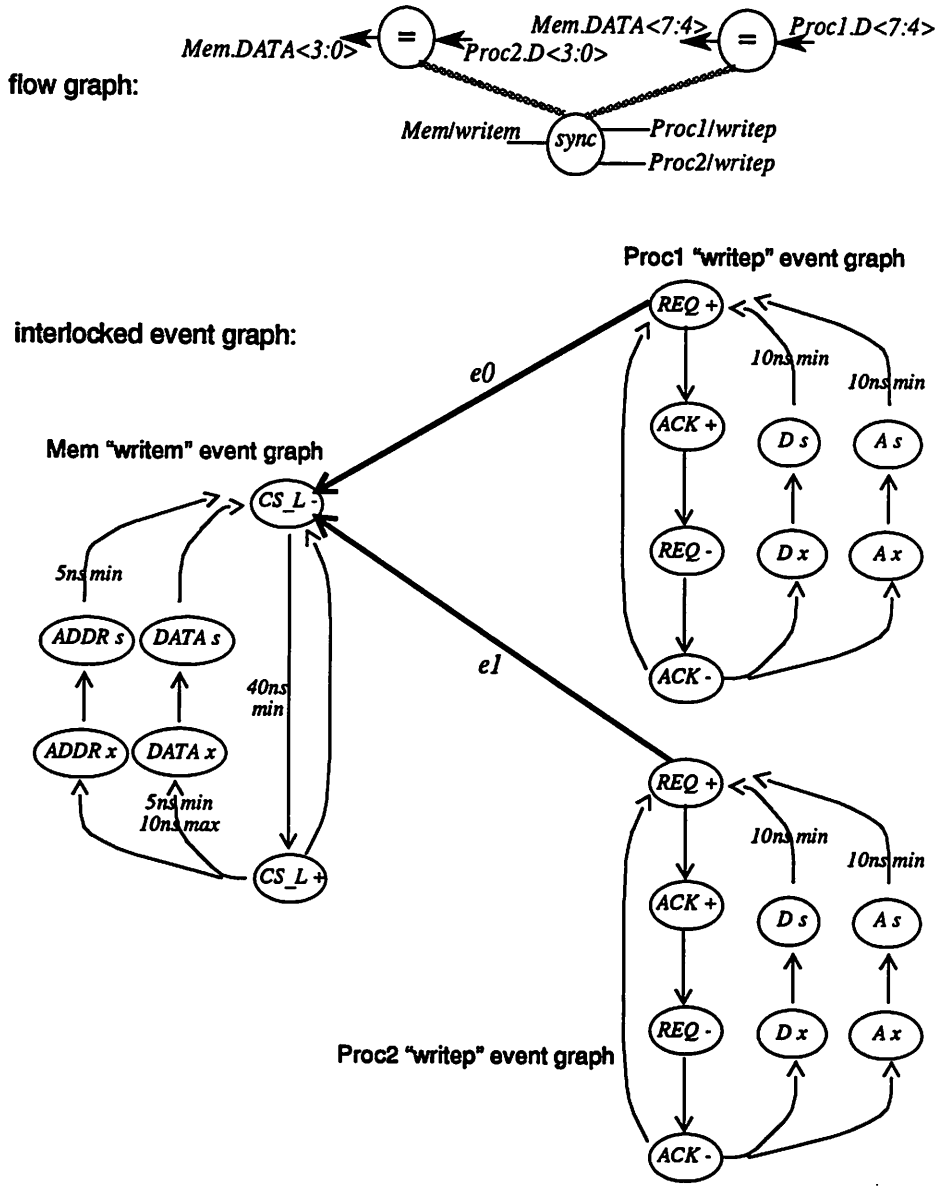


Figure 7-4 : Interlocked Event Graph for Two Sources and a Destination

$$[Proc.REQ+ \rightarrow Mem.CS\_L-]$$

The constraints for in Figure 7-4 is expressed as:

$$[Proc1.REQ+ \text{ AND } Proc2.REQ+ \rightarrow Mem.CS\_L-]$$

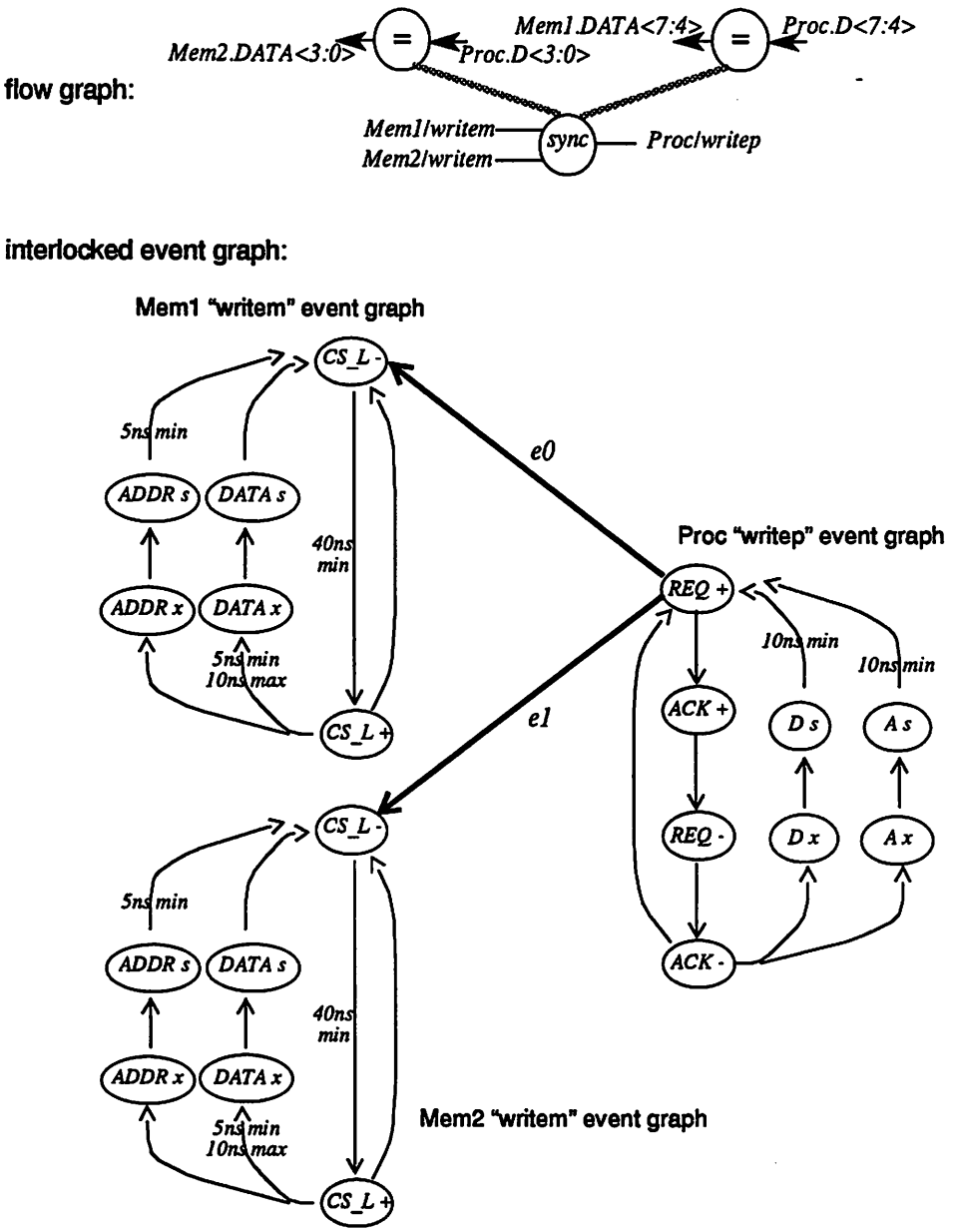


Figure 7-5 : Interlocked Event Graph for a Source and Two Destinations

And, the constraints in Figure 7-5 is expressed as:

$$[Proc.REQ+ \rightarrow \text{Mem1.CS\_L-};\ Proc.REQ+ \rightarrow Mem2.CS\_L\text{-}]$$

The relationship of interlock constraints to interconnect edges and guarded commands demonstrates that this synthesis phase is actually the front end to the synthesis of protocol control logic.

7.1.2 Synchronization with Data Operations

The previous step synchronizes the source and destination ports which are external to the interface. In addition, the transfer must be synchronized to the interface module itself. This is covered by the second step in generating an interlocked event graph. As described in Chapter 6, the interface template includes an interface controller that configures both the datapath and the protocol controller. This means that the interface controller synchronizes data operations to the associated protocol conversion, represented with the interlocked event graph generated in the above step. The datapath switches to a new configuration according to the Dsel word issued by the interface controller, while the protocol controller switches to a selected event graph according to the Esel word, as shown in Figure 6-4. From the viewpoint of the protocol controller, the interface controller is yet another source port, and it needs to be synchronized with the external ports.

The interface controller port is called "Ictrl". The Esel word is accompanied by a 4-phase handshake protocol implemented on the request and acknowledge control signals, R and A. The event graph for the Ictrl handshake is built into the module library, and the R+ event indicates that the Dsel and Esel word are valid, as shown Figure 7-6. The handshake graph is interlocked with the event graph generated from data dependencies of the external transfer. Since the Ictrl port is considered a source, an event precedence edge is inserted from the Ictrl.R+ event to the destination event. Figure 7-7 illustrates this additional interlock constraint for the processor to memory example. The [*Ictrl.R+* → *Mem.CS_L-*] interlock constraint provides a second synchronization point for the write transfer. The constraint is also a part of the complete event graph for the multiple source and destination examples in Figures 7-4 and 7-5, though not shown.

The Ictrl handshake plays an important role when the transfer is between external ports and the

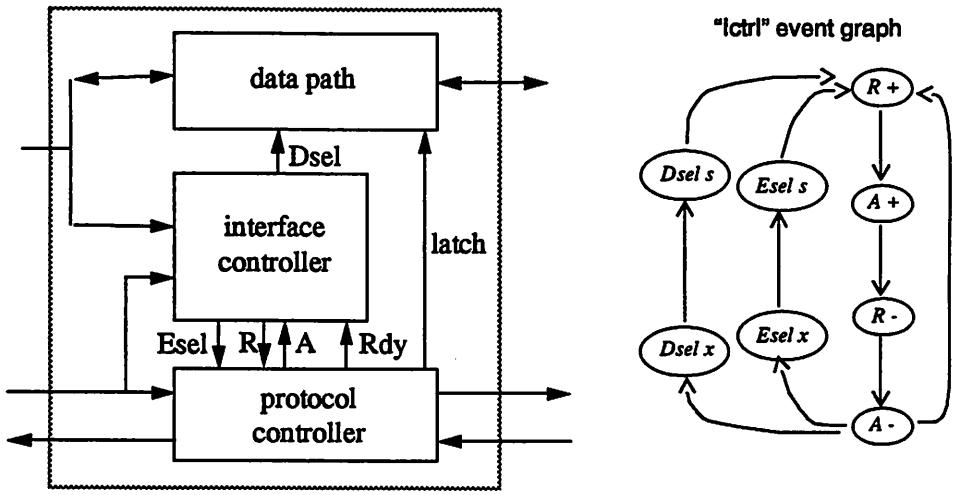


Figure 7-6 : Interface Controller and the Ictrl Handshake

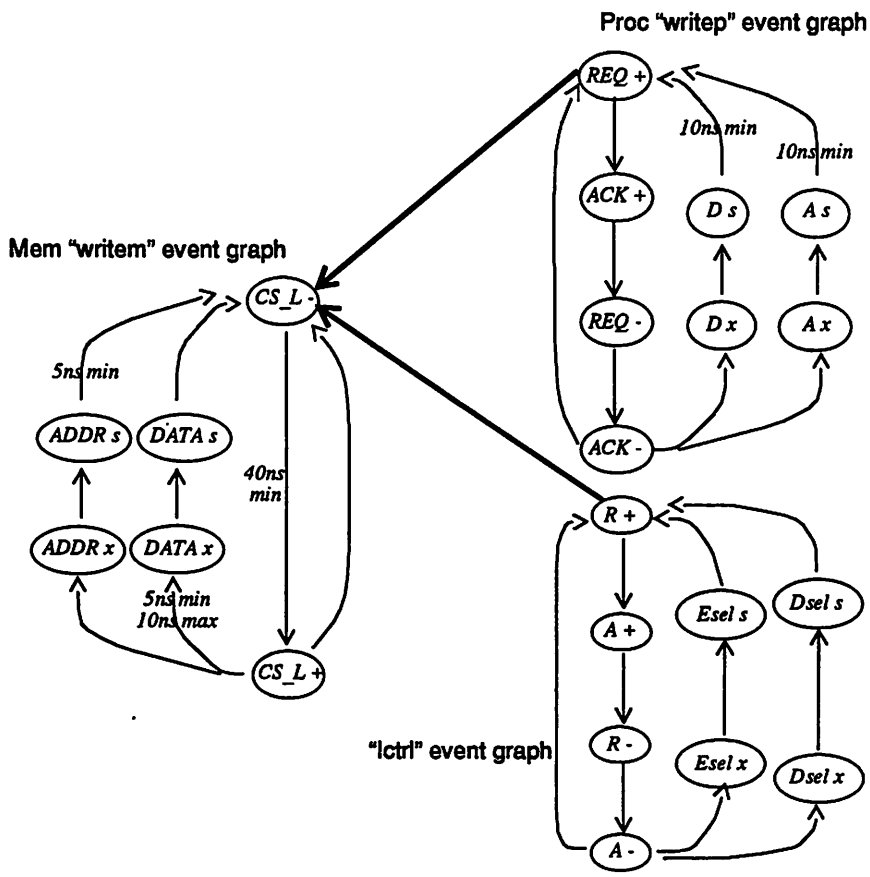


Figure 7-7 : Interlocked Event Graph with Ictrl Handshake

delay operation, which represents internal storage. In this case, the external ports consist of only sources or only destinations. If they are only sources, then the internal Ictrl port acts as a destination since the delay operation receives the input information. If they are only destinations, then the Ictrl port acts as a source since the delay operation provides information. Figure 7-8 shows an example of synchronizing the delay operation with an external source and also an external destination. In the first control step of the flow graph, the delay operation captures an input value in internal storage. So, the source protocol, "*send*," is synchronized with the Ictrl handshake using the interlock constraint [*STRB*+ → *Ictrl.A*+], as shown in the top event graph. In the second control step, the value captured in the delay node is exported to the destination. As shown in the bottom event graph, the Ictrl handshake is synchronized with the destination protocol, "*recv*," using the interlock constraint [*Ictrl.R*+ → *CS*+].

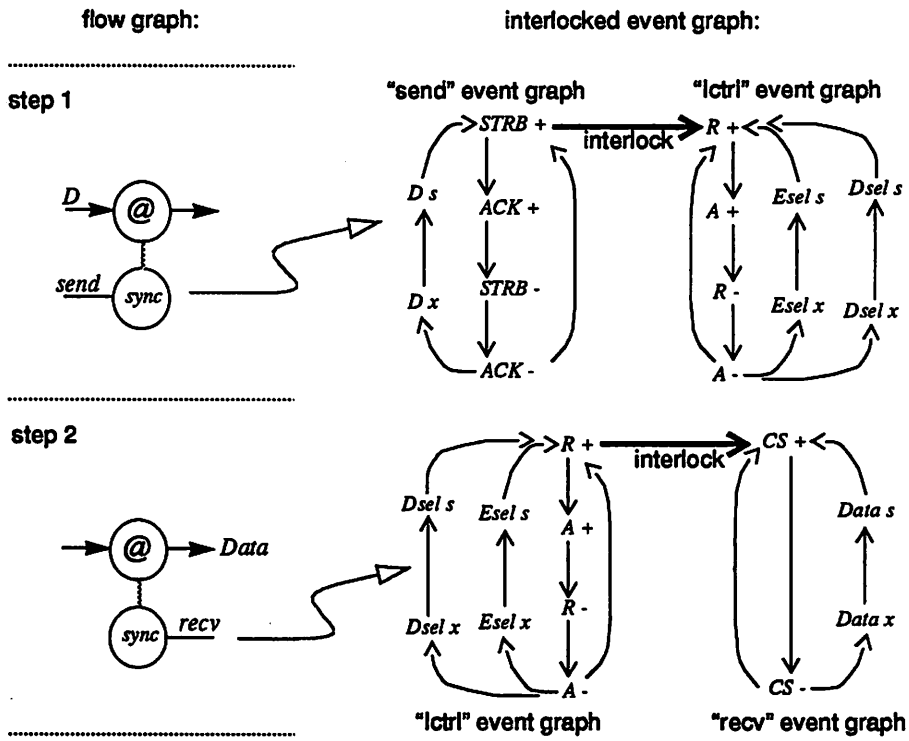


Figure 7-8 : Synchronizing Delay Operations with the Ictrl Handshake

At this point, synthesis has generated an initial interlocked event graph for an inter-module transfer that satisfies the data dependency requirements.

7.2 Optimizations

The interlock constraints introduced in previous steps are the minimum amount of constraints needed to correctly synchronize a transfer. The next synthesis phase inserts additional interlock constraints to optimize the final logic implementation of the event-level behavior.

7.2.1 Performance and Area Trade-offs

Buffers are inserted into an inter-module transfer for two possible reasons. In either case, the use of a buffer trades area for performance, and it affects how the source and destination event graphs are interlocked.

First, inter-module transfers are often buffered to increase the system communication performance. For example, a buffer (also called FIFO, queue or pipeline register) temporarily stores M words of width N (bits) that are being transferred from a fast source port to a slow destination port. Without the buffer, the source port must hold the data until the destination acknowledges that it has received the data. With a buffer, the source device can release the data, as soon as the buffer has latched it, and proceed to some computation task. In the meantime, the destination device can read in the data at its own rate. However, for the gain in performance, the communication implementation requires more area than an unbuffered implementation. In the buffered case, the MxN buffer itself requires additional datapath area. Also, to take care of full and empty buffer status, the controller requires extra logic and hence area. Using a buffer to increase communication performance is a system level decision made by the designer.

A second reason to buffer an inter-module transfer is to satisfy local time constraints. Here, the source port provides valid data for a specific amount of time regardless of how long it takes the

destination to capture the data. During the optimization phase, this situation is detected and synthesis *allocates* a buffer (or any form of storage) to properly synchronize the transfer. To illustrate this step, Figure7-9 shows a slight variation of the processor to memory write transfer. The inter-module data flow is the same as before, but the processor uses a protocol implemented with only a request type signal, and the memory protocol uses a request and acknowledge type signal. As shown by the interlocked event graph, the source protocol does not have an acknowledge. The destination receives corrupt data, because it acknowledges after the source has already released the data. To maintain the integrity of the transferred data, a buffer is inserted to the datapath to latch the data before the source releases it. Of course, the memory has to complete the write cycle before the next processor word is latched. In contrast to the first reason, buffering makes for correct transfers rather than for maximum performance. The overhead of buffering is the buffer itself and perhaps some extra protocol control logic.

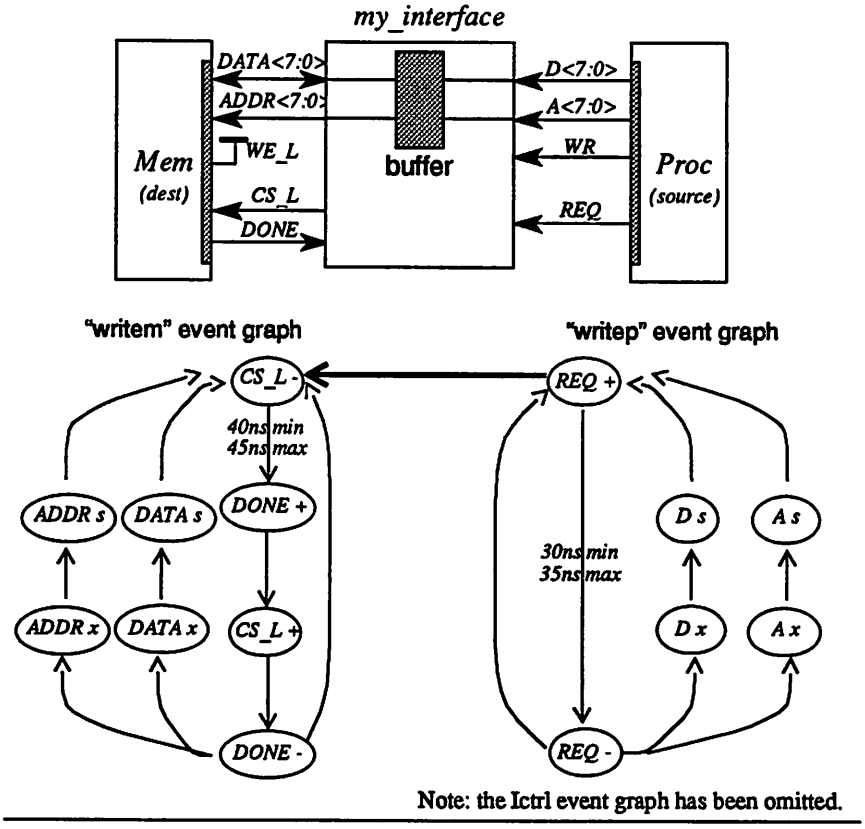


Figure 7-9 : Transfer Synchronization Using a Buffer

When the inter-module transfer is buffered, the interlocked event graph generated from data dependency requirements is sufficient to properly synchronize the transfer. When the transfer is unbuffered, the source can release its data only after the destination has acknowledged capturing the data. Additional interlock constraints are needed to ensure this condition, and are constructed as follows.

Starting from the initial interlocked event graph, synthesis finds a protocol event indicating that the destination has captured the data and a protocol event indicating that the source is allowed to disassert its data. These are just acknowledge events occurring on control signals using names like "ACK" or "DONE." The destination acknowledge must precede the source acknowledge, and this forms an *additional* interlock constraint. For example, assume now that the processor and memory protocols use an acknowledge, as shown in Figure 7-10. The source and destination event graphs are interconnected with two precedence edges to properly synchronize the transfer. The first is the original interlock constraint that fulfills data dependency requirements. The second is the acknowledge interlock constraint [$DONE+ \rightarrow ACK+$] that eliminates the need for a buffer, and hence minimizes the implementation area.

7.2.2 Merging Event Graphs

So far, the interlocked event graphs have been generated and optimized for each sync node in the flow graph. The complexity of the protocol controller is determined by the number of different interlocked event graphs to be synthesized. The next optimization merges interlocked event graphs from different sync nodes to reduce the complexity of the final implementation. In other words, some sync nodes can use the same interlocked event graph, even though the data dependence requirements are different. The results is a smaller amount of area to realize the protocol controller. This optimization may compromise the communication performance.

Event graphs can be merged only if they involve the same set of ports. Figure 7-11 illustrates the optimization. Suppose that the sync node in the first control step maps to the interlocked event

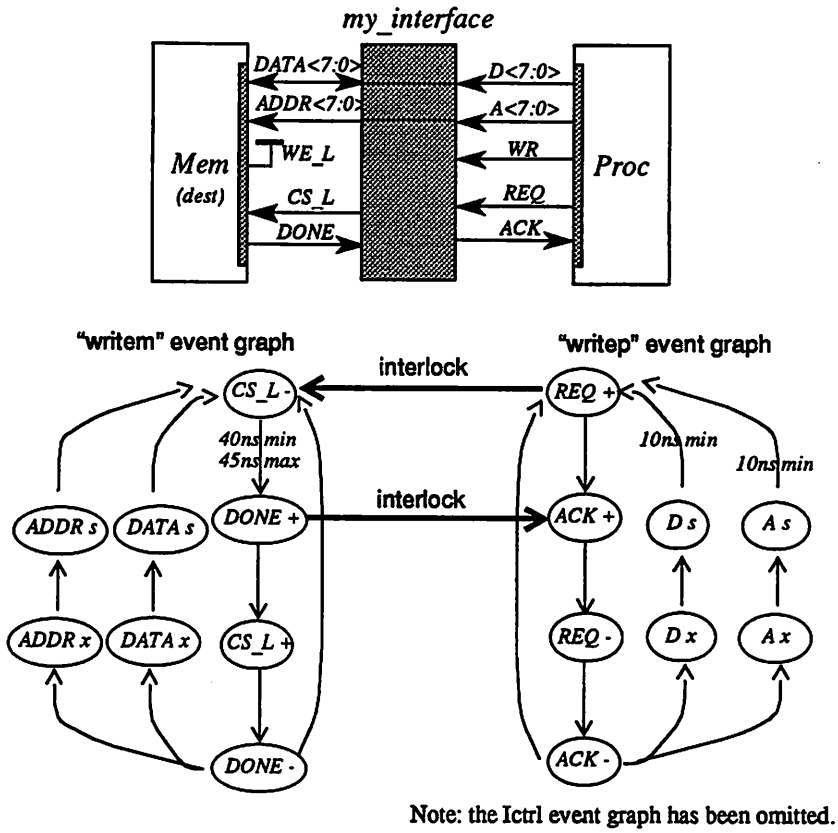


Figure 7-10 : Interlocked Event Graph for Unbuffered Transfer

graph shown. There is only one synchronization point formed from the associated data dependency. The sync node in the second control step interlocks the same individual event graphs with two synchronization points, including an acknowledgment. Looking at both event graphs, the second meets the interlock constraints of the first. So, the first sync node can use the same event graph as the second one. The acknowledge constraint forces the source to wait for the destination which increasing the write cycle time, but the original interlock constraint is still met. The final solution maps both sync nodes to the interlock constraints [$Req1+ \rightarrow Req2+$; $Ack2+ \rightarrow Ack1+$]. Only one event graph is implemented instead of two and leads to savings in area.

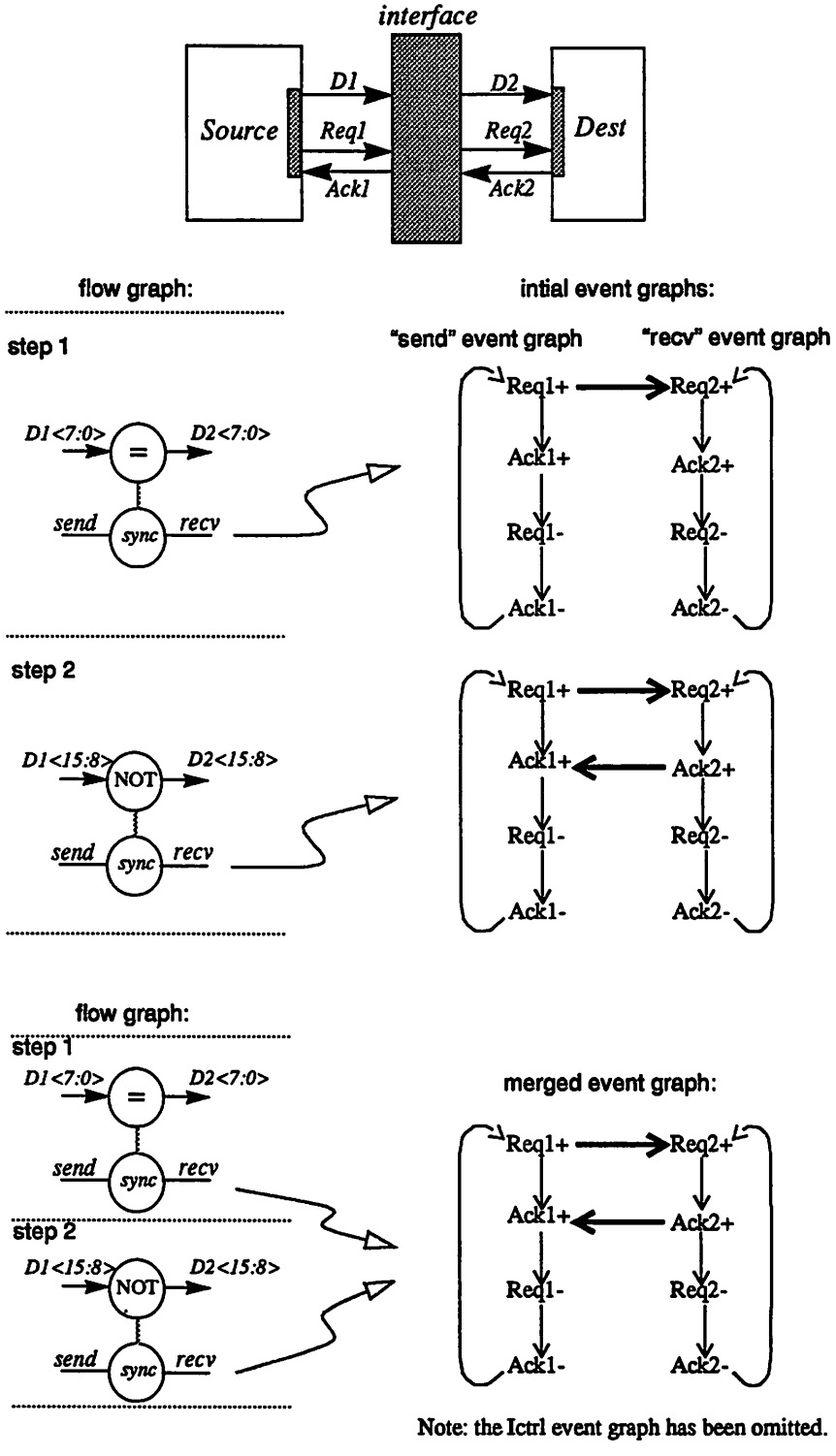


Figure 7-11 : Merged Event Graphs

7.3 Algorithm Summary

The first two sections have described how synthesis brings the design specification of the interface from the flow graph level to the event graph level. Most importantly, these sections demonstrate how behavioral synthesis and the module library introduce protocol events and time constraints into the design specification, hiding the low-level details from the designer. From the data dependency requirements of a transfer, and the support of the module library, protocol event graphs are interlocked into an overall event graph that describes the event-level synchronization, including time constraints. The initial event graph is a minimum behavior specification that can then be optimized to reduce the implementation area or increase communication performance. The optimizations further build the interlocked event graph by adding interlock constraints.

At the conclusion of this step, the design specification consists of the flow graph, describing the high-level data and control flow behavior, and a set of event graphs, describing the event-level synchronization and timing behavior. Each sync node in the flow graph has a corresponding event graph, and some sync nodes map to the same event graph. The flow graph is ready for transformation into a register-transfer level datapath and interface controller. The interlocked event graphs serve as input to existing tools that can synthesize the protocol controller logic. In the interface template, the Esel word selects an event graph depending on the state of the interface controller.

The next section presents examples of interlocked event graphs generated from the techniques discussed. The next chapter discusses the transformation from behavior in the flow graph and event graphs to the register-transfer structure implementation based on the interface template.

7.4 Examples of Interlocked Event Graphs

The following examples use the four interface examples from Section 6.4 of Chapter 6 to explain how the interlocked event graphs are generated from the flow graph and module library.

Only the sync nodes within the scheduled flow graphs are shown in this section. The complete scheduled flow graphs from Chapter 6 are not repeated here, but they should be looked at while reading the following explanations.

Optical Link to D/A Module Interface

The scheduled flow graph for this interface which performs demultiplexed transfers from the TAXI optical receiver module to a bank of D/A converters was shown in Figure 6-10. It contains two sync nodes. The first represents synchronization between the source TAXI port and the interface module itself when a data word is latched into temporary storage. Since the destination is the interface, the TAXI protocol is interlocked with the Ictrl handshake from the module library, as shown in Figure 7-12. The *DSTRB+* event indicates that the TAXI has a valid word on the *Do* bus, and it triggers the Ictrl *A+* event to synchronize the transfer, as expressed by the interlock constraint [*taxid.DSTRB+* → *Ictrl.A+*]. The TAXI protocol has no acknowledge event, and,normally, this synthesis phase would allocate storage to capture the TAXI data word before it is disasserted. However, the storage allocated during the scheduling phase already serves this purpose, making another storage element redundant.

The second sync node shows that the TAXI port is synchronized with the D/A destination port as a new TAXI word and the stored word are transferred to the destination. The Ictrl *R+* event signals that the storage element has a valid word, and the *DSTRB+* event indicates that the TAXI module is presenting a new data word. As shown in Figure 7-12, both these events are interlocked with the D/A *LDAC_L-* event, which signals valid address and data on the *BankSel* and *D* busses at the destination. The synthesized interlock constraint is [*Ictrl.R+* AND *taxid.DSTRB+* → *dac.LDAC_L-*]. Since the TAXI protocol has no acknowledgment, a storage element is allocated to the transfer from the TAXI to the D/A.

In both control steps, source words are placed in temporary storage. This is implemented with a register or latch which uses a clock-type event to sample the input information. The clock event, or

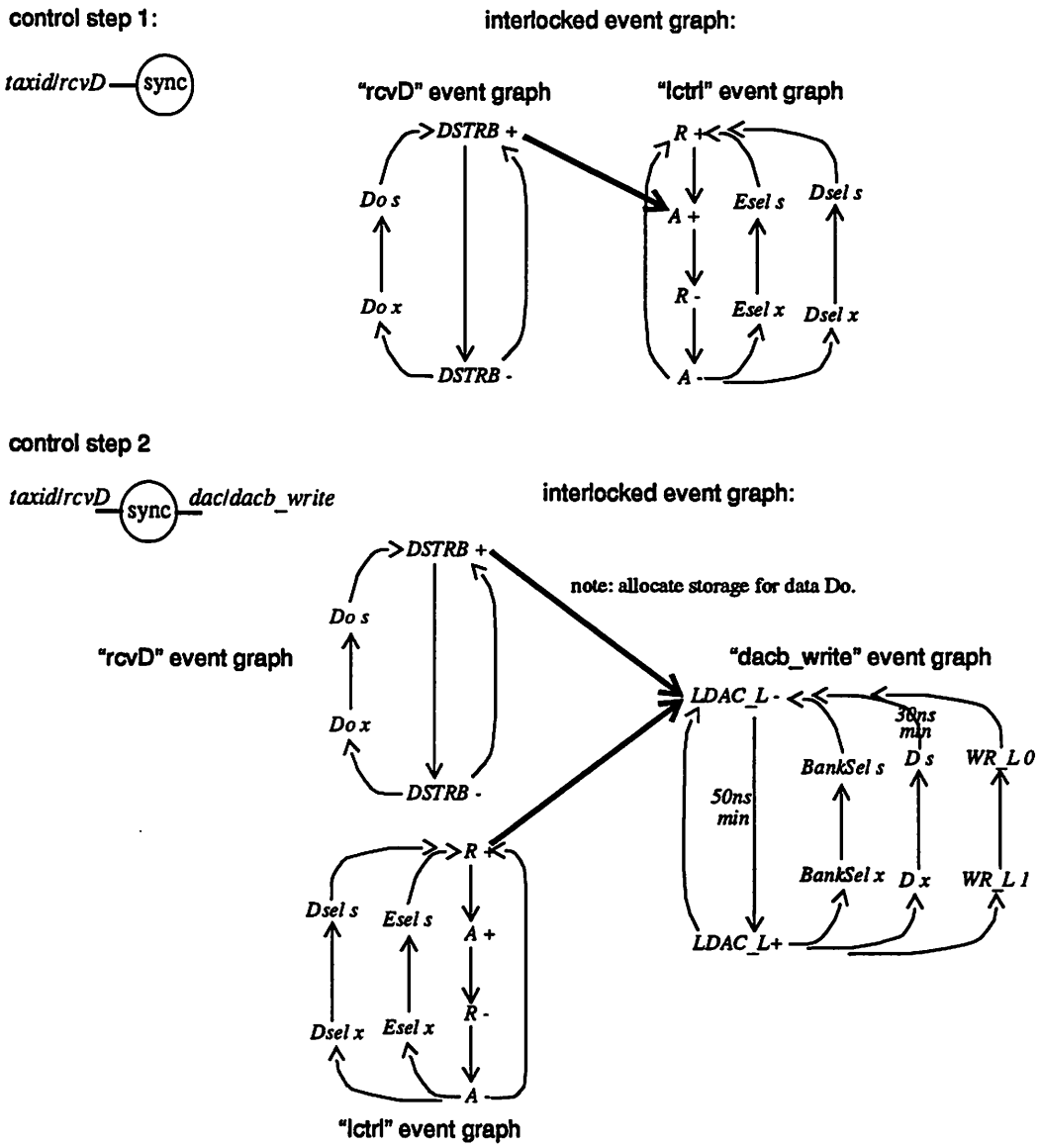


Figure 7-12 : Interlocked Event Graphs for Optical Link to D/A Interface

latch event, is derived from events in the interlocked event graph. Generating the latching events is explained in the next Chapter.

VME System Bus Interface

Figure 6-11 illustrated the scheduled flow graph for read and write accesses between the VME system bus and a static RAM module.

Figure 7-13 shows the interlocked event graphs for the write transfer. During the write cycle, the VMEbus presents new address (indicated by *AS_L-*), new data (indicated by *DS_L-*), and a new read/write status (indicated by *DS_L-*). The memory receives all these three words upon the *CS-* event. The sync node associated to the write transfer interlocks the VME "*dtb_write*" protocol and the memory "*write*m" protocol, loaded from the module library. Due to data dependencies, synthesis generates the interlock constraint [*Ictrl.R+* AND *vme.DS_L-* → *mem.CS-*]. The *DS_L-* event occurs after the *AS_L-* event, as shown in the VME event graph, so it is used to indicate that both the address and data are valid. During the optimization step, synthesis recognizes that both the memory and the VMEbus have acknowledgments and generates the interlock constraint [*mem.CS+* → *vme.DTACK_L-*], providing another synchronization point for the write transfers. Because the acknowledgment forces the VMEbus to wait for the memory access time, the write transfer is unbuffered.

During the read access, the VMEbus is the source of address and read/write status, and the memory is the destination of these words. As shown in Figure 7-14, this generates the interlock constraint [*Ictrl.R+* AND *vme.DS_L-* → *mem.CS-*]. In the same transfer, the memory is the source of the selected data word and the VMEbus is the destination. This generates the interlock constraint [*mem.Done-* → *vme.DTACK_L-*], since the *Done-* event signals that the memory has placed the selected word on the data bus, and since the *DTACK_L-* event indicates to the VME master that the accesses word is available on the bus. It is important to observe here that the acknowledge synchronization is generated from the data dependency of the read access, rather than from optimizations.

In fact for the read access, the previous interlock constraint also serves as the acknowledgment for

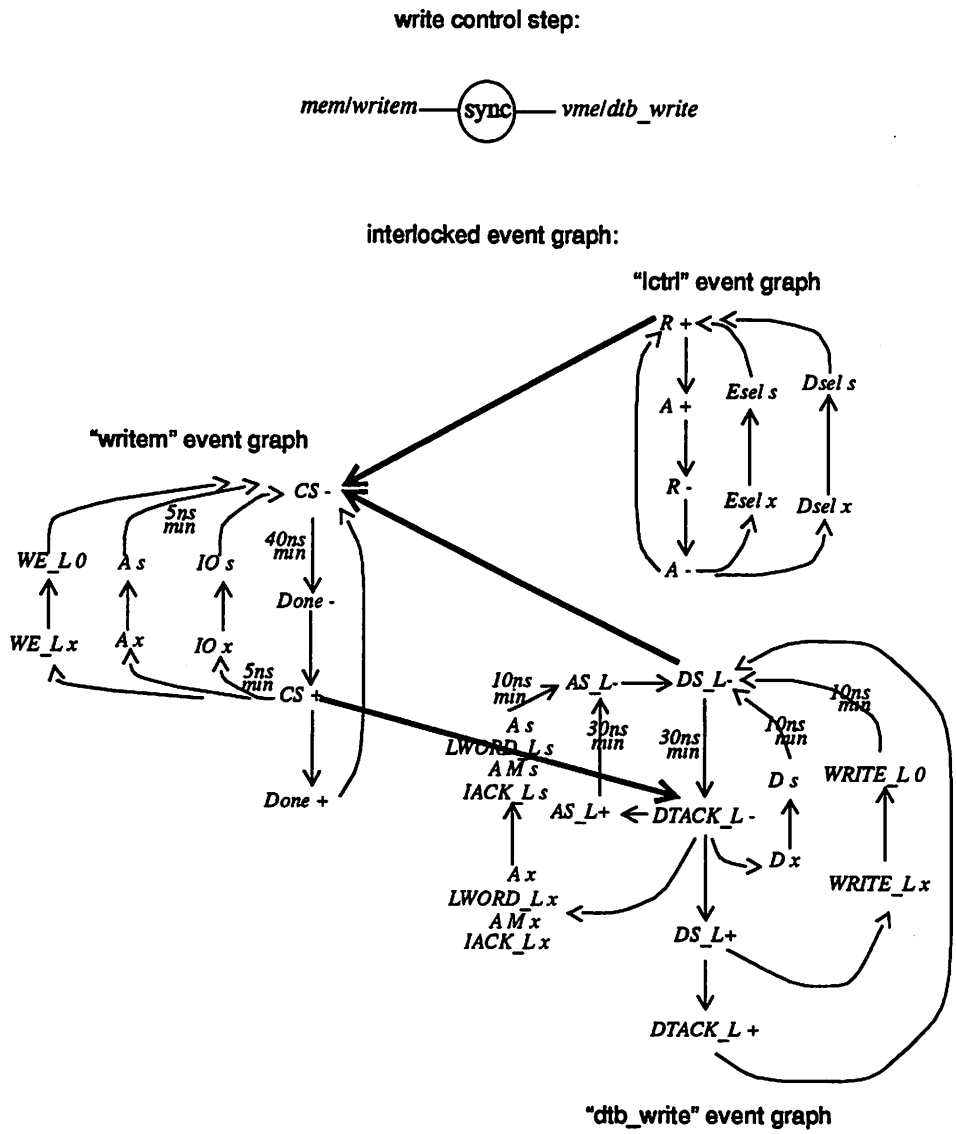


Figure 7-13 : Interlocked Event Graph for VMEbus Write Access

the read/write status transfer, and this is recognized during the optimization step. The interlock constraint [*vme.DS_L+* → *mem.CS+*] acknowledges the data transfer, and the constraint is generated by synthesis during the optimization step. Optimizing for the address transfer is not straight-forward compared to the other two. In this case, the address acknowledge constraint

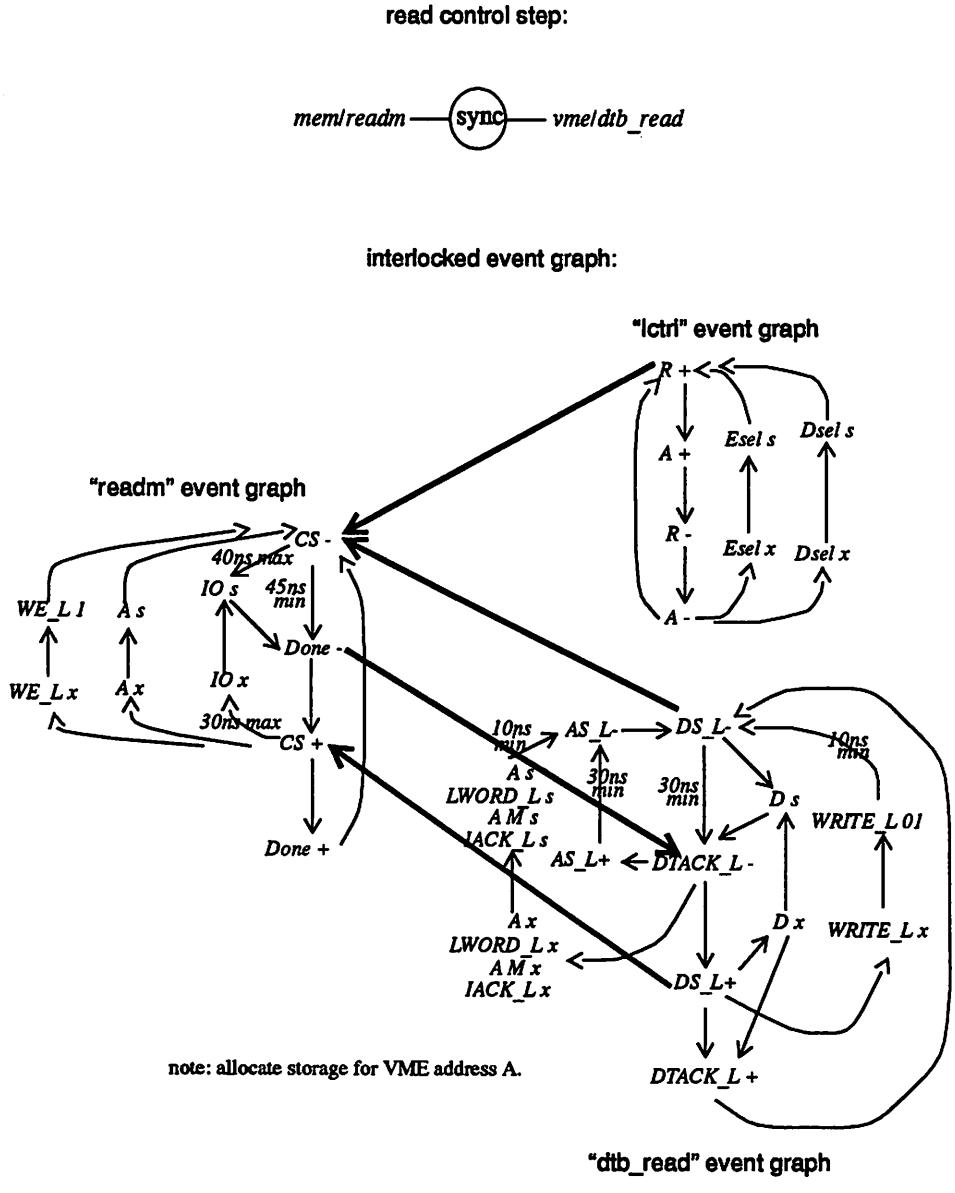


Figure 7-14 : Interlocked Event Graph for VMEbus Read Access

would conflict with the data acknowledge constraint. To resolve this, synthesis only chooses one of the constraints, which is the data transfer, and then allocates a storage element to the other, since an acknowledgment can not be used to hold the transfer.

TMS320 to Optical Link Interface

This interface multiplexes address and data from the TMS320 uni-processor to the TAXI optical transmitter, which can only send one of the words at a time. The scheduled flow graph describing the behavior is shown in Figure 6-12. In the first control step, the TMS and the TAXI are synchronized as the address is transferred and as the data is latched into temporary storage; the synchronization is represented by the sync node. As shown in Figure 7-15, the TMS address and data words are both valid when the *XRDY_L-* event occurs, and the taxi accepts a new word when its *STRB* signal is asserted. Accordingly, the event graphs are interlocked with the constraint [*tms.XRDY_L-* AND *Ictrl.R+* → *taxi.STRB+*]. Although the interface is the destination of the stored data word, the [*Ictrl.R+* → *taxi.STRB+*] constraint works equally well as [*tms.XRDY_L-* → *Ictrl.A+*], and the latter is not needed.

Both the TMS and TAXI protocols have an acknowledgment, and this potentially allows area optimizations such as making the TMS data and address transfer unbuffered. However, the data must be stored for scheduling reasons, and the acknowledgment is not exploited. Acknowledging the address transfer with the precedence [*taxi.ACK+* → *tms.XRDY_L-*] would conflict with the [*tms.XRDY_L-* → *taxi.STRB+*] constraint generated from data dependency requirements. For correct synchronization, the address acknowledgment constraint is not used, and optimization allocates an address latch to control step 1.

In the second control step, the only transfer is from the data storage element and constant in the interface datapath to the TAXI port. The associated sync node shows that the Ictrl handshake is synchronized to the TAXI "*xmt*" protocol, loaded from the module library. As illustrated in Figure 7-15, data dependency requirements generate the interlock constraint [*Ictrl.R+* → *taxi.STRB+*]. No optimizations are made.

This example also demonstrates the subtleties of storage allocation. In the first control step, the data storage element was allocated during scheduling of the flow graph operations. It holds data

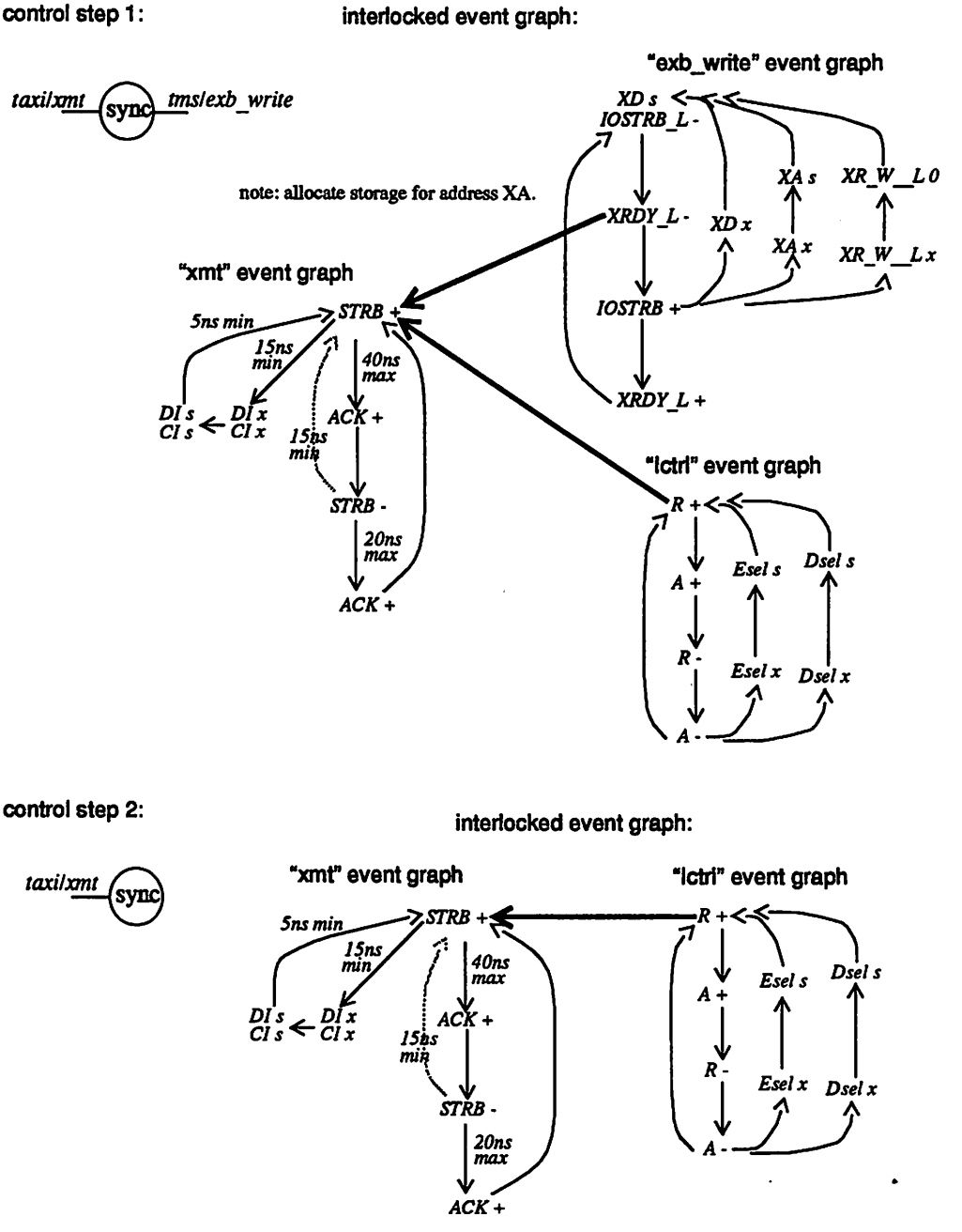


Figure 7-15 : Interlocked Event Graphs for TMS320 to Optical Link Interface

across multiple control steps. The address storage element is allocated during the event graph generation step, because the source or destination time constraints are incompatible. It only holds

data during one control step to maintain the integrity of a transfer. Section 7.2.1 explained that a buffer, also a storage element, can be specified by the system designer to improve system communication performance. Altogether, storage elements are used from the system level design abstraction to the data flow and the low-level event abstractions.

A/D Module to Optical Link Interface

Figure 7-16 illustrates the interlocked event graphs generated from the scheduled flow graph in Figure 6-13. The flow graph describes the communication behavior between an A/D conversion module and the TAXI optical transmitter. In the first step, a constant generated by the interface is sent to the taxi module. As shown by the associated sync node, the Ictrl handshake is interlocked with the TAXI "*xmt*" handshake to synchronize this transfer. Since the interface is the source and the TAXI port is the destination, the interlock constraint generated using data dependency is [*Ictrl.Req+* → *taxi.STRB+*]. Because the interface can hold the constant value for as long as needed, the optimization step determines that the constant is neither latched nor is the acknowledge constraint necessary.

For the iterated transfers, the A/D "*b_read1*" and TAXI "*xmt*" protocols are interlocked, as illustrated in Figure 7-16. The interface is the source of the constant address to the A/D module, which accepts an address on the *RD_L-* event. The A/D module becomes the source of the requested data to the TAXI destination. The A/D places valid data on the *DB* bus as indicated by the *RD_L+* event, and the TAXI accepts the data on its *DI* bus when the *STRB+* occurs. From these data dependencies represented in the flow graph and the information in the event graphs, synthesis generates the interlock constraints [*Ictrl.R+* → *adc.RD_L-*; *adc.RD_L+* → *taxi.STRB+*]. Although the A/D and TAXI module both have an acknowledge signal, a data acknowledge constraint would conflict with the address transfer constraint, [*adc.RD_L+* → *taxi.STRB+*], and is therefore not used. To insure transfer integrity, a storage element is allocated to the data word transfer, as explained in Section 7.2.1.

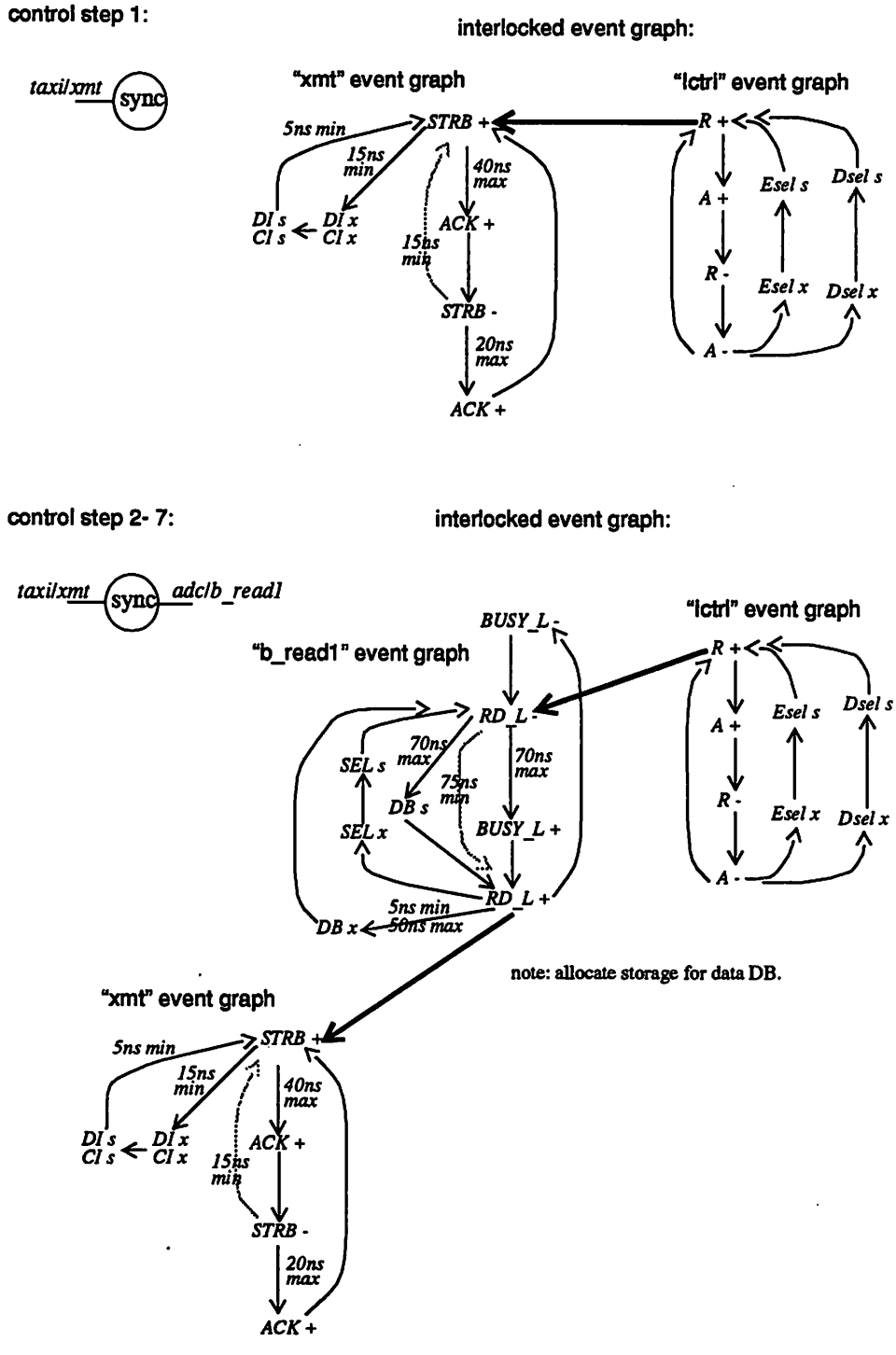


Figure 7-16 : Interlocked Event Graphs for A/D Module to Optical Link Interface

CHAPTER 8

FROM BEHAVIOR TO STRUCTURE

The ultimate goal of behavioral synthesis is to find a register-transfer structure that implements the specified behavior, represented with a flow graph and a set of event graphs. ALOHA transforms a behavior specification into a structure based on the interface template of Figure 6-4, consisting of a datapath and two sequential controllers. As shown in Figure 8-1, this is the final phase of high-level interface design.

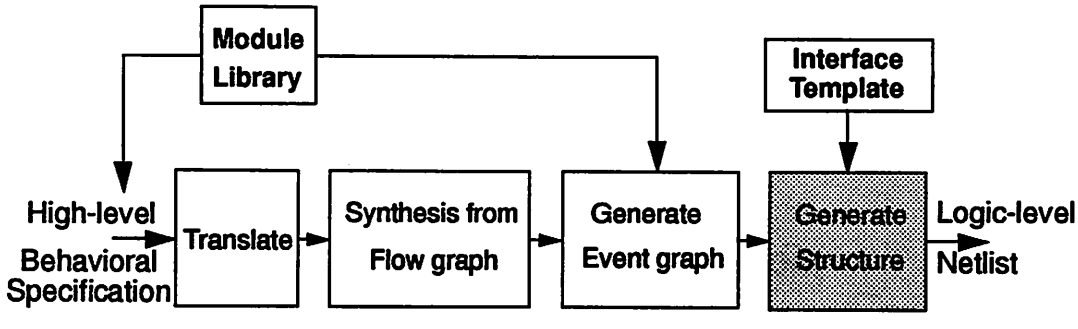


Figure 8-1 : RTL Structure Generation and the Design Flow

The flow graph captures the data and control flow in inter-module communications. Its allocated

data flow elements are mapped to datapath units, and its control flow schedule is mapped to a finite state machine specification of the interface controller. The event graphs capture the I/O synchronization requirements of a transfer, and the protocol control logic is synthesized from the representation. The final output from synthesis is a netlist of register-transfer units in the SDL format [Richards], with some units having a combinational logic description in the BDS [Segal88] or EQN [Sentovich88] format, and also time constraints expressed in the CLOVER format [Doukas91].

This chapter describes the techniques and issues in mapping from a flow graph and event graph to register-transfer level (RTL) hardware. The first section discusses datapath compiling. The second covers synthesis of protocol control logic from event graphs, and the third discusses generation of the interface controller logic.

8.1 Datapath

The datapath implements data flow behavior. It contains the register-transfer units that route information between source and destination ports and performs computations on the information. This section discusses datapath generation as shown in Figure 8-2. To construct the datapath, data flow nodes and data edges in the flow graph are mapped to busses, multiplexors, functional units, storage elements, and tri-state buffers. The construction is completely independent of the control flow. Using the Dsel signal in the interface template, the datapath is made configurable on each control step.

8.1.1 Compiling the RTL Network

The datapath is constructed in three steps. The multiplexors and output wires and busses are generated first. Then the storage and functional units are created, consisting of registers or latches, adders and combinational logic units. The last step places tri-state buffers on the appropriate

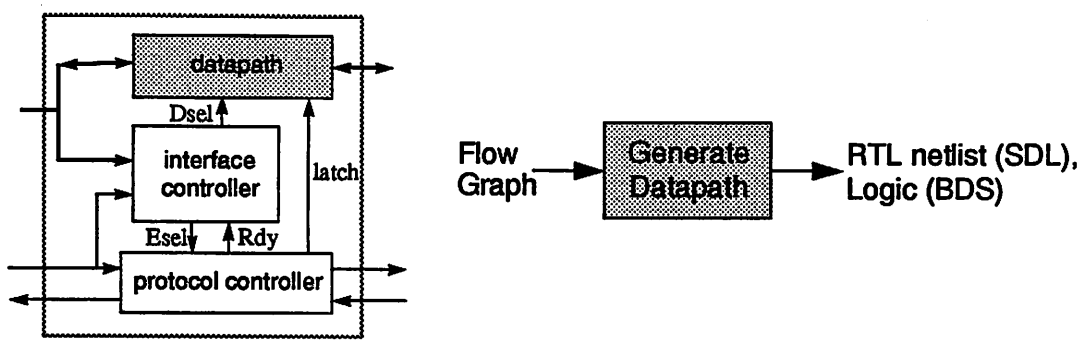


Figure 8-2 : Datapath Generation and Output Formats from Synthesis

output signals in the network. At each step, the newly created units are interconnected with the previously created units using newly or previously created busses. In this section, the term "bus" refers to both a single wire and a bus, which is a collection of wires. The term "register" and "latch" is also used interchangeably.

Multiplexors and Output Busses

The generation of output signals and multiplexors uses a method similar to the one in [Hill78] for microprocessor RTL programs. Data edges in the flow graph which are also outputs translate into an output bus in the in the datapath. The width of the bus is identical to the width of the signal represented by the data edge.

The need for a multiplexor is implied by the data flow among control steps rather than directly mapped from a data flow node or data edge like the other datapath elements. Multiple control steps in the flow graph may contain data flow nodes that place a value on the same output signal. To buffer the multiple drivers from the output, the corresponding output bus is assigned a multiplexor (or mux). An input bus to the mux is created for each data flow node that places a value on the output. The input is selected using an unique Dsel wire, which synthesis assigns and connects to the mux. If all inputs are deselected, then the mux outputs a "0" logic level. To capture input/Dsel

relationship in the flow graph, the corresponding data flow node is annotated with the Dsel bit. Figure 8-11 illustrates the Dsel annotation for the TMS320 interface, and the interface controller synthesis uses this information. In the next synthesis step, datapath units for the data flow node are created, and their fan-outs are connected to the appropriate mux input.

To illustrate output bus and multiplexor creation, the VME interface described in Figure 6-11 is used. The output data edges are the VME data D signal and the memory A, WE_L and IO signals. So, an output bus is generated for each. The only output that has more than one source is the memory WE_L signal. In the write access it is driven by a constant "0" node, and in the read access it is driven by the constant "1." So, a multiplexor is used to drive the WE_L signal, as shown in Figure 8-3. Constants correspond to a hardwired logic level, and one input to the mux is the "1" level while the other input is the "0" level. A Dsel bit is assigned to each constant input to select between the two. The composite of the two Dsel bits forms the complete select word Dsel<1:0>. This mux is formed regardless of the VME interface schedule. If the write access were followed by a scheduled read access, the datapath structure will still be the same. Although not explicitly in the flow graph, the address transfer is buffered with a storage element allocated during the event graph generation, as shown in Figure 7-14. Such allocated storage is mapped to a latch, as shown in Figure 8-3.

The multiplexor network created so far may be optimized. For example, in the above datapath, the constant "0" will be transmitted to the mux output as long as Dsel<1> and Dsel<0> are disasserted. So, the "0" input can be eliminated, resulting in a minimized multiplexor as shown in the figure. In the final datapath netlist generated by ALOHA, a multiplexor is actually represented using a combinational logic unit with a BDS description of the specific multiplexor logic function.

Functional and Storage Units

The data flow nodes, listed in Table 5-1, map to functional and storage units in the datapath. This synthesis step takes care of all the node types from assignment to addition, and to the function

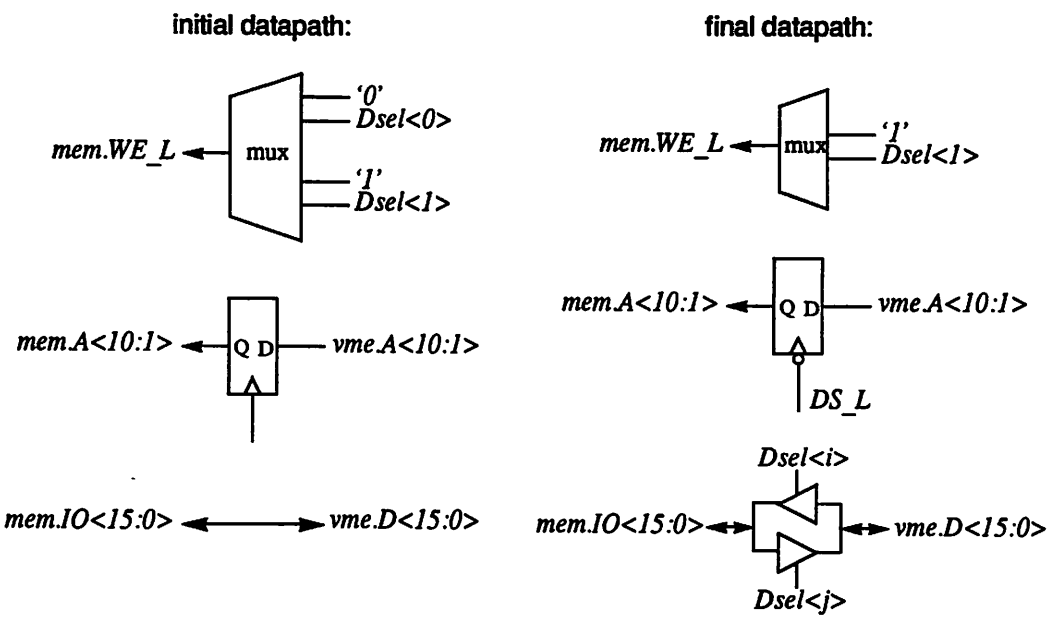


Figure 8-3 : Generating a Register-Transfer Level Datapath

type. Assignment nodes translate to a bus, as expected. Mentioned above, a constant node is implemented with a hardwired bit vector, where the vector is the parameter of the constant node. Delay nodes represent allocated storage, and a register is created in the datapath. Logical NOT, AND, OR, NAND, NOR and XOR nodes map to a logic gate of the same function, while the add node maps to an adder. Like the multiplexor case, synthesis creates a generic combinational logic unit for FUNCTION nodes, and the unit's parameter is the name of the "bds" file named by the FUNCTION node. Fan-ins to the generated datapath units connect to other functional and storage unit outputs, or input busses. Fan-outs of generated units drive other functional units, storage units, multiplexor inputs or output busses.

The datapath subblock for the *BankSel<5:0>* output signal in the optical link to D/A interface illustrates this synthesis step. In the flow graph of Figure 6-10, the *BankSel* signal is produced by capturing the Do input with a delay node and decoding the Do word with the function node in the

first control step, and then exporting the decoded result using the assignment node in the second control step. As shown in Figure 8-4, the delay node maps to a storage element, and the function node maps to a combinational unit with the parameter "decode.bds". The assignment node maps to a bus that connects the fan-out of the combinational unit to the *BankSel* output signal. At this point in synthesis, a clock wire is also created for the storage elements. The clock is derived from protocol events occurring in the protocol controller; this is explained in a later section.

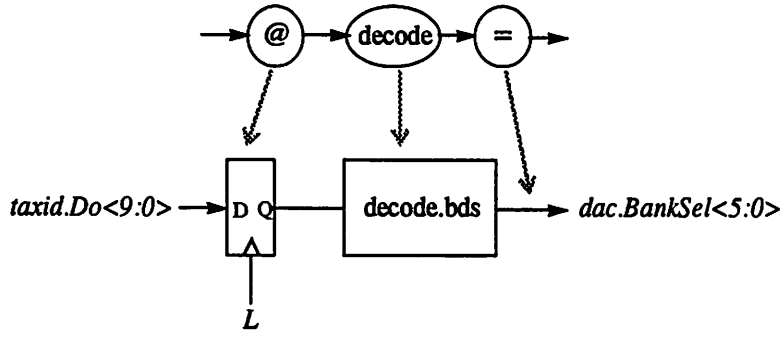


Figure 8-4 : Generating Functional and Storage Units

Tri-state Buffers

Bidirectional signals in the flow graph are assigned a bidirectional bus in the datapath and a tri-state buffer. As shown in Figure 8-5, one side of the bidirectional bus connects to the datapath boundary, and it imports signals to the datapath on the X_in fork while exporting signal to outside ports on the X_out fork. A Dsel bit enables the tri-state buffer during a control step where the X_out fork is used. For example, the bidirectional signals in the VME interface of Figure 6-11 are the VME data, D, and the memory data, IO. From the previous steps, the data assignment operations are mapped to two busses, illustrated in Figure 8-3. This step creates the tri-state buffers and connects the two bidirectional signals.

At this point, synthesis has created a netlist of register-transfer datapath units. The network can be put through combinational logic optimizations that cluster the various combinational units in the

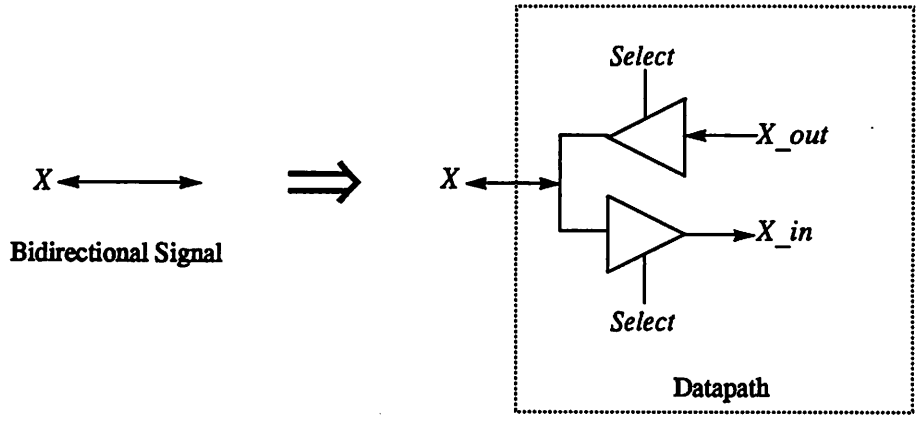


Figure 8-5 : Bidirectional Signal Implementation

datapath and minimize the logic and partition the logic onto specific hardware gates. Because of the top-down design methodology used in ALOHA, these optimizations are made at the low-level design phase which includes logic synthesis. This is discussed in the last section of this chapter.

8.1.2 Deriving Latching Events

The datapath stores information with edge-triggered registers. Since system interfaces usually operate in an asynchronous or mixed asynchronous/synchronous environment, the clock signal that latches data into the register is derived from protocol events rather than from from a periodic clock signal. The latching events are generated by the protocol controller, as shown in the interface template of Figure 6-4.

The key to deriving latching events is recognizing that information can be captured in a time interval when the source is providing valid data and before the destination port must receive it. This makes sense intuitively. So, the earliest time the information can be latched into temporary storage is upon the occurrence of a source protocol event that indicates it has valid data. The latest time the information can be latched is upon the occurrence of a protocol event that indicates the

information has arrived at the destination. These facts translate to the principle: *suitable latching events are those which lie on the interlock constraint edge*, described in Section 7.1.1 of Chapter 7. Only the initial interlock constraint corresponding to sending information from the source to destination is considered. The acknowledge interlock constraint is not considered. The selection principle applies even when the source or destination is an internal datapath register allocated during scheduling. In both these cases, The Ictrl handshake serves as the protocol that accompanies the internal register port.

For example, the event graphs for the VME interface are illustrated in Figure 7-13. The event selected to latch the address can be either *DS_L-* or *CS-*, both of which are used to interlock the VME and memory write event graphs. If *DS_L-* is chosen, then the latch delay must be less than the circuit delay from *DS_L-* to *CS-*. Otherwise, the integrity of the address transfer is destroyed. If the *CS-* event is used, then the memory should receive a delayed version of the *CS* signal, where the delay is greater than the latch delay. This also ensures the address transfer integrity.

The actual latching signal is formed by conditioning the selected latching event on the assigned Esel word, which enables the event graph associated to the current inter-module transfer. Using the VME example of Figure 7-13 to illustrate this, suppose the interlocked event graph for the write cycle is enabled with the Esel<0> bit, and the graph for the read cycle is enabled with the Esel<bit>. The write cycle address is latched with the signal formed from the logic AND of the Esel<0> bit and the DS_L complement, (Esel<0> AND !DS_L). Similarly, the read cycle address is latched with the signal, Esel<1> AND !DS_L. The composite latching signal for the address latch is (Esel<0> AND !DS_L) OR (Esel<1> AND !DS_L), which reduces to !DS_L after logic minimization. The datapath in Figure 8-3 shows the latch signal. If there were no read cycle, then the latching signal would just be (Esel<0> AND !DS_L).

The assignment of the Esel word is covered in Section 8.2.2. The issue of how to choose latching events from the event graph or STG is also treated in [Borriello88b] and [Meng88], respectively.

8.1.3 Examples

The four standard examples used in the previous chapters serve as full design examples to illustrate various steps in the design process. This chapter presents the final RTL specifications for each example. In this section, the datapaths shown are synthesized from the flow graphs in Section 6.4.

Optical Link to D/A Module Interface

The scheduled and allocated flow graph for demultiplexed transfers from the TAXI optical link to D/A module was shown in Figure 6-10. Constructing this interface's datapath was mostly explained in the Section 8.1.1. It is illustrated in Figure 8-6. The path from the TAXI *Do* input to the D/A *BankSel* signal is constructed from the data flow shown in the first and second control steps. The path from the input *Do* input the D/A output *D* accounts for the right assignment node in the second control step. Initially, there was a separate latch at the beginning of both paths. One was to take care of the delay node scheduled in the first control step, and the other is the latch (for *Do*) allocated in the event graph generation phase, which is shown in Figure 7-12. Datapath optimizations merged the two latches into the one shown in the datapath figure. Using the event graphs in Figure 7-12, the Do word can be latched with the TAXI *DSTRB* rising edge. The *DSTRB* control signal is routed from the protocol controller.

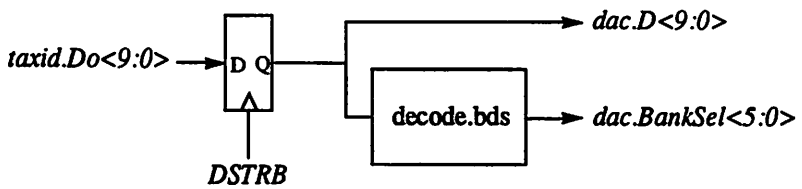


Figure 8-6 : Datapath for the Optical Link to D/A Interface

VME System Bus Interface

The complete datapath for the VME system bus interface is shown in Figure 8-7. Section 8.1.1 explained how the multiplexor, address latch, and tri-state buffers for the bidirectional data bus were generated from the flow graph in Figure 6-11. In the complete datapath, the three Dsel lines form a Dsel word, Dsel<2:0> issued by the interface controller. Using the event graph in Figure 7-14, the address latching event is the falling edge of the VME *DS_L* control signal, routed from the protocol controller.

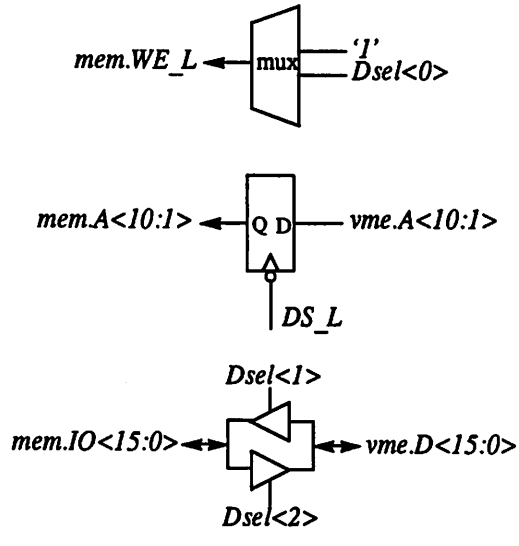


Figure 8-7 : Datapath for VMEbus Interface

TMS320 to Optical Link Interface

The flow graph representing multiplexed communication between the TMS320 uni-processor and the TAXI optical transmitter was shown in Figure 6-12. The datapath synthesized from the data flow nodes and edges is illustrated in Figure 8-8. The TMS *XA* latch corresponds to an address latch allocated during generation of the event graphs, shown in Figure 7-15. In the first interlocked event graph, the falling edge of the TMS *XRDY_L* signal is suitable for latching the TMS address and data. It is also possible to latch the address with the TMS *IOSTRB_L* falling edge, while the

data is still latched with the *XRDY_L* falling edge. The multiplexor transfers the TMS address *XA* or data *XD* to the TAXI *DI* bus depending on which Dsel bit is asserted. It is generated because the TAXI *DI* signal receives address from the TMS *XA* signal during the first control step and receives data from allocated storage element during the second step.

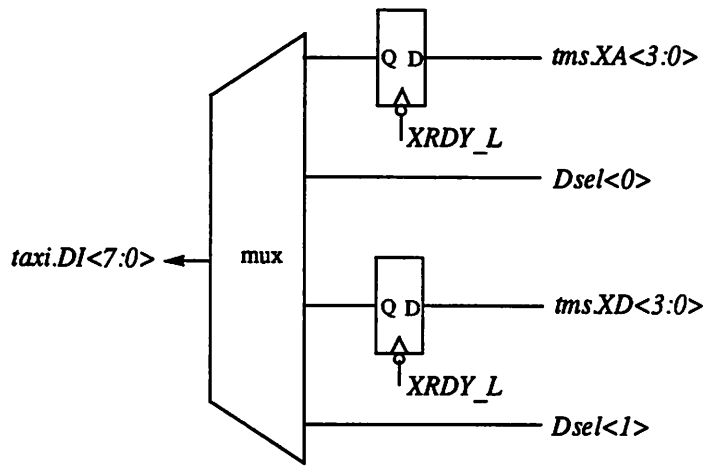


Figure 8-8 : Datapath for TMS320 to Optical Link Interface

The interface controller asserts the Dsel<0> select line during the first control step, followed by the Dsel<1> line during the second. This multiplexes the TMS address first and then the data to the TAXI destination. This example demonstrates how the datapath provides the link from sources to destinations, but it is the interface controller that determines when certain paths are enabled according to the schedule..

A/D Module to Optical Link Interface

The datapath for block transfers between an A/D module and the TAXI optical link is shown in Figure 8-9. It is generated from the flow graph in Figure 6-13. During the first control step, the interface sends a header '10' vector to the TAXI CI bus. This bus remains disasserted during the next control steps. Accordingly, in the datapath, the first multiplexor passes the "10" bit vector to the CI destination when the Dsel<0> bit is asserted. It should be kept in mind that the multiplexor

is just a functional requirement. In the final implementation, the multiplexor function may be implemented with a AND gate instead of an actual mux gate. This will be determined during the final step of logic synthesis.

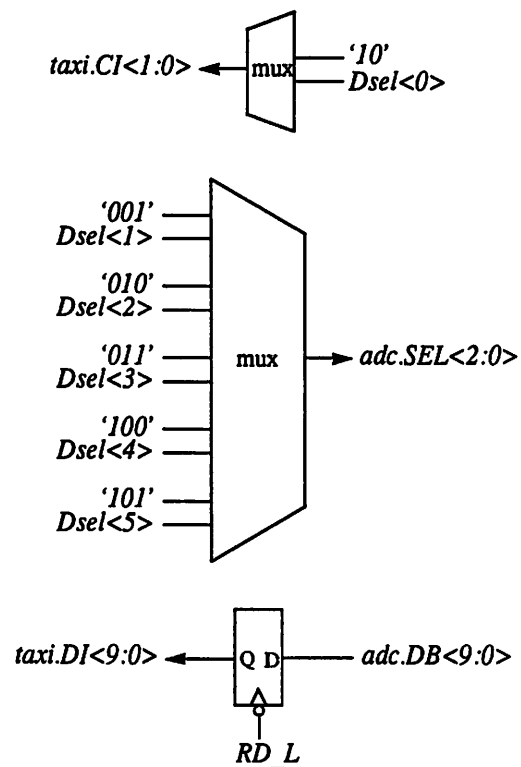


Figure 8-9 : Datapath for A/D Module to Optical Link Interface

During the next six control steps, the interface consecutively sends out a constant address. This behavior is mapped to the second multiplexor. Each constant is selected with an unique Dsel bit. Originally, the multiplexor had a constant '000' accompanied by a Dsel bit. Datapath optimizations eliminated these two terminals, because a zero vector is passed when all the Dsel bits have been disasserted. The complete Dsel word from the interface controller is Dsel<5:0>.

As shown in Figure 6-13, the address selects one of six converters in the A/D module to output a data word on the *DB* bus, which is transferred to the TAXI *DI* bus. The data transfer is represented with the assignment node in control steps 2 through 7. Each assignment node has identical input

and output signals, so they map to one data link as shown in the datapath figure. As shown in Figure 7-16, the event graph generation step allocated a data latch, which is inserted on the data bus during this synthesis step. The data is latched with the A/D *RD_L* control signal routed from the protocol controller.

8.2 Protocol Controller

The synthesis of the protocol controller shown in Figure 8-10 is elaborated in this section. The protocol controller coordinates the sequencing and timing of protocol events to synchronize clustered transfers. Specifically, it implements the behavior represented in the event graphs.

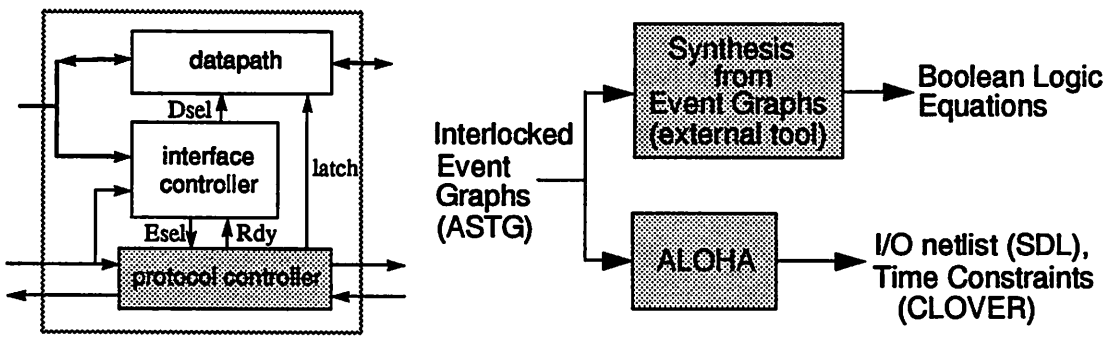


Figure 8-10 : Protocol Controller Generation Using ALOHA and External Tools

There are three general techniques for synthesizing a logic network from the event graphs: asynchronous, mixed asynchronous/synchronous and synchronous. Section 2.1 of Chapter 2 reviewed existing asynchronous and mixed asynchronous/synchronous techniques applicable to automatically creating logic from event graphs. Tools external of ALOHA are already available to synthesize the logic using the reviewed techniques, and they are used instead of developing new ones. In contrast, synthesis of the datapath and interface controller logic is built into ALOHA. The netlist of protocol controller input and output signals and list of time constraints specified in the event graphs are generated by ALOHA.

For the sake of complete coverage, this section surveys all three techniques and comments on their usefulness toward board-level systems. It also presents other important implementation details that apply for any technique used. The section concludes by illustrating the synthesized control logic for the standard four applications.

8.2.1 Synthesis from the Event Graph

Asynchronous Design Style

The techniques based on signal transition graphs can be used to synthesize asynchronous deadlock-free and hazard-free logic [Chu87b] [Meng89] [Martin86] [Moon91]. They work best with event graphs formed from asynchronous handshake protocols, such as the last three examples presented above. Since STG-based techniques are based on the formalism of the representation, they can detect deadlock and hazards in the logic to be synthesized from the graph input itself. The graphs can also be manipulated to eliminate these types of conditions. However, the one real limitation is that these synthesis techniques (rather than their representation) do not effectively work for interfaces with mixed synchronous and asynchronous protocols which are present in many real systems.

At the time of this writing, the only existing tools available to ALOHA for synthesizing the protocol control logic are ones based on the STG. To link into these tools, ALOHA translates the event graphs into the ASTG text format, which is compatible with two of these tools, Async [Jones90] and ASTG [Moon91]. Although they use different algorithms, both can synthesize the asynchronous and delay insensitive finite-state machine from the STG version of the event graph. The FSM implementation consists of boolean equations for each output control signal, where an output is also a state. Each output equation consists of product-terms describing the state of the inputs and outputs that cause the next output level to rise or to fall. ASTG is part of the Berkeley Sequential Interactive System (SIS), and it provides hazard-free and delay-insensitive verification capabilities based on COSPAN [Har'El88][Moon92]. Async and ASTG considers only the event

sequencing constraints in the graphs, ignoring time constraints like its predecessors [Chu87b] [Meng89] [Martin86], requiring the logic implementation to be verified for time constraint violations that may be present in the protocol. Currently, the logic implementation is manually modified to eliminate time constraint violations.

Mixed-mode Design Style

Janus is a tool that can synthesize protocol control logic from acyclic event graphs [Borriello87]. The technique supports a mix of asynchronous and synchronous protocols and also considers time constraints. This makes Janus readily suitable for real system interfaces, but no suitable software implementation for our computer network was available at the time of this writing. All the event graphs for the interface examples in the previous section can be synthesized with this tool. The resulting logic will be hazard-free and satisfy time constraints, but it may not be delay-insensitive (not including time constraints, of course) or deadlock-free. The last one is an artificial limitation. Deadlock is a behavior condition where the sequential circuit gets stuck in an unintended state. Fortunately, the condition can be detected in the original cyclic event graph by checking if the graph is strongly connected. If it is not, then precedence edges can be modified or inserted to make the graph strongly connected, yet satisfying the original precedence constraints. From the new event graph, a deadlock-free logic network is synthesized.

Comparing the asynchronous and mixed-mode techniques, synthesis from acyclic event graphs can have as robust results as the STG techniques as long as the event graph is treated like a STG before synthesis. No matter which representation is used, it should be checked for hazard or deadlock conditions before synthesis. The deadlock condition can be eliminated in the same manner for either representation. However, no formal methods have been developed for specifying how to manipulate the event graph to eliminate deadlock. Currently, that is also done manually.

Synchronous Design Style

Besides the asynchronous or mixed asynchronous-synchronous realizations, the protocol controller can also be implemented with a strictly synchronous finite state machine. In fact, some interface designs today are implemented with synchronous FSMs. The clock samples protocol events according to its (clock's) own periodicity, which must be finer than the smallest time constraint. For example, if the interface converts between an asynchronous protocol with a "30ns max" time constraint and a synchronous protocol with a clock period of 50ns, the protocol controller clock should run using a period less than 30ns. Also, a period greater than 50ns will cause the control logic to miss synchronous events occurring every 50ns. The synchronous method generates a FSM implementation with an equal or less number of states than the previous two methods and which uses the absolute minimum amount of product-terms. The penalty for less complexity is a reduction in performance. This is because a periodic clock does not sample asynchronous events as soon as the events occur. For example, an event occurring 60ns after a 50ns clock sample edge will be detected after the second clock sample (100ns). Whereas, the asynchronous and mixed-mode controllers will react to any event as soon as the event occurs.

Synchronous circuits operating in an asynchronous environment are susceptible to two possible types of failures. First, synchronous control logic for mixed asynchronous and synchronous protocols may have metastability problems. To reduce the chance of metastability, sampling of an input protocol signal can be done with a "metastability hardened" register or with multiple stages of registers [Kleeman87]. In the multi-stage technique, experience with actual implementations [Srivastava91b] shows that two stages are usually sufficient for the minimum amount of additional hardware. Obviously, interfaces that integrate hardware operating from a single global clock are ideal candidates for the synchronous implementation style. For physically large systems, such interfaces can suffer from clock skew if the condition exists.

Perspective

This section has surveyed three implementation styles for the protocol control logic. The appropriate style to choose depends on the types of protocols being converted. Looking at this issue from a high-level synthesis view, the key is that ALOHA can integrate any tool for synthesis from event graphs as they are made available. The flow graph and event graph representations do not restrict the implementation of the event graph behavior to a particular style or synthesis technique. Currently, ALOHA outputs the event graphs in the ASTG format which is compatible with the Berkeley Synthesis tools, but other formats also can be obtained by translation. In addition, information about the time constraints must be passed on to the synthesis tools, the low-level design tools or the simulation/verification tools. Timing information is accessed from the protocol event graphs, and they too can be translated to various tool formats.

8.2.2 Synthesis from Multiple Event Graphs

The protocol controller can execute many event graphs. As inter-module communication progresses from one transfer to the next, the interface controller selects the event graph that accompanies the particular transfer using the Esel word, as shown in the interface template of Figure 6-4. The previously described techniques synthesize logic for one event graph at a time. ALOHA combines the individual solutions to form a complete controller by assigning Esel values and adding *logic* constraints, independent of the event graph synthesis technique used.

The Esel word is N bits wide, where N is the number of unique interlocked event graphs exercised by the protocol controller. Each event graph is assigned an Esel value that enables it. For the Nth event graph, ALOHA forms the specific value by setting the (N-1)th bit to 1 and the other bits to 0. For example, in the TMS320 interface of Figure 7-13, the two event graphs contribute to a two bit Esel word, Esel<1:0>. The interlocked event graph for the write access is selected with the Esel<0> bit, or Esel<1:0>='01'. The interlocked graph for the read access is enabled with the Esel<1> bit, or Esel<1:0>='10'. To capture this information in the flow graph, the sync node is

annotated with the Esel bit that corresponds to the event graph, as shown in Figure 8-11.

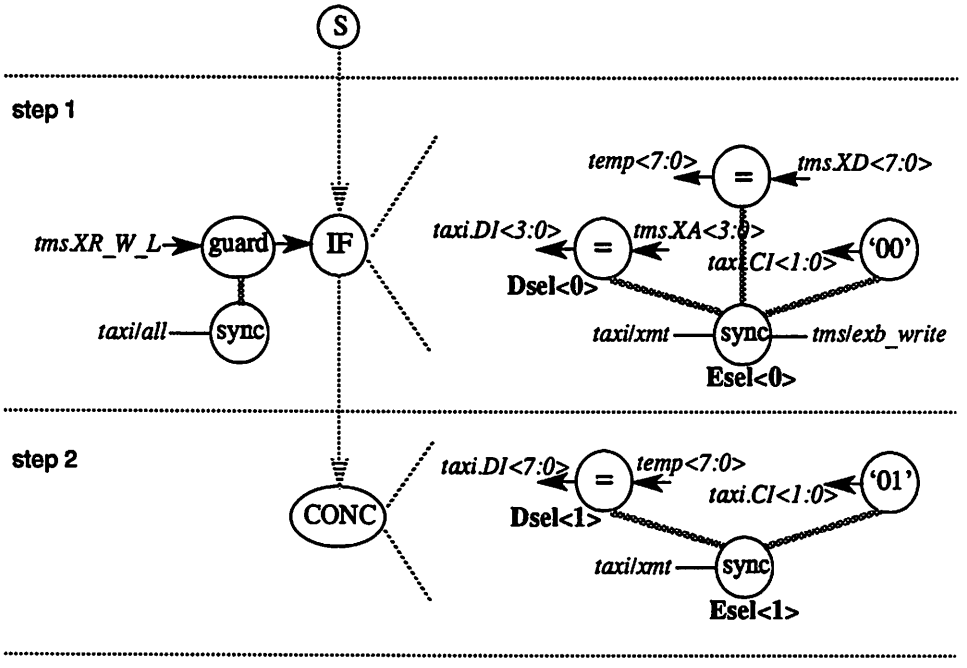


Figure 8-11 : Annotating A Flow Graph with Assigned Dsel and Esel Bits

A protocol controller that implements multiple event graphs also requires additional logic constraints. As described before, the logic equation for an output control signal is synthesized separately from each event graph. Then ALOHA conditions the entire logic expression on the assigned Esel value. This is the first logic constraint. When an output signal is inactive in a particular event graph, then is must be set to a logic 0 level if the signal is active high, or set of a logic 1 level if the signal is active low. This is the second logic constraint. After the equations (for an output) have been synthesized from each event graph, the final logic equation for an output signal is the logical OR of the individual expressions. The resulting equation may have redundant minterms, some of which are retained to eliminate hazard conditions, whilethe others are dropped to minimize the expression.

8.2.3 Expressing Time Constraints

From experience with currently available tools, the designer has to manually deal with time constraints at some point in the event graph synthesis phase or in the low-level design phase. To support this, ALOHA produces a summarized list of all the time constraints in the CLOVER format [Doukas91]. In fact, the list is similar to the tables found in data sheets and specification manuals that describe time constraints or "AC characteristics". The CLOVER format has two basic types of statements. The first describes a general time constraint. As discussed in Chapter 3, a time constraint is an interval of time between two events, and an event is a transition on a signal. Legal transition values are shown in Table 3-2 in Chapter 3. So, the time constraint statement has the format:

signal1 value1 → [min_time max_time] signal2 value2;

The first signal and value pair describe the initial event, the second signal and value pair describe the last event, and the closed interval describes the time interval. The general CLOVER statement is like a weighted guarded command. Examples of time constraints expressed in this format are shown in Figure 8-12. The event graph for the SRAM write cycle is from the module library and repeated from Figure 3-5. The fall and rise transition values are denoted with "f" and "r", respectively. The first CLOVER statement describes the address set-up time constraint. It corresponds to the event precedence from address stable (ADDR s) to chip select falling (CS_L-). The precedence has an explicit "5ns min" time constraint and an implicit infinite max time. The CLOVER statement denotes infinite max time with "...". The time constraint from the chip select rise to the data invalid event has an explicit min and max time.

The second type of CLOVER statement specifically describes pulse widths. A pulse width is just the time interval between a rise and fall transition occurring on the same signal, creating a logic-high pulse or a logic-low pulse. It has the formats:

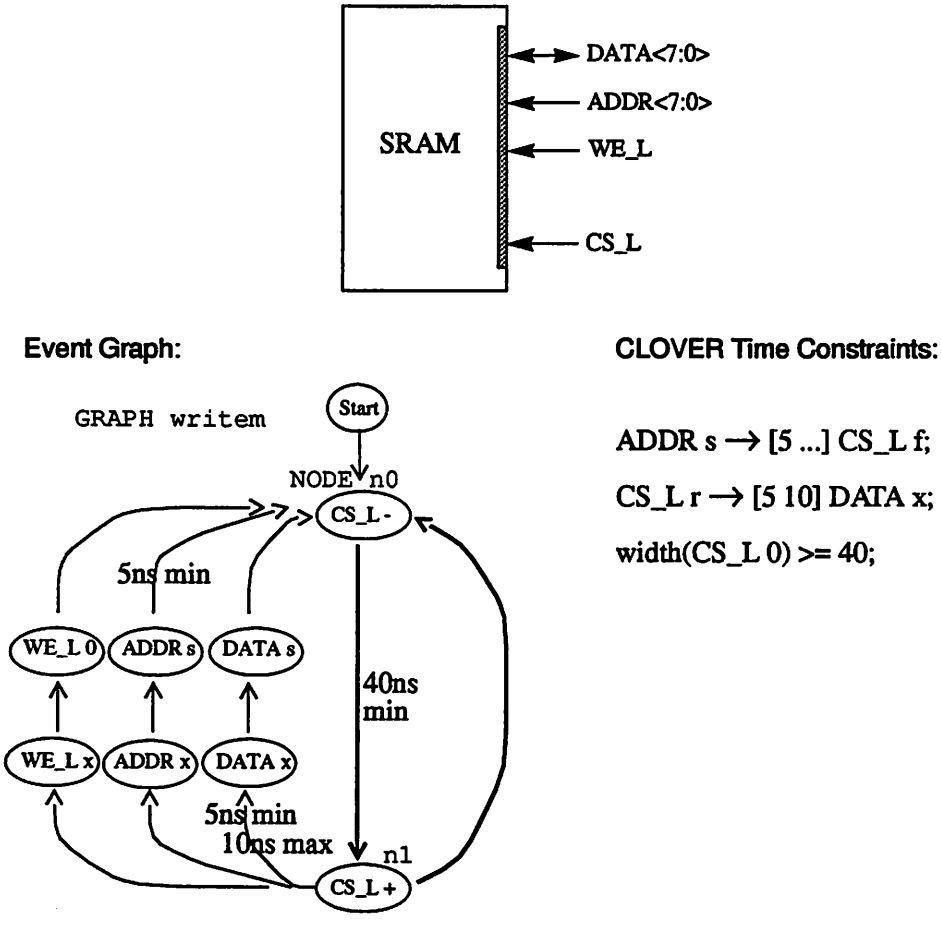


Figure 8-12 : Time Constraints in the CLOVER Format

width (signal value) >= min_time;

width (signal value) <= max_time;

The former is for minimum pulse width, and the latter is for maximum width. The transition value is either "0" or "1", representing a low pulse or high pulse. For example, the time constraint for the chip select low pulse is shown in Figure 8-12. This statement is just a specialized and brief form of the general statement for pulse widths.

8.2.4 Examples

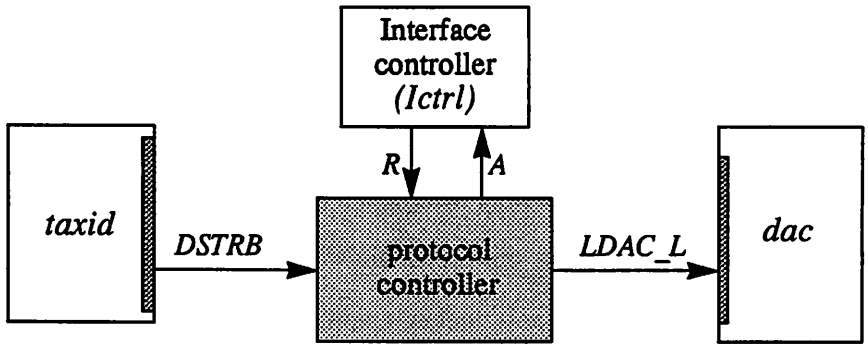
This section presents the boolean equations that implement the event graphs for each of the four standard examples. They were produced by the ASTG tool in the EQN format [Sentovich88]. The Esel signals were inserted after synthesis with ASTG. Included are the time constraints that accompany each example. The time constraints are expressed in the CLOVER format, and they should be compared to their interlocked event graphs in Section 7.4. This section uses each example to highlight important but different features of the protocol controller logic and timing specification.

Optical Link to D/A Module Interface

Figure 8-13 illustrates the I/O structure of the protocol controller inside the optical link to D/A module interface, and it also gives the synthesized sequential boolean equations. The control outputs are the D/A *LDAC_L* signal and the acknowledge *A* signal to the interface controller. From the two event graphs in Figure 7-12, a boolean equation is synthesized for each output signal, as shown in Figure 8-13 in the EQN format. To read an equation, "+" represents a logical OR, "*" represents a logical AND, "!" means the complement of a signal, and ";" denotes the end of the equation. Since the equation expresses sequential logic, output signals have a current and previous value. So, LDAC_L is the current value, while LDAC_L' is the previous value.

Each output equation consists of product-terms describing the state of current inputs and previous outputs that cause the current output level to rise or fall. For example, in the first event graph of Figure 7-12, the output signal, A, is asserted when the R and DSTRB signals both become high. This leads to the "taxid.DSTRB*Ictrl.R" product-term in the boolean equation for the Ictrl.A signal, shown in Figure 8-13. In the second event graph, the output A is asserted when the R signal becomes high, leading to the "Ictrl.R*Ictrl.A'" product-term in the same boolean equation.

Since the boolean equations are generated from two event graphs, the Esel<0> bit enables the product-terms that implement the first event graph in Figure 7-12, while the Esel<1> bit enables




```
dac.LDAC_L =
Esel<0> +
Esel<1>*( !Ictrl.R*dac.LDAC_L' + !taxid.DSTRB*dac.LDAC_L' +
          dac.LDAC_L'*Ictrl.A' + !taxid.DSTRB*Ictrl.A' );

Ictrl.A =
Esel<0>*( taxid.DSTRB*Ictrl.R + Ictrl.R*Ictrl.A' +
          taxid.DSTRB*Ictrl.A' ) +
Esel<1>*( Ictrl.R*Ictrl.A' + !dac.LDAC_L');
```

Figure 8-13 : Protocol Controller for Optical Link to D/A Interface

the second event graph. For example, in the boolean equation for the R output, the product-terms conditioned on Esel<0> produce rise and fall transitions on the output according to the first event graph. Also, the product-terms conditioned on Esel<1> produce output transitions according to the second event graph.

The time constraints that this interface must satisfy are listed in Figure 8-14 using the CLOVER format. The constraints are imposed by the individual ports involved in the communications, so ALOHA generated them from the individual event graphs shown in Figure 7-12. The event graph name corresponding to a set of time constraints is given in the "# Module port/protocol:" comment line. The time constraints are given in "ns" units.

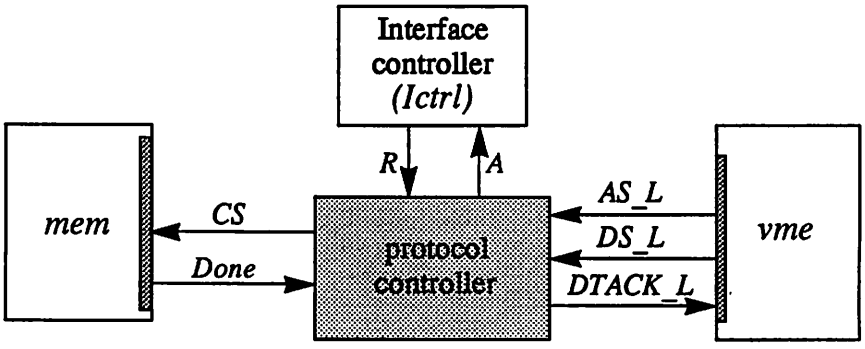
```
# CLOVER Timing Constraints.
#
# Time Unit: "ns"

# Module port/protocol: dac/dacb_write
#
width(dac.LDAC_L 0) >= 50;
dac.LDAC_L r -> dac.D x;
dac.D s -> [30 ...] dac.LDAC_L f;
dac.LDAC_L r -> dac.WR_L 1;
dac.WR_L 0 -> dac.LDAC_L f;
dac.LDAC_L r -> dac.NA_L x;
dac.LDAC_L r -> dac.NB_L x;
dac.LDAC_L r -> dac.NC_L x;
dac.NA_L s -> dac.LDAC_L f;
dac.NB_L s -> dac.LDAC_L f;
dac.NC_L s -> dac.LDAC_L f;
dac.LDAC_L r -> dac.BankSel x;
dac.BankSel s -> dac.LDAC_L f;

# Module port/protocol: taxid/rcvD
#
width(taxid.DSTRB 1) >= 3*period/4;
taxid.DSTRB f -> taxid.Do x;
taxid.Do s -> [2*period/4 ...] taxid.DSTRB r;
```

Figure 8-14 : Time Constraints for Optical Link to D/A Interface

VME System Bus Interface

Figure 8-15 shows the I/O structure and the boolean equations that implement protocol conversion between the VMEbus and a static memory module. The VME *DTACK_L*, memory chip select *CS* and interface controller *A* outputs go to an external block. Using the event graphs from Figures 7-13 and 7-14, the boolean implementations are synthesized for each of these output signals, as shown in Figure 8-15. There is also another equation for the signal *x*, which is actually a signal internal to the protocol controller. As explained before, the event graph structure is modified with extra precedence edges to eliminate hazardous behavior like deadlock. In addition,

sometimes extra events need to be inserted to ensure that the new event graph is implementable, such as using $x+$ and $x-$.




```
vme.DTACK_L =
Esel<0>*( !mem.CS' + vme.DS_L + x'*vme.DTACK_L' ) +
Esel<1>*( mem.CS' + mem.Done );

mem.CS =
Esel<0>*( !mem.Done*mem.CS' + !Ictrl.R*mem.CS' +
          Ictrl.A'*mem.CS' + !x'*mem.CS' + vme.DS_L +
          !vme.DTACK_L' + !x'*!Ictrl.R*!mem.Done +
          !x'*Ictrl.A'*!mem.Done ) +
Esel<1>*( !Ictrl.R*vme.DS_L + vme.DS_L*mem.CS' +
          vme.DS_L*Ictrl.A' + !mem.Done*mem.CS' +
          !Ictrl.R*mem.CS' + Ictrl.A'*mem.CS' );

Ictrl.A =
Esel<0>*( Ictrl.R*Ictrl.A' + !mem.CS' ) +
Esel<1>*( Ictrl.R*Ictrl.A' + !mem.CS' ) =
(Esel<0> + Esel<1>)*( Ictrl.R*Ictrl.A' + !mem.CS );

x =
Esel<0>*( vme.AS_L + x'*mem.CS' ) +
!Esel<1>;
```

Figure 8-15 : Protocol Controller for VMEbus Interface

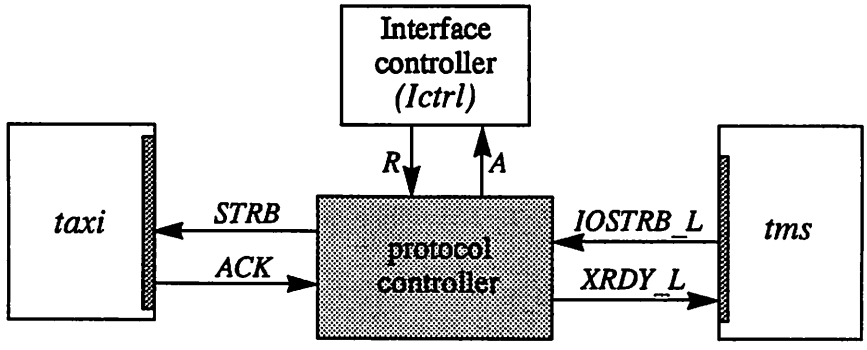
ALOHA assigned the Esel<0> bit to select the event graph that synchronizes the write access, shown in Figure 7-13. The Esel<1> bit selects the event graph in Figure 7-14 to synchronize the read access. The CLOVER time constraints are listed in Figure 8-16.

```
# CLOVER Timing Constraints.
#
# Time Unit: "ns"

# Module port/protocol: mem/writem2
mem.CS_L f -> [40 ...] mem.ACK_L f;
mem.CS_L r -> mem.ADDR x;
mem.ADDR s -> [5 ...] mem.CS_L f;
mem.CS_L r -> mem.WE_L x;
mem.WE_L 0 -> mem.CS_L f;
mem.CS_L r -> [5 ...] mem.DATA x;
mem.DATA s -> mem.CS_L f;

# Module port/protocol: vme/dtb_write
width(vme.AS_L 1) >= 30;
vme.DTACK_L f -> vme.AM x;
vme.DTACK_L f -> vme.IACK_L x;
vme.AM s -> [10 ...] vme.AS_L f;
vme.IACK_L s -> [10 ...] vme.AS_L f;
vme.DTACK_L f -> vme.A x;
vme.DTACK_L f -> vme.LWORD_L x;
vme.A s -> [10 ...] vme.AS_L f;
vme.LWORD_L s -> [10 ...] vme.AS_L f;
vme.DS_L r -> vme.WRITE_L x;
vme.WRITE_L 0 -> [10 ...] vme.DS_L f;
vme.DTACK_L f -> vme.D x;
vme.D s -> [10 ...] vme.DS_L f;

# Module port/protocol: mem/readm
mem.CS_L f -> [45 ...] mem.ACK_L f;
mem.CS_L r -> mem.ADDR x;
mem.ADDR s -> mem.CS_L f;
mem.CS_L r -> mem.WE_L x;
mem.WE_L 1 -> mem.CS_L f;
mem.CS_L r -> [0 30] mem.DATA z;
mem.CS_L f -> [0 45] mem.DATA s;
mem.DATA s -> mem.ACK_L f;

# Module port/protocol: vme/dtb_read
width(vme.AS_L 1) >= 30;
vme.DTACK_L f -> vme.AM x;
vme.DTACK_L f -> vme.IACK_L x;
vme.AM s -> [10 ...] vme.AS_L f;
vme.IACK_L s -> [10 ...] vme.AS_L f;
vme.DTACK_L f -> vme.A x;
vme.DTACK_L f -> vme.LWORD_L x;
vme.A s -> [10 ...] vme.AS_L f;
vme.LWORD_L s -> [10 ...] vme.AS_L f;
vme.DS_L r -> vme.WRITE_L x;
vme.WRITE_L 1 -> [10 ...] vme.DS_L f;
vme.DS_L f -> vme.D s;
vme.D s -> vme.DTACK_L f;
```

Figure 8-16 : Time Constraints for VMEbus Interface

TMS320 to Optical Link Interface

The protocol controller I/O and boolean equation for the TMS320 to optical link interface is illustrated in Figure 8-17. It has three output signals, and a boolean equation is synthesized for each output from the two interlocked event graphs in Figure 7-15. The Esel<0> bit selects the first event graph which interlocks the TMS and TAXI protocols. The Esel<1> bit selects the second event graph for synchronizing a transfer from an interface register to the TAXI port.




```
tms.XRDY_L =
Esel<0>*( tms.IOSTRB_L*taxi.STRB' +
          tms.IOSTRB_L*tms.XRDY_L' +
          tms.XRDY_L'*taxi.STRB' ) +
Esel<1>;

taxi.STRB =
Esel<0>*( !tms.XRDY_L'*!taxi.ACK*Ictrl.R*!Ictrl.A' +
          !taxi.ACK*taxi.STRB' +
          !tms.XRDY_L'*taxi.STRB' +
          !Ictrl.A'*taxi.STRB' ) +
Esel<1>*( Ictrl.R*!taxi.ACK*!Ictrl.A' +
          !taxi.ACK*taxi.STRB' + taxi.STRB'*!Ictrl.A' );

Ictrl.A =
Esel<0>*( taxi.STRB' + Ictrl.R*Ictrl.A' ) +
Esel<1>*( taxi.STRB' + Ictrl.R*Ictrl.A');
```

Figure 8-17 : Protocol Controller for TMS320 to Optical Link Interface

The time constraints that the interface must fulfill are shown in Figure 8-18. These constraints come from the external port protocols, represented with the "xmt" event graph for the TAXI protocol and the "exb_write" event graph for the TMS protocol. Accordingly, ALOHA generates

two sets of time constraints. As shown in the figure, the first set is the time constraints imposed by the TAXI port. For example, the constraint "*taxi.STRB* r → [0 40] *taxi.ACK* r" corresponds to the *40ns max* time constraint on the *STRB+* to *ACK+* precedence edge in Figure 7-15. The "*taxi.DI* s → [5 ...] *taxi.STRB* r" time constraint describes the *5ns min* set-up time for the TAXI *DI* data with respect to the *STRB+* event. The "*taxi.STRB* r → [15 ...] *taxi.DI* x" constraint specifies the hold time for the TAXI *DI* data with respect to *STRB+*. The precedence edge [*STRB-* → *STRB+*] corresponds to the time constraint "width(*taxi.STRB* 0) >= 15".

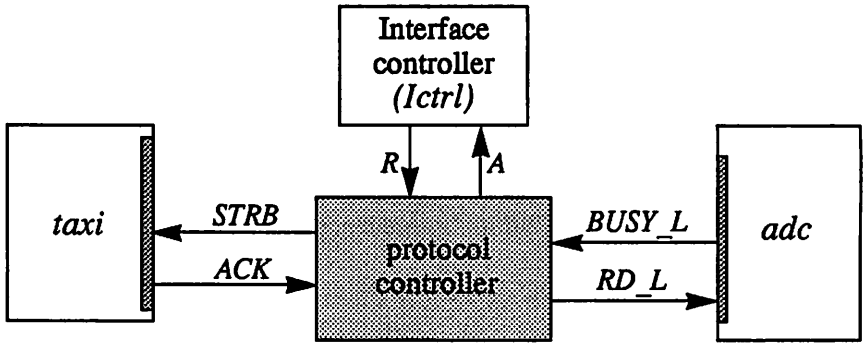
The second set of time constraints are imposed by the TMS port. The time constraint "*tms.XA* s → *tms.IOSTRB_L* f" specifies a *0ns min* address set-up time before the *IOSTRB_L-* event occurs. This comes from the [*XA* s → *IOSTRB_L-*] precedence edge in Figure 7-15. This edge has no explicit time constraints accompanying it, like the other examples above. In this case, the unweighted precedence edge implies a time interval from 0 to infinity, which is equivalent to *0ns min*.

```
# CLOVER Timing Constraints.
#
# Time Unit: "ns"

# Module port/protocol: taxi/xmt
#
taxi.STRB r -> [0 40] taxi.ACK r;
taxi.STRB f -> [0 20] taxi.ACK f;
taxi.STRB r -> [15 ...] taxi.CI x;
taxi.CI s -> [5 ...] taxi.STRB r;
taxi.STRB r -> [15 ...] taxi.DI x;
taxi.DI s -> [5 ...] taxi.STRB r;
width(taxi.STRB 0) >= 15;

# Module port/protocol: tms/exb_write
#
tms.IOSTRB_L r -> tms.XD x;
tms.XD x -> tms.IOSTRB_L f;
tms.IOSTRB_L r -> tms.XA x;
tms.XA s -> tms.IOSTRB_L f;
tms.IOSTRB_L r -> tms.XR_W_L x;
tms.XR_W_L 0 -> tms.IOSTRB_L f;
```

Figure 8-18 : Time Constraints for TMS320 to Optical Link

A/D Module to Optical Link Interface

This protocol controller synchronizes the transfer of a header and then a block transfer both from the A/D module to TAXI optical link. As shown in Figure 8-19, the boolean equations that implement synchronization are synthesized from the two event graph in Figure 7-16. The Esel<0> bit enables the first event graph for the header transfer. The Esel<1> bit enables the second event graph for the block transfer. The accompanying time constraints are listed in Figure 8-20.




```
adc.RD_L =
Esel<0> +
Esel<1>*( adc.BUSY_L*!taxi.STRB'*Ictrl.A' +
          !Ictrl.R*adc.RD_L' + adc.BUSY_L*adc.RD_L' +
          adc.RD_L'*Ictrl.A' + !taxi.STRB'*adc.RD_L' );

taxi.STRB =
Esel<0>*( !taxi.ACK*Ictrl.R*!Ictrl.A' +
          !taxi.ACK*taxi.STRB' + taxi.STRB'*!Ictrl.A' ) +
Esel<1>*( !taxi.ACK*adc.RD_L' + !taxi.ACK*taxi.STRB' +
          taxi.STRB'*adc.RD_L' );

Ictrl.A =
Esel<0>*( taxi.STRB' + Ictrl.R*Ictrl.A' ) +
Esel<1>*( !adc.RD_L' + Ictrl.R*Ictrl.A' );
```

Figure 8-19 : Protocol Controller for A/D Module to Optical Link Interface

In Figure 8-19, the boolean equation for the *RD_L* output signal consists of two parts. The last part consists of product-terms that implement the *RD_L* rise and fall transitions shown in Figure 7-16. Comparing the two interlocked event graph in the figure, the *RD_L* signal is only active during the

block transfers. It remains inactive during the header transfer of control step 1. On this control step, the interface controller issues Esel<0>=1 and Esel<1>=0 connected to the protocol controller, which in turn disables the product-terms and the *RD_L* signal. *RD_L* is active low, so it is set to logic level "1" to disassert it.

```
# CLOVER Timing Constraints.
#
# Time Unit: "ns"

# Module port/protocol: taxi/xmt
#
taxi.STRB r -> [0 40] taxi.ACK r;
taxi.STRB f -> [0 20] taxi.ACK f;
taxi.STRB r -> [15 ...] taxi.CI x;
taxi.CI s -> [5 ...] taxi.STRB r;
taxi.STRB r -> [15 ...] taxi.DI x;
taxi.DI s -> [5 ...] taxi.STRB r;
width(taxi.STRB 0) >= 15;

# Module port/protocol: adc/b_read1
#
adc.RD_L f -> [0 70] adc.BUSY_L r;
adc.RD_L r -> adc.CS_L x;
adc.CS_L s -> adc.RD_L f;
adc.RD_L f -> [0 70] adc.DB s;
adc.DB s -> adc.RD_L r;
adc.RD_L r -> [5 50] adc.DB x;
adc.DB x -> adc.RD_L f;
width(adc.RD_L 0) >= 75;
```

Figure 8-20 : Time Constraints for A/D Module to Optical Link

8.3 Interface Controller

The interface controller implements the flow graph schedule and configures the datapath and protocol controller on each control step, using the Dsel and Esel words, respectively. The interface controller is synthesized onto a hardwired finite state machine, where a control step of the flow graph is equivalent to a single state of the FSM. Highlighted in Figure 8-21, generation of the interface controller and also the timing interaction between the interface controller and the rest of the interface are described below.

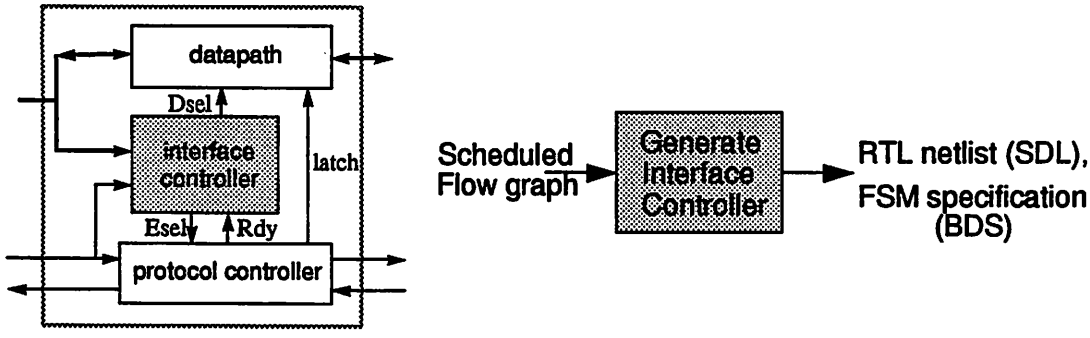


Figure 8-21 : Interface Controller Generation and Output Formats from Synthesis

8.3.1 Compiling the FSM Specification

At this point in the synthesis process, the flow graph schedule is represented by hierarchical control nodes (conditional IF and concurrent CONC) and by control precedence edges. Recall, that hierarchical iteration nodes have been unrolled into concurrent nodes. A control node, or control step, corresponds to a unique FSM state, while a precedence edge corresponds to a state transition. Guard nodes that drive the IF control nodes determine which one of several states to branch to, so it maps to a condition on the state transition. ALOHA constructs a state transition graph from the schedule using these correspondences. Since the state graph is another internal representation, it is then translated into a specification of a Moore finite state machine in the BDS format [Segal88].

Creating FSM States

A FSM state is created for each hierarchical control node in the flow graph. Synthesis assigns a symbolic name and binary representation to each state as it is created. During translation to the BDS specification, the state maintains both identifications. In a Moore implementation, the Dsel and Esel select words are outputs dependent on the state. The specific output value is formed from all the annotated Dsel and Esel bits in the control step under consideration. Figure 8-22 illustrate this step with the TMS320 interface example. This figure only shows the control flow; the full flow graph is shown in Figure 8-11. In Figure 8-22, the IF control node transforms to the first state, which outputs the Dsel word "01" and the Esel word "01". The CONC control node maps to the second state, which outputs the Dsel word "10" and the Esel word "10".

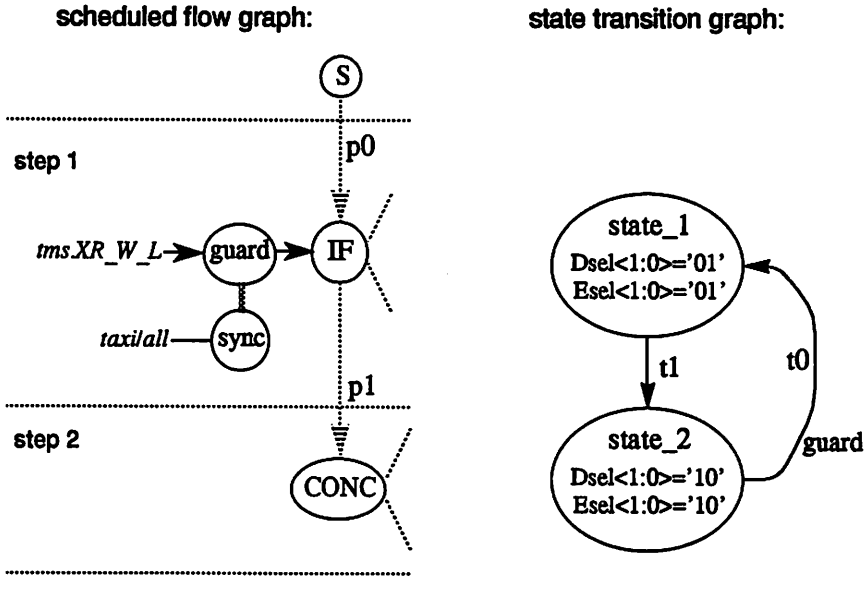


Figure 8-22 : Deriving the State Transition Graph from the Flow Graph

Creating State Transitions

Once all the FSM states have been created from the schedule, the next step is deriving the transitions from one state to the next. For each state, synthesis gets the corresponding control step

in the flow graph, and then traverses the output precedence edges. Each edge leads to the next control step, which determines the next state. When traversing the output precedence edges, there are three possible situations.

First, only one precedence edge exists and it leads to a concurrent node. The state corresponding to the concurrent node is the only possible next state. The precedence edge labeled "p1" in Figure 8-22 demonstrates this case. It maps to the unconditional state transition "t1".

Second, the precedence edge leads to one or more IF nodes such as the "p0" edge in the Figure 8-22. A state transition, such as "t0", is created for each branch and conditioned on the guard that drives the corresponding IF node. The FSM uses *information* signals from external ports and its current state to evaluate the guard. When there are multiple IF nodes, only one guard evaluates to true, so that the branches are mutually exclusive. If no guards evaluate to true, then no branch is taken, and the interface controller remains in the same state. On the next FSM cycle, the controller evaluates the guards again until a true occurs.

In the last case, the original control step is the final step in the schedule, and there are no output precedence edges. The concurrent node in Figure 8-22 is an example of such a control step. The interface controller actually repeats the schedule, so the state transition returns control to the first state, as illustrated by the transition labeled "t0" in the figure.

Optimizations

The constructed state graph can be optimized to simplify the FSM logic implementation. So far, the interface controller is assumed to be sequential. In some cases, it can be reduced to a combinational controller. This optimization is made by examining the flow graph for exactly one control step. For example, the VMEbus interface in Figure 8-28 meets these requirements, and its state graph is reduced into a combinational truth table. In a sense, its interface controller is always in a default state which is testing for a read or write access.

Another optimization seeks to reduce the number of output signals from the interface controller, instead of reducing the number of states as in the previous case. Figure 8-22 illustrates this optimization. Since the Dsel and Esel word outputted from every state is identical to one another, they can be merged onto one select word. The number of output signals in this case is cut in half.

After the state graph optimizations, ALOHA generates the BDS specification of a FSM that configures the datapath and protocol controller according to the schedule captured in the flow graph. Examples are shown later.

8.3.2 FSM Timing

The FSM logic compiled from the flow graph schedule describes the computational engine of the interface controller. The complete controller consists of the FSM and a timing block, as shown in Figure 8-23. The FSM has state memory and a combinational core which is driven by external information signals and the internal current state. The input information signals are usually status words like read/write, address that needs to be decoded and possibly data. *ResetI* is a special input signal introduced by ALOHA to the interface controller structure. Under system control, asserting the signal resets the interface. Specified in the BDS combinational logic format, the core produces the next state and the output Dsel/Esel select words, both of which are latched into a set of memory elements.

Just like inter-module communications, the passing of information between the FSM and the external ports, datapath and protocol controller need to be synchronized. The timing block contains logic that generates events for the memory clock signal and the Ictrl handshake.

The Timing Block

In the timing block consists of three circuits. First, the completion circuit emulates the computational delay of the FSM core. The *Go1* or $\overline{Go2}$ signal carry an event that triggers the beginning of a computation cycle. The completion circuit detects either event (edge-triggered) and

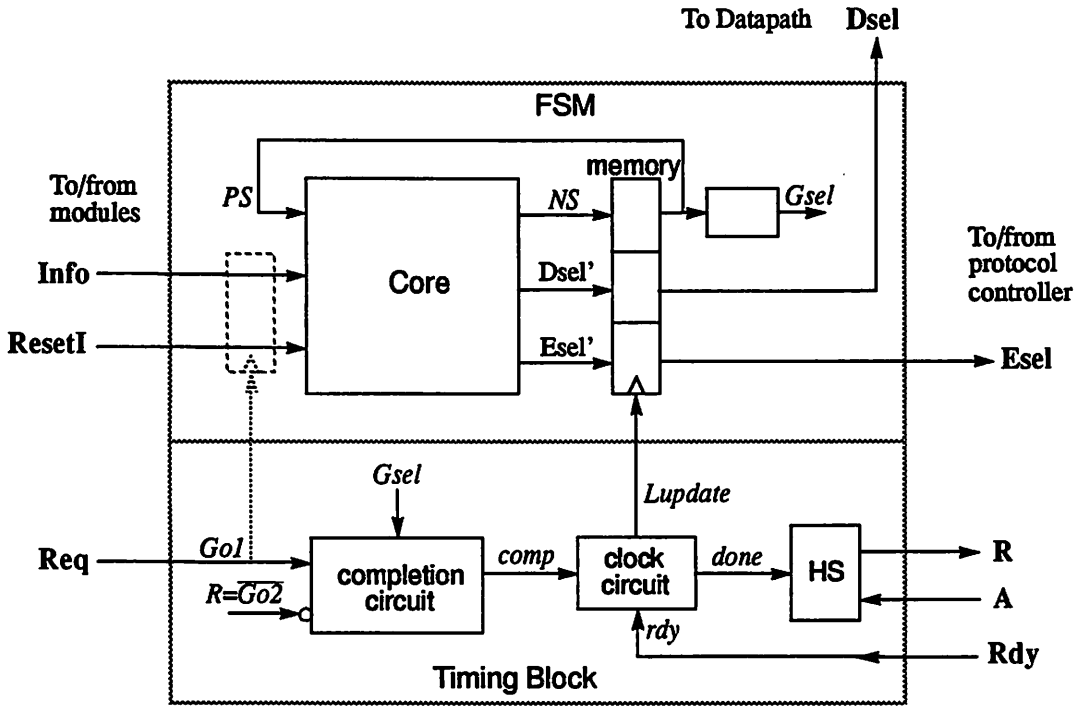


Figure 8-23 : Internal Structure of the Interface Controller

then delays the event by the worst case propagation delay of the combinational core. After this amount of time, the next state *NS* and Dsel/Esel outputs arrive at the memory elements, and the completion circuit generates a "completion" event on the *comp* signal.

As shown in the figure, there are two *Go* signals. *Go1* triggers any computation cycle where the FSM next state depends on input information words. The accompanying request signal, *Req*, indicates the input information words are valid, so it is connected to *Go1*. For example in Figure 8-22, the TMS320 to optical link interface controller monitors the *XR_W_L* status, an input to the FSM core, and its validity is indicated by the *IOSTRB_L*, as shown in Figure 7-15. The falling *IOSTRB_L* event is conveyed on *Go1* and detected by the completion circuit. The second $\overline{Go2}$ signal is generated within the interface controller. It triggers any computation cycle where the

FSM next state depends only the present state *PS* and not on any external information signals. For example, advancing to state_2 in the state transition graph of Figure 8-22 only depends on the present state, whereas advancing to state_1 depends on the input signal *XR_W_L*. So, the second computation cycle should be triggered by an internal event rather than an external event like TMS320 *IOSTRB_L-*. In other words, the interface controller self-times its own computations using the $\overline{Go2}$ signal. The request signal R produced by the handshake circuit "HS" is fed back to the $\overline{Go2}$ signal, and its falling edge triggers a new computation cycle. The Gsel signal selects between the two Go signals, and it is a combinational function of the present state. Figure 8-24 illustrates a possible implementation of the completion circuit. The two edge-triggered flip-flops perform edge detection, and the delay element should have a delay that matches the worst case propagation delay of the FSM core. In the case where there is no external *Req* signal that accompanies the input information, the completion circuit reduces to only the delay element driven by $\overline{Go2}$. In the case where the input information is used in every computation cycle, the completion circuit also reduces to the delay element but driven by *Go1*.

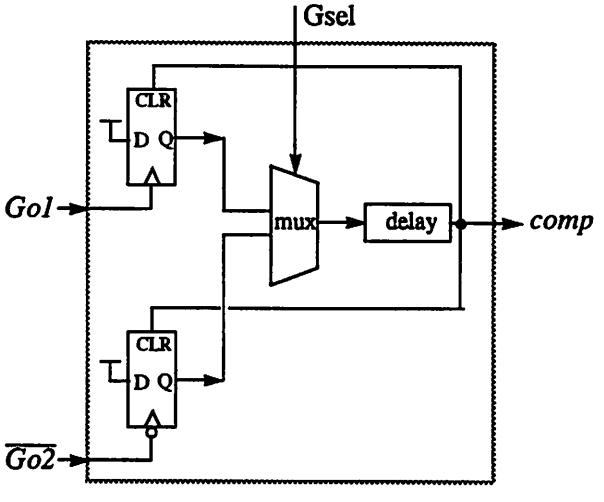


Figure 8-24 : Completion Circuit Implementation

In Figure 8-23, the clock circuit derives a local clock, called *Lupdate*, to latch the next state word (*NS*) and the Dsel/Esel configuration words into the memories. The asynchronous clock pulse is

generated when the next state and configuration words have been computed, as indicated by the completion event on the *comp* signal, and when the current transfer has finished executing, as indicated by the *rdy* event. The protocol controller generates the ready event when all the external protocol signals begin to disassert, which indicates the end of the current transfer. When the clock circuit has detected both these events, an internal delay element mimics the delay of the memory elements, and produces a *done* pulse. Hazardous behavior can occur if the source port for the input information words does not have an acknowledge protocol event, because the information word may disassert while the FSM is still computing. In this case, the information words should be latched with the *Go1* signal, as shown in Figure 8-23. A possible implementation for the clock circuit is shown in Figure 8-25. Like the completion circuit, it uses two edge-triggered flip-flops to detect signal events on the *comp* and *rdy* signals. The AND gate forces the clock circuit to wait for both these events to occur before the delay element produces the done pulse.

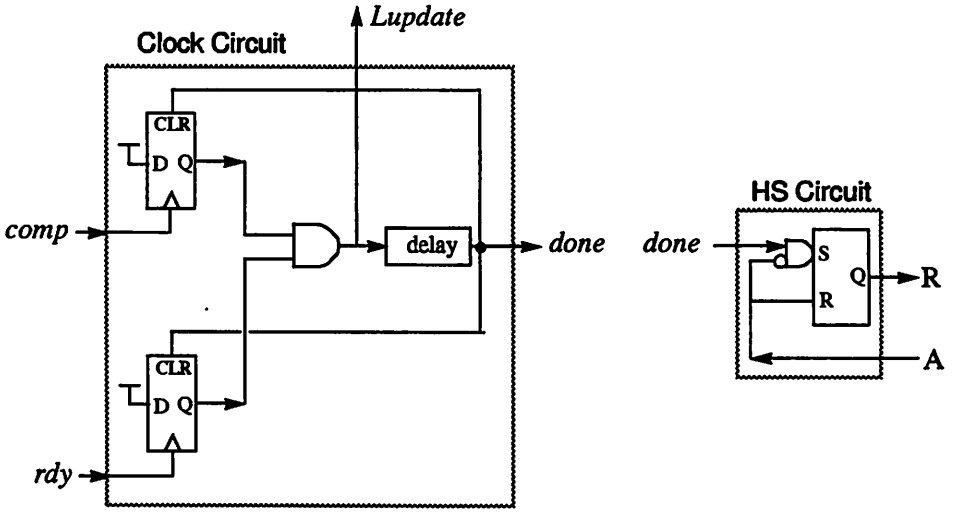


Figure 8-25 : Clock and Handshake Circuit Implementations

As described in the previous chapter, the Ictrl handshake synchronizes the action of the protocol controller to the interface controller. The "HS" circuit in Figure 8-23 generates the Ictrl request (*R*) from the *done* signal and the acknowledge (*A*) issued from the protocol controller. The *done* signal

triggers the handshake since a done event indicates that the memories have latched in new Esel words used by the protocol controller (and Dsel words used by the datapath). Figure 8-25 shows a possible implementation for the handshake circuit.

Perspective

Local clock generation is the key to timing the FSM evaluation of the next state and outputs and timing the update of memory. This concept allows a synchronous design style for the FSM along with asynchronous operation in a mixed asynchronous/synchronous environment. In effect, the interface controller is a locally clocked asynchronous FSM, combining the advantages of both methodologies.

In the synchronous design style, memory is a clocked latch compared to a SR-latch or delay element in the asynchronous FSM case. The synchronous style has much easier design techniques and much more efficient logic implementations compared to the asynchronous alternative, because it is unaffected by combinational hazard and race problems that burdens the asynchronous style. However, to avoid metastability and retain the performance of an asynchronous design, the FSM memory elements are latched with an asynchronous clock that is derived from local events rather than from a periodic clock. The combined advantages are easy and efficient implementations and good performance.

The locally clocked scheme described above works best for state machines that perform computations rather than protocol conversion. It is similar to the scheme used in [Hayes81]. Of course, the technique described for generating the local clock is not the only way to do it. Research in local clock generation techniques from [Nowick91] [Chuang73] [Rey 74] [Yenersoy79] are best applied to synthesis of protocol control logic.

8.3.3 Examples

The FSM descriptions of the interface controllers for the four standard examples are presented here. This includes the state transition graphs and the equivalent BDS format descriptions, and explanations of the state graph apply equally well to the BDS equivalent. In the BDS format, the "!" character denotes a comment line.

The descriptions only account for the FSM core in the interface controller. ALOHA also generates a register-transfer netlist comprised of the combinational core, the memory elements and timing circuits. The BDS description is a parameter of the core cell. This section uses each example to highlight important yet different features of the interface controller implementation and specification.

Optical Link to D/A Module Interface

Figure 8-26 illustrates the complete state transition graph derived from the scheduled flow graph. The equivalent BDS description of the state graph is shown in Figure 8-27. As explained in Section 8.3.1, each control step corresponds to a state node in the state graph. For example, control step 1 maps to the state node with the symbolic name "state_1," which in turn carries over to the BDS description. Similarly, the second control step maps to the "state_2" node in the state transition graph and the "state_2" value in the BDS description. The BDS description also assigns an integer code to each state as shown in the "State Assignments" portion of the description. The integer code is actually a short-hand notation for the equivalent binary code.

The state called "state_top" is a reset state that the interface controller starts up at when the interface is reset. It corresponds to the start node in the flow graph. This state is not implied by the IDL input specification, and ALOHA introduces the reset state for all sequential interface controllers. Discussed in Section 8.3.2, the *Resetl* signal functions as an interface reset signal. Shown with the shaded edges, asserting *Resetl* forces the interface controller to return to the reset state regardless of which state it is currently in. So, all the control step states have an output

transition that leads to the reset state, which is taken if *ResetI* has a logic "1" level. The other output transitions can be taken as long as *ResetI* has a logic "0" level, as illustrated with the solid edges. In the BDS description, the first IF statement of the "main" ROUTINE describes the reset behavior.

The interface controller FSM generates Dsel and Esel output values, as shown within the state nodes of the state transition graph. The Dsel word controls multiplexors and tri-state buffers created when the datapath was synthesized. The Esel word selects an appropriate event graph and was assigned when the protocol controller logic was synthesized. During both those synthesis phases, the data flow node (in the flow graph) that corresponds to a mux or tri-state buffer and the sync node (in the flow graph) that corresponds to an event graph are annotated with appropriate Dsel or Esel bit. When the state node is created, the Dsel bit vector is formed from the annotated data flow nodes in the corresponding control step. Similarly, the Esel bit vector is formed from the annotated sync nodes. For example, the sync node in the first control step of Figure 8-26 is annotated with the Esel<1> bit, and the Esel<1> bit is inactive. So, the Esel<1:0> output word for the "state_1" state is '01'. In the second control step, the sync node is annotated with Esel<1>. The corresponding "state_2" state produces an Esel<1:0> output value of '10'. No data flow nodes have a Dsel bit annotated to it, so the FSM for this particular interface does not produce a Dsel word. The output logic description is shown in the "Output Logic" section of the BDS description.

The interface controller FSM implements the state transition graph in Figure 8-26. The controller is implemented as shown in Figure 8-23. For this example, the completion circuit within the timing block uses only the *GoI* input. This simplification is made, because the input information is used in every FSM cycle to compute the next state and outputs as shown in the signal transition graph.

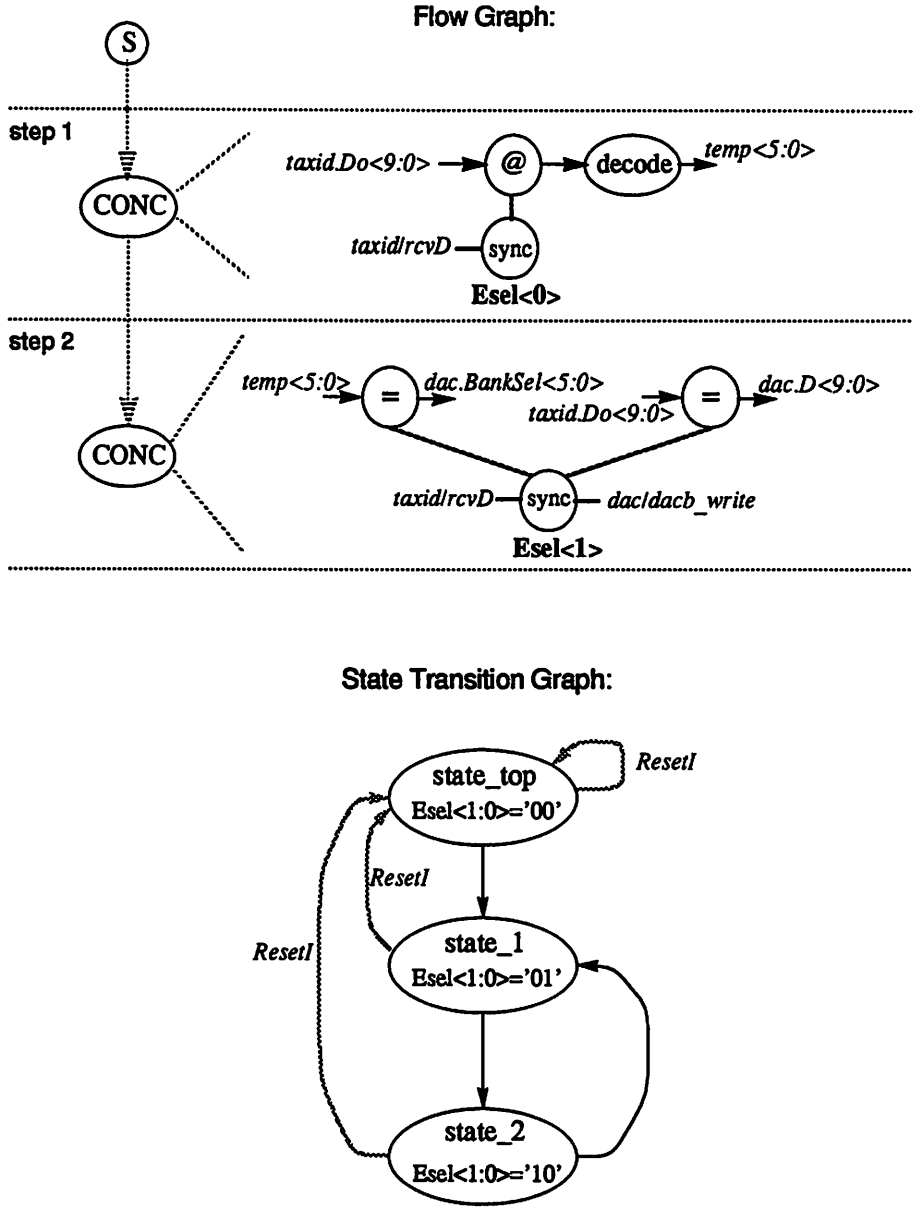


Figure 8-26 : Interface Controller for the Optical Link to D/A Interface

```
MODEL IctrlFsm
!! Outputs
   nextState<1:0>,
   zEsel<1:0>
   =
!! Inputs
   rst__powerup_reset<0:0>,
   taxic__Co<1:0>,
   ResetI<0:0>,
   currState<1:0>;

!! State Assignments
CONSTANT state_top = 0;
CONSTANT state_1 = 1;
CONSTANT state_2 = 2;

! Subroutines:
ROUTINE reset<0:0>(xResetI<0>);
   RETURN (xResetI);
ENDROUTINE reset;

! Main routine:
ROUTINE main;
!! State Transition Logic
   nextState = currState; !default value

   IF reset(ResetI<0:0>)
      THEN nextState = state_top
   ELSE SELECT currState FROM
      [state_top]: BEGIN
         nextState = state_1; END;
      state_1]: BEGIN
         nextState = state_2; END;
      [state_2]: BEGIN
         nextState = state_1; END;
      [OTHERWISE]: BEGIN
         nextState = state_top; END;
   ENDSELECT;

!! Output Logic
   zEsel = 00#2;
   SELECT nextState FROM
      [state_1]: BEGIN
         zEsel = 01#2; END;
      [state_2]: BEGIN
         zEsel = 10#2; END;
   ENDSELECT;
ENDROUTINE main;

ENDMODEL IctrlFsm;
```

Figure 8-27 : BDS Description of Interface Controller FSM

VME System Bus Interface

The interface controller for the VME system interface illustrates how optimizations generate a combinational controller rather than a sequential controller. As shown in Figure 8-28, the flow graph schedule does not contain any precedence between the read and write cycles. It only specifies testing the input address and read/write status and branching to the write or read cycle. Since the flow graph does not imply sequential control, the state transition graph is reduced to a combinational representation. The flow graph shows three Dsel bits and two Esel bits. For the write cycle, synthesis creates the Dsel<2:0> value '010' and the Esel<1:0> value '01'. For the read cycle, Dsel<2:0> takes on the value '101' and Esel<1:0> takes on the value '10'.

The internal structure of the resulting interface controller is shown in Figure 8-28. The core is specified by the BDS description in Figure 8-29. As illustrated by the structure and the "main" routine in the BDS description, the core accepts VME address and read/write status inputs and produces the Dsel and Esel words, and the core is independent of a state. The output latches in Figure 8-28 buffer the two select words from the datapath and protocol controller. There is no feedback from the core output back to its input, and hence no state. Also, there is no reset signal.

In the BDS description, the subroutines correspond to the guards in the flow graph. Since this example has two guards that control the IF nodes, there are two BDS subroutines. The "main" routine corresponds to the overall control flow in the flow graph. For this example, the first IF statement produces the Dsel/Esel values for the write access, while the second IF statement produces the select values for the read access.

The original IDL specification, in Figure 4-7 conditions both the write and read on the result of a decode function described with its own BDS file (not shown). To account for this, the controller core includes a combinational subcell (not shown) described by the decode BDS. The subcell's output is the decode result. This output is an input to a main cell described by the BDS description of Figure 8-29. In this description, the input called "return_6" corresponds to the subcell's output.

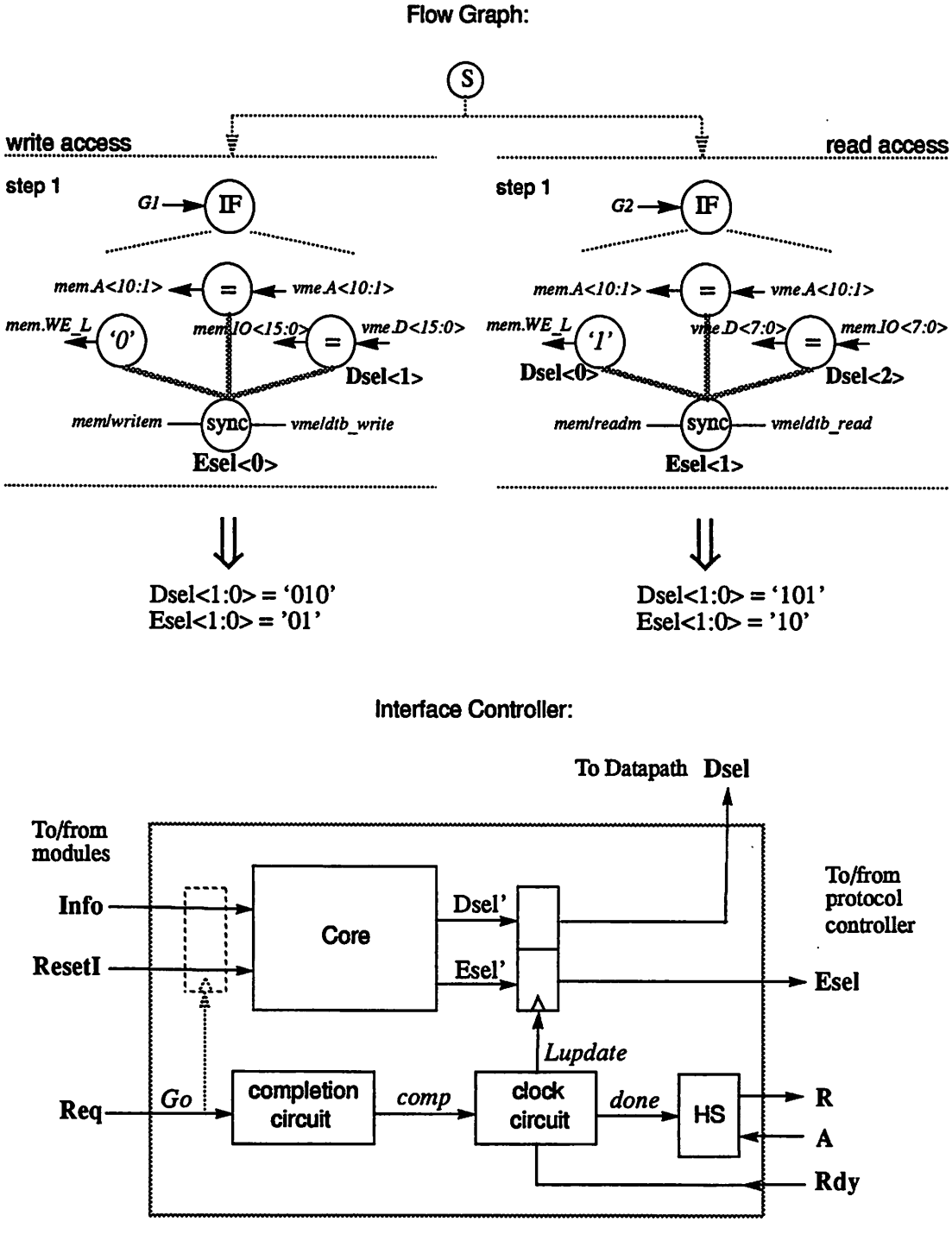


Figure 8-28 : Interface Controller for the VMEbus Interface

```
MODEL IctrlFsm
!! Outputs
   zDsel<2:0>,
   zEsel<1:0>
   =
!! Inputs
   return_6<0:0>,
   vme__WRITE_L<0:0>,
   vme__A<14:11>;

! Subroutines:
ROUTINE cond_4<0:0>(xp_91<0:0>,xp_92<0:0>,dummy);
   STATE g_6;
   STATE g_5;

   g_6 = xp_91<0:0> EQL 1#2 ;
   g_5 = xp_92<0:0> EQL 0#2 ;
   RETURN (g_6 AND g_5 );
ENDROUTINE cond_4;

ROUTINE cond_9<0:0>(xp_93<0:0>,xp_94<0:0>,dummy);
   STATE g_11;
   STATE g_10;

   g_11 = xp_93<0:0> EQL 1#2 ;
   g_10 = NOT (xp_94<0:0> EQL 0#2 );
   RETURN (g_11 AND g_10 );
ENDROUTINE cond_9;

! Main routine:
ROUTINE main;
!! Output Logic
   zDsel = 000#2; ! default value for outputs
   zEsel = 00#2;

   IF cond_4(return_6<0:0>,vme__WRITE_L<0:0>,0) THEN
      BEGIN
      zDsel = 010#2;
      zEsel = 01#2;END
    ELSE IF cond_9(return_6<0:0>,vme__WRITE_L<0:0>,0) THEN
      BEGIN
      zDsel = 101#2;
      zEsel = 10#2;
   END;

ENDROUTINE main;

ENDMODEL IctrlFsm;
```

Figure 8-29 : BDS Description for Interface Controller

TMS320 to Optical Link Interface

The complete state transition graph generated from the schedule for the TMS320 to TAXI optical link interface is illustrated in Figure 8-30. The equivalent BDS description is shown in Figure 8-31. The FSM and timing are implemented as shown in Figure 8-23.

The IF control node maps to the "state_1" state node, while the CONC control node maps to the "state_2" state node. The "state_top" reset state "state_top" is introduced by synthesis. The *ResetI* attached to the input transition shows that the reset state is entered whenever the *ResetI* input is asserted. Similarly, the *XR_W_L* complement on the input transitions to "state_1" shows that state is entered from "state_top" or "state_2" when that condition is met. The condition comes from the guard node that enables the IF behavior (see corresponding IDL description in Figure 4-8). In general, the condition specified with the IF behavior maps to conditions attached to the transitions in the state transition graph.

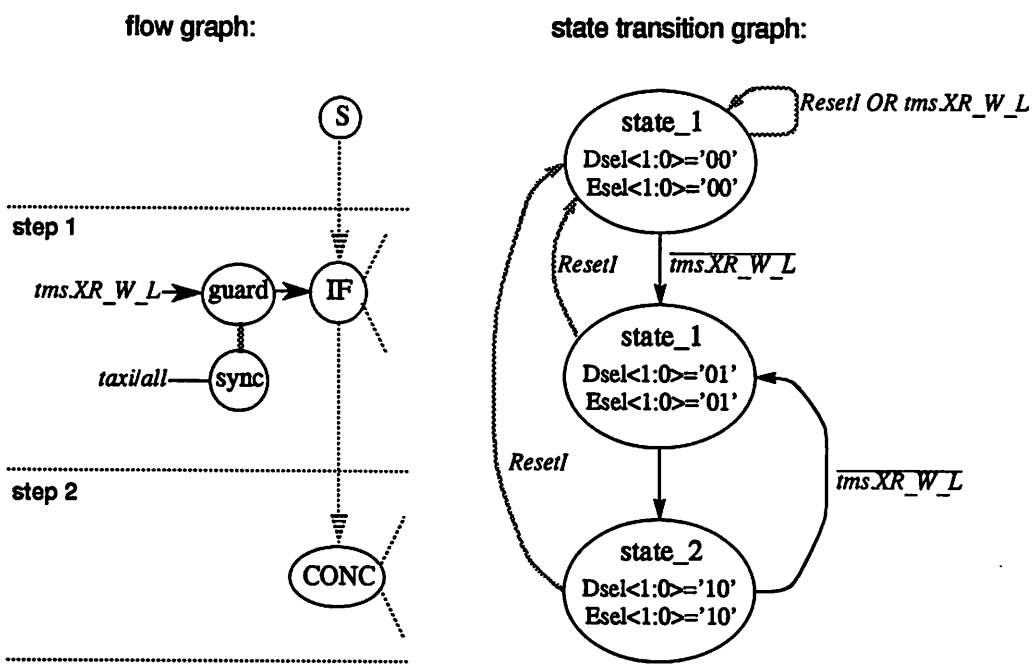


Figure 8-30 : Interface Controller for TMS320 to Optical Link Interface

```
MODEL IctrlFsm
!! Outputs
   nextState<1:0>, zDsel<2:0>, zEsel<1:0> =
!! Inputs
   tms__XR_W_L<0:0>, ResetI<0:0>, currState<1:0>;

!! State Assignments
CONSTANT state_top=0, state_1=1, state_2=2;

! Subroutines:
ROUTINE reset<0:0>(xResetI<0>);
   RETURN (xResetI);
ENDROUTINE reset;

ROUTINE cond_6<0:0>(xp_78<0:0>,dummy);
   STATE g_8;
   g_8 = xp_78<0:0> EQL 0#2 ;
   RETURN (g_8);
ENDROUTINE cond_6;

! Main routine:
ROUTINE main;
!! State Transition Logic
   nextState = currState; ! default value

   IF reset(ResetI<0:0>)
   THEN nextState = state_top
   ELSE SELECT currState FROM
      [state_top]: BEGIN
         IF cond_6(tms__XR_W_L<0:0>,0) THEN
            nextState = state_1; END;
      [state_1]: BEGIN
         nextState = state_2; END;
      [state_2]: BEGIN
         IF cond_6(tms__XR_W_L<0:0>,0) THEN
            nextState = state_1; END;
   [OTHERWISE]: BEGIN
      nextState = state_top; END;
   ENDSELECT;

!! Output Logic
   zDsel = 00#2; zEsel = 00#2; !default value

   SELECT nextState FROM
      [state_1]: BEGIN
         zDsel = 01#2; zEsel = 01#2; END;
      [state_2]: BEGIN
         zDsel = 10#2; zEsel = 10#2; END;
   ENDSELECT;
ENDROUTINE main;

ENDMODEL IctrlFsm;
```

Figure 8-31 : BDS Description for Interface Controller FSM

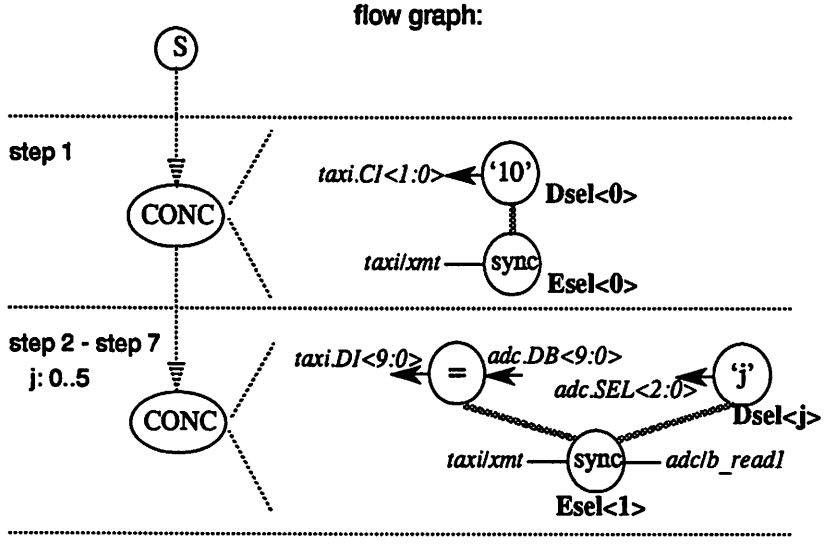
A/D Module to Optical Link Interface

Figures 8-32 and 8-33 show the state transition graph and BDS description generated from the schedule for block transfers between an A/D module and the TAXI optical link. The real BDS description is very lengthy, so the "State Assignments" portion and the reset subroutine have been omitted to condense the description. The reset subroutine is identical to the one in Figure 8-31.

The start node in the flow graph corresponds to the reset state in the state transition graph. The first control step maps to "state_1", and the subsequent six control steps correspond to states "state_2" through "state_7." The interface controller cycles through these seven states to coordinate the transfer of a header word followed by six consecutive A/D words.

The flow graph shown in Figure 8-32 originated from the initial flow graph constructed from the IDL input specification, as shown in Figure 5-12. The iteration node was expanded to six individual concurrent nodes, which in turn generated six control states for the interface controller FSM. In implementation, the sequencing due to iterations can be controlled with a counter. In effect, the generated BDS description is also a description of a counter.

The interface controller FSM implements the state transition graph. For this example, the completion circuit within the timing block uses only the *Go1* input. This simplification is made, because the input information is used in every FSM cycle to compute the next state and outputs as shown in the signal transition graph.



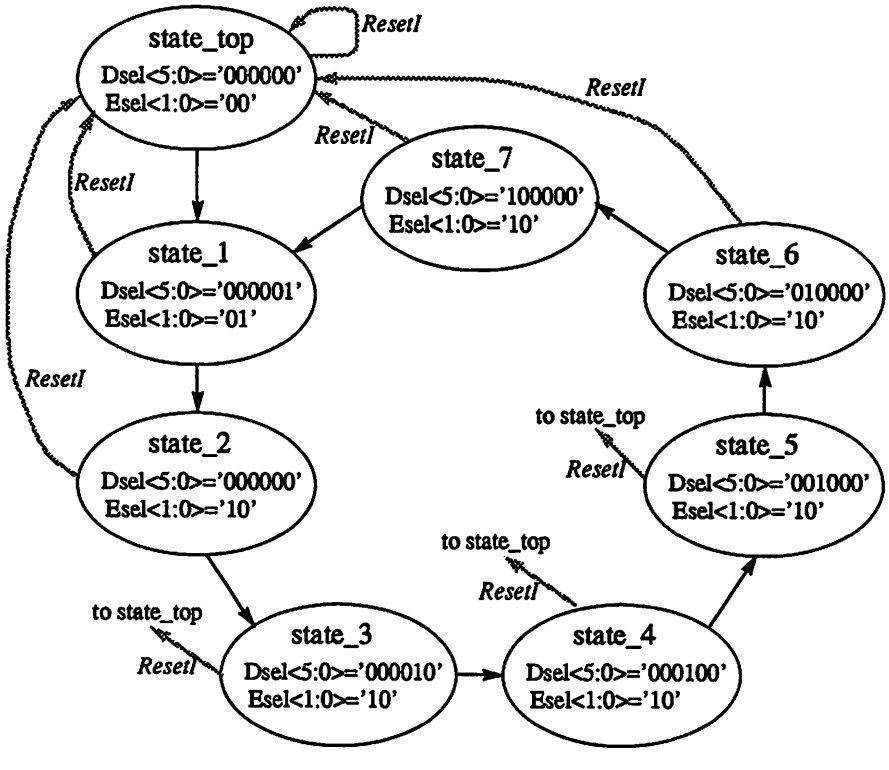


Figure 8-32 : Interface Controller for A/D Module to Optical Link Interface

```
MODEL IctrlFsm
!! Outputs
   nextState<3:0>, zDsel<7:0>, zEsel<1:0> =
!! Inputs
   ResetI<0:0>, currState<3:0>;

! Main routine:
ROUTINE main;
!! State Transition Logic
   nextState = currState; ! default value
   IF reset(ResetI<0:0>)
      THEN nextState=state_top
   ELSE SELECT currState FROM
      [state_top]: BEGIN
         nextState = state_1; END;
      [state_1]: BEGIN
         nextState = state_2; END;
      [state_2]: BEGIN
         nextState = state_3; END;
      [state_3]: BEGIN
         nextState = state_4; END;
      [state_4]: BEGIN
         nextState = state_5; END;
      [state_5]: BEGIN
         nextState = state_6; END;
      [state_6]: BEGIN
         nextState = state_7; END;
      [state_7]: BEGIN
         nextState = state_1; END;
      [OTHERWISE]: BEGIN
         nextState = state_top; END;
   ENDSELECT;

!! Output Logic
   zDsel = 000000#2; zEsel = 00#2; ! default values
   SELECT nextState FROM
      [state_1]: BEGIN
         zDsel = 000001#2; zEsel = 01#2; END;
      [state_2]: BEGIN
         zDsel = 000000#2; zEsel = 10#2; END;
      [state_3]: BEGIN
         zDsel = 000010#2; zEsel = 10#2; END;
      [state_4]: BEGIN
         zDsel = 000100#2; zEsel = 10#2; END;
      [state_5]: BEGIN
         zDsel = 001000#2; zEsel = 10#2; END;
      [state_6]: BEGIN
         zDsel = 010000#2; zEsel = 10#2; END;
      [state_7]: BEGIN
         zDsel = 100000#2; zEsel = 10#2; END;
   ENDSELECT;
ENDROUTINE main;

ENDMODEL IctrlFsm;
```

Figure 8-33 : BDS Description for Interface Controller FSM

8.4 Summary

This chapter has described the mapping of the behavior in the flow graph and event graph to an RTL structure and logic. The synthesis of the datapath and interface controller is *built into* ALOHA, whereas ALOHA uses *external* tools that are already available to synthesize the protocol controller. The final specification of the register-transfer implementation includes a SDL netlist of generic logic units for the datapath and interface controller, consisting of logic 1 wires, logic 0 wires, busses, registers, adders, primitive logic gates, combinational logic units, delay elements and timing circuits. The logic functions for combinational logic units are described in BDS, such as mux or decode functions. The netlist for the protocol controller is accompanied by sequential boolean equations in the EQN format and time constraints in the CLOVER format. From the RTL design abstraction, low-level design takes the logic and timing description into the physical implementation, and the related issues are presented in the next chapter.

CHAPTER 9

LOW-LEVEL DESIGN AND RESULTS

Transforming flow graphs and event graphs into a RTL structure and logic description concludes the high-level design of interface generation, as shown in Figure 9-1.The low-level design phase transforms the RTL design specification through the logical and physical abstraction levels to get the final physical implementation. Tools are already available for both logic synthesis and physical design, and they are utilized instead of developing new ones within ALOHA.

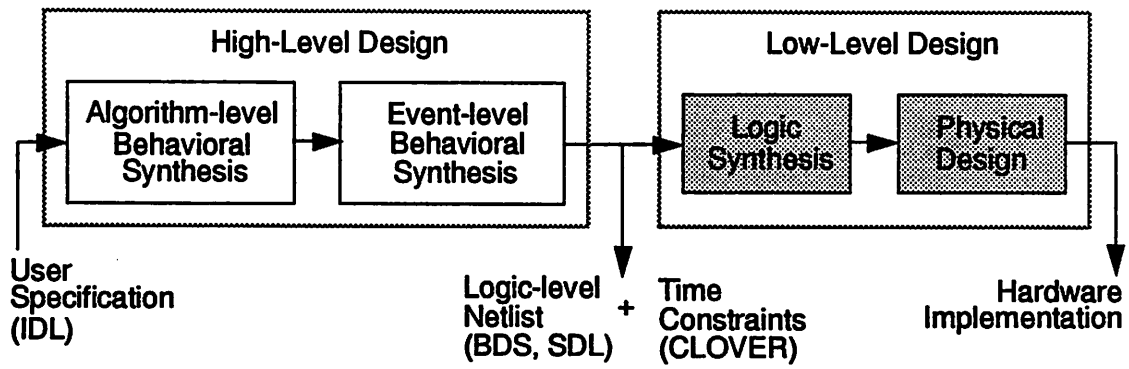


Figure 9-1 : Design Flow for Interface Generation

9.1 Logic Synthesis

The OCTTOOLS collection of logic synthesis tools can perform logic optimization and minimization [Octtool91], and mapping onto a desired technology. The MIS [Rudell88] and SIS [Sentovich88] tools take a two level approach toward logic synthesis. First, the technology independent optimizations and minimizations are made. The minimizations include eliminating redundancies from boolean expressions in the BDS or EQN format. Optimization tools include NOVA which assigns binary vectors to FSM states to minimize the logic of the combinational core [Villa88].

At this level, there is one very important issue specifically related to system interfaces. This is how the boolean equations for the protocol controller are minimized and optimized. Much of the MIS and SIS techniques assume a synchronous design style, which does not suffer from "static/ dynamic hazard" and "races" as the asynchronous design styles does [Eichelberger65] [Torng72]. These two types of problems appear at the logic level and can not be detected at the event graph level. The results from MIS and SIS can be simulated given gate delays and corrected manually to eliminate detected hazards and races. The ASTG package within the SIS targets the asynchronous style, but the resulting optimized and minimized logic must be implemented in the two-level sum-of-products form, which is difficult for many programmable device technologies (PLDs and FPGAs). The boolean logic in the datapath and interface controller can use the results from MIS or SIS directly.

After the technology independent phase, MIS or SIS performs the technology mapping from the logic description to the actual physical hardware cells. This also includes technology dependent optimizations. Suitable technologies for system interfaces include programmable devices, off-the-shelf components specialized for system interconnects, and even ASICs for high-performance and application-specific interfaces. Programmable devices result in quick implementation times and are reconfigurable, which is highly desirable for interface functions like decoding an address map.

Many complex system interfaces today are implemented with a mix of technologies rather than just one. This complicates the technology mapping task, since there are so many types of technologies available. This also leads to the issue of selecting the appropriate technologies for a certain interface application. Programmable devices have traditionally served as decoding or glue logic functions like protocol conversion. However, FPGA architectures have been growing in complexity over the past few years, and they are expected to grow further in the years ahead. As this happens, FPGAs will be suitable to implement main stream computational tasks as well as the glue logic. It is thus becoming possible for one or more FPGAs to implement entire complex system interfaces.

At the technology dependent level, the main issue is how to select the particular technology(s) and hardware cell(s) that a RTL unit or primitive logic gate is mapped to. Recall that the RTL specification only places structural requirements. For example, the edge-triggered registers in the datapath and interface controller allocated by ALOHA represent a generic register that has one N-bit input terminal, one N-bit output terminal and a clock signal. Many times, the technology being mapped onto provides several types of registers which all satisfy these requirements, but may differ in polarity of the latching edge. Another issue is how to implement the delay elements. If a technology does not have such a primitive, then a chain of gates will have to be used.

9.2 Simulation

The VHDL simulation of the IDL input specification only addresses the global communication requirements but not the event-level protocol requirements. For completeness, simulation or verification is needed to check that the technology dependent logic implementation satisfies timing constraints. VHDL [MCC91] has extensive delay modeling capabilities necessary for effective simulation of timing behavior. THOR [THOR] is another simulator that will give sufficient but not excellent timing results, because its delay modeling capabilities are simplistic.

9.3 Physical Design

After technology mapping, the design consists of a netlist (or schematic) of hardware cell instances. Physical design brings the netlist into the physical implementation at the board level or chip level. For instance, the PLDS package [Yu91] partitions and maps a logic netlist onto programmable logic devices using MISII for logic synthesis and translating the result to the manufacturer formats such as those for the Actel FPGA, Altera PLD and Xilinx FPGA. Another software package converts the board level netlist and placement information stored in the OCT database into the input format of a commercial tool [Racal], which does the final multilayer routing and generation of gerber files for actual board fabrication.

9.4 Summary of Results

A first version of the ALOHA interface generator has been implemented in the C programming language, consisting of approximately 20,000 lines of source code developed over 18-man months. The organization of the software implementation is elaborated in Appendix C. The IDL specification, library of protocols, and high-level synthesis methods have been applied to several examples representative of a wide class of applications. The major ones were used to illustrate the specification and synthesis concepts described in the previous chapters, and examples were actually simplified to keep the example sizes reasonable within the space of a chapter.

Table 9-1 summarizes the design results from the examples. The first, called *"vme2sram"*, was one of the motivating examples for this research. It is a VMEbus interface providing read and write access from the VMEbus master to a slave static memory module. The second interface was used in a image processing system [Chandrakasan91], and the interface is a similar but more complex version of the first example. It was implemented on a printed circuit board and tested for successful functionality and performance, providing early confidence in the presented design methods. The last three examples were from a robot controller and robot peripheral subsystems

[Srivastava92]. The synthesized results have also been simulated.

All these examples demonstrate that from a compact and high-level input description, ALOHA automatically generates the corresponding flow graphs, event graphs, and finally an interface logic

Table 9-1 : Interface Examples and Results

Design Name	Communication Description and I/O Protocols Description: # data bits, # address bits, # control signals / # events, # precedences, # timing constraints	Results			
		Speed access rate	Datapath # functional units	Protocol Ctrl[1] # event graphs/ # states/ #p-terms/ # literals	Interface Ctrl # FSM states
vme2sram pipelined version	**Provides pipelined read and write access from the VMEbus host to a static RAM module.** VMEbus read protocol: 16,21,6 / 21,20,17 VMEbus write protocol: 16,21,6 / 21,22,17 SRAM read protocol: 16,10,3 / 10,14,8 SRAM write protocol: 16,10,3 / 10,15,7	4 Mhz	1 read data latch 1 write data latch 1 address latch address decoder	2/4/10/13	2
sequential version	**Non-pipelined read and write access from the VMEbus host to a static RAM module.**		1 address latch address decoder	1/2/2/6	0
vme2image	**Read and write communications from a VMEbus host with memory-mapped ASICs on an 9U VME board for real-time image processing.** VMEbus read protocol: 16,21,6 / 21,20,17 VMEbus write protocol: 16,21,6 / 21,22,17 ASIC read protocol: 8,0,3 / 8,12,5 ASIC write protocol: 12,0,3 / 8,10,5	3 Mhz	1 tri-state buffer for data 1 address latch address decoder	2/8/10/18	2
adc2taxi	**Block transfers from a bank of AD7870 A/D converters to an Am7968 TAXI fiber optic transmitter. Part of a 14"x16" board for robot arm sensing.** AD7870 protocol: 10,0,3/ 8,12,8 Am7968 protocol:12,0,2 / 8,11,7	10 Mhz	1 data bus status & error handling logic address decoder	2/8/15/10	13
tms2taxi	**Time multiplexed communication of data and address packets from the TMS320C30 DSP to an Am7968 TAXI transmitter. Part of a 14"x16" board for robot control.** TMS320C30 DSP protocol: 4,5,4 / 10,13,6 Am7968 protocol:12,0,2 / 8,11,7	10 Mhz	2 multiplexers 1 data latch	2/8/9/7	3
taxi2dac	**Time demultiplexed packet communications from an Am7969 TAXI fiber optic receiver module to a bank of DAC811 D/A converters. Part of a 14"x16" board for robot arm sensing.** Am7969 protocol: 12,0,4 / 6,6,3 DAC811 protocol:10,0,6 / 8,15,11	6 Mhz	1 data bus 1 data latch address decoder	2/8/10/7	3

[1]for sum-of-products form

implementation. They also show the two main advantages of the described design methodology. First, the design methodology and CAD support relieves the user from gathering, learning and dealing with communication and synchronization details (flow graph, event graph, timing and structure). The non-expert can produce functional interface between various modules, since the expertise is moved into the synthesis system.

The second advantage is the savings in system hardware design time offered by automating interface generation. Table 9-2 shows the CPU times on a SPARCstation2 for each of the design examples. The second column gives the total ALOHA run-times for synthesizing the interface netlist (SDL), logic (BDS), event graphs (ASTG) and time constraints (CLOVER) from the user-provided IDL specification. The other columns show the CPU time for each of the major transformation in the synthesis process, not including translation from the IDL specification. Column 3 shows the cumulative time for flow graph clustering, scheduling and allocation, as described in Chapter 6; column 4 gives the time for generating event graphs from the flow graph and module library, as described in chapter 7; finally, the last column indicates the time for generating an interface structure implementation from the flow graph as described in Chapter 8.

Table 9-2 : CPU Times for Interface Examples

Design Name	ALOHA Synthesis	Synthesis from Flow Graph	Generate Event Graphs	Generate RTL Structure
vme2sram	.9s	.6s	.3s	.4s
vme2image	.9s	.6s	.3s	.4s
adc2taxi	1.0s	1.1s	.1s	.5s
tms2taxi	.7s	.5s	.1s	.3s
taxi2dac	.6s	.4s	.2s	.2s

CHAPTER 10

CONCLUSIONS AND FUTURE DIRECTIONS

The work presented provides the design methodology and computer-aided techniques for interconnecting and synchronizing communicating modules into a system. A part of the SIERA system design environment, the ALOHA interface generator achieves its primary goal of reducing the effort required from a designer to integrate hardware. This allows the design focus to be placed on the system application and on the architectural issues.

Summarized in Figure 1-7 (Chapter 1), the design methodology presented takes a system-level approach. It starts with a behavioral specification of an interface module and generates an asynchronous logic-level implementation. The high-level input specification, a module library and behavioral synthesis methods can handle communication between multiple source and destination modules and a mix of arbitrary I/O protocols and time constraints. Synthesis transforms the design specification through two levels of abstraction to generate the interface logic. ALOHA covers the technology independent synthesis phase, while pre-existing and mature design tools are pulled in to generate the final gate-level and physical implementation.

10.1 Conclusions and Contributions

This thesis has demonstrated that a critical issue in system level hardware integration is the generation of interfaces between communicating modules. Typically, the modules use various technologies and have incompatible I/O protocols; the interface must resolve these differences while transferring information. The desire for a high-level design abstraction in the face of low-level protocol details is the key difficulty in interfacing system modules and thus presents a challenge in interface specification and synthesis. The important lesson learned from attacking this problem is that the solution can use concepts extended from VLSI chip design while it requires new design techniques that address issues specific to board-level interfacing.

Chip level design concepts that continue to be extremely useful at the system level are hierarchy, module generators and libraries. Applying hierarchy to system design, system modules are composed from finer grained modules in which the primitives are chip components from various technologies. Module generators produce system hardware (and even software) at each level in the hierarchy. The interface is viewed as just another hardware module, but dedicated to communication and synchronization tasks and is produced by the special generator ALOHA. Interfaces are used to integrate hardware into a higher level module that can be placed into a module library. Like chip level cell libraries, the module library contains information about each module that the generators can use. The design methodology presented applies the library method to capture protocol information used by ALOHA.

The new ideas introduced by this work for the specification and synthesis of interfaces take advantage of the above concepts. The key contributions of this work are:

- A high-level interface specification language for describing inter-module communication behavior. The IDL language is concise, expressive and easy to use. The specification is made independent of the module technologies through a policy for capturing I/O protocols into a module library.

- New synthesis techniques for transforming a high-level behavioral specification of the interface into the register-transfer level and event graph implementations of the interface datapath and controllers. The techniques are based on the flow graph and event graph representations, and also an interface template. Unlike previous work related to interface synthesis, these techniques effectively support a wide range of interface applications from complex direct-memory-access controllers to simple protocol converters.
- A module generator, ALOHA, that automatically implements the above specification and synthesis techniques and that is integrated into a computer-aided-design environment, SIERA, for system design. ALOHA covers design from the system level of abstraction to the timing and synchronization level and then to the boolean logic and structural level. In contrast, previous tools related to interfaces focused on only one level of abstraction and concentrated on synthesizing just the datapath plus interface controller or only synthesizing the protocol control logic.

Overall, the design methodology and ALOHA synthesis tool ***raises*** the interface abstraction to the system level compared to previous methods. As shown in Figure 1-7, the designer starts with knowledge of the information I/O pins and the global inter-module communication requirements. Just this knowledge is sufficient for specifying an interface module. The designer does not need to know the internal interface structure or behavior. The details related to protocol events and time constraints local to a module port are already captured in the library. Without the module library, the designer would have to gather the information from data sheets, become familiar with all the protocols and describe an enormous amount of events and timing constraints as part of the input specification. To demonstrate this, the input IDL descriptions for the standard examples were shown in Section 4.4 (Chapter 4), and the protocol descriptions were shown in Section 7.4 (Chapter 7). With the support of abstraction, the designer only deals with the inter-module communications, and is relieved from dealing with the synchronization and timing requirements.

Comparing the input descriptions of Section 4.4 to the datapaths of Section 8.1.3, the protocol control logic and time constraints shown in Section 8.2.4, and the central interface controllers of Section 8.3.3, the design method presented hides low-level design details from the designer and significantly reduces the design effort to go from behavioral specification to logical structure. Raising the design abstraction to the system level allows a non-expert or system designer to produce interfaces and easily integrate hardware into the system, as demonstrated by the examples.

10.2 Future Directions in System Integration

The past chapters presented a top-down treatment of interface generation. The abstraction levels ranged from system architecture, to register transfer, to boolean logic, to IC components and finally to board layout. Open areas for enhancements to ALOHA and long-term research questions stemming from this work are presented here for all abstraction levels.

10.2.1 ALOHA Enhancements

This section discusses enhancements to the basic design methodology and interface generation capabilities. Some were discussed in detail in the past chapters, and some currently use a manual solution. All these enhancements could be immediate upgrades to ALOHA.

- Automatically linking the system hardware architecture generated in SIERA to the ALOHA synthesis software. Currently, the IDL input specification is manually defined by the system designer. Ideally, it should be generated from the architecture of communication hardware modules, which SIERA produces a VHDL simulation model for [Srivastava92].
- Specifying communication constraints and arbitration behavior as part of the inter-module communication requirements. Constraints include communication throughput, channel buffer depth and even power. Arbitration in its pure form is non-deterministic, but uses a deterministic algorithm in practice. The current IDL language implementation can express buffer depth, but the other features will need new high-level constructs. Communication throughput and arbitration involve at least one source and one destination port, so their constructs will need to specify these parameters.
- Automatically generating the VHDL simulation model from the IDL input specification. Currently, the model is manually constructed. The simulation model is used to validate the inter-module communication specification. Also, simulation can be effectively applied toward communication performance evaluation and adjusting system communication parameters, such as arbitration algorithm and buffer depth. Section 4.5 discusses VHDL model generation in detail.
- Extending the module library to include I/O electrical characteristics and parameterized I/O protocols. The first allows synthesis to meet electrical constraints. The second is useful for designing application-specific modules where the I/O protocol is not pre-defined. Both of these are further discussed in Section 3.5.
- Graphical editor for event graphs. The event graph is currently entered into the library using the AFL ascii text format (see Appendix A). A graphical entry system would allow the user to enter

nodes, precedence edges, and properties on both those objects such as signal name, transition value and time constraints. The user should also be able to select nodes, edges and properties to modify them. The event graph consistency checks discussed in Section 4.5 can be incorporated into the editor system, and the editor should also outputs the AFL description to store the protocol into the module library. Another alternative is a timing diagram editor and a translator which transforms the timing diagram into the event graph equivalent and AFL description for synthesis. An example of a timing diagram editor is Waves [Borriello88b].

- Automatically generating the OEsim model for event graphs. The model supports both validating the library event graphs and interlocked event graphs. Model generation details are presented in Section 3.4
- Enhancements to flow graph synthesis techniques. This includes checks for all types of inconsistencies, some discussed in Section 5.4. Also, support of the "WHILE" control construct, unrestricted control flow, user-defined internal states and user-defined buffers increase the basic synthesis capabilities. Another important feature is estimating cost and performance from the flow graph and underlying register-transfer units. These enhancements would be useful for effective synthesis of complex interface modules such as DMA controllers and I/O processors.
- Incorporating other control logic synthesis tools. As discussed in Section 8.2.1, besides ASTG there are other existing techniques for generating the protocol control boolean logic from the event graphs. An example is Janus for mixed asynchronous/synchronous environments.

10.2.2 Long-term Directions

Although this thesis has presented a detailed and top-down methodology for designing interface modules, there are other important and related problems that are open for long-range research directions. This section generally describes the issues, some of which currently use manual techniques.

- An important aspect of system specification is designing the communication channel. The research presented in this thesis assumes that the inter-module communication behavior is given. Naturally, an extended research topic is determining *what* the inter-module communication should be, given that the system is modeled by concurrent and communicating processes (implemented by a hardware or software module). A major design issue is allocating communication channels among interacting processes; in other words, whether several processes should share one physical channel or multiple (but not necessarily the same number) of channels. Examples are multiplexing large messages across a smaller width bus, and, at the extreme, transmitting parallel data on a serial link. When many master modules share an assigned channel, this leads to the problem of either scheduling individual transfers on the channels a priori to avoid contention (static), or allowing the modules to arbitrate over the channel when they need

to communicate (dynamic).

Other related design related issues include designing an efficient and fair arbitration algorithm, and determining the optimal channel buffer depth. Another problem is how to achieve high-level communication synchronization. Choices include polling or an interrupt mechanism. The solution to all the above interrelated problems depends on various factors such as communication throughput and channel utilization. Some of these issues have been explored by the research in system architecture design for SIERA [Srivastava92], which provides a starting point.

- Another system level issue related to the above topic is the design and verification of I/O protocols for application-specific modules and busses. For basic communications, this usually consists of the information transfer protocol (or message passing protocol). For modules and busses that handle an array of communication function, this includes the transfer protocol, interrupt protocol and arbitration protocol. Design factors to consider are communication performance, global versus local communications, diversity of ports to be interconnected, and the application. Protocol styles include asynchronous and synchronous timing, and hybrids between the two.

- Compared to the current techniques in ALOHA, sophisticated scheduling and allocation methods for complex data and control flow behavior present an open area in behavioral synthesis research. This is especially important to automatically generate complex modules such as controllers and I/O processors. In fact, from a high-level perspective, the presented synthesis techniques should be combined with new and existing methods to generate entire subsystems that implement an overall communication function. Examples are a network control node or an integrated image decompression and digital-to-analog conversion module. The modules are formed from several submodules. Subsequently, the behavioral synthesis task for such subsystems is difficult since it must be partitioned among different hardware modules. The hardware is chosen from many available technologies such as various FPGAs, TTL drivers and various IC components for system interconnection.

- Logic synthesis techniques for mixed synchronous-asynchronous environments. Currently, logic synthesis for logic minimization and synchronous design style is mature. However, in the course of the research presented in this thesis, a major bottleneck in the generation path was synthesizing asynchronous control logic from the interlocked event graph to satisfy time constraints and verifying the logic for time constraints. The basic strategy is: first check and correct the input event graph for deadlock and hazard conditions, and then, given a technology (gate library), produce a minimized gate-level implementation that satisfies the time constraints. Performance constraints are considered a part of the time constraints. In the first phase, an open research topic is methods that specify exactly *how* to modify the event graph to eliminate the problem conditions, and past STG research provides some starting points [Chu87a][Meng89]. The second phase can use past event graph research as a starting point [Borriello92]. Research in both phases is becoming more and more important as a variety of programmable hardware

(PLDs and FPGAs) are made available for both control logic and datapath functions.

- At the physical level, a complete treatment of the system integration problem includes the design of physical interconnects from the chip-level to the package and to the board-level. Currently, technology is scaling down and moving toward high-density packaging such as surface mount. This allows dense systems to run at very high speeds while the global interconnects remain as long as they were with previous technologies. The physical interconnect suffers from transmission line impairments such as crosstalk and reflections. Over the past years, this has become an active area of research and open areas include design rules for minimizing transmission line problems, layout extraction for printed-circuit boards, quick estimation based on the extracted layout, as well as automatically generating models for simulation and verification of critical nets.

The previous chapters have covered interface design methods from the system architecture level of abstraction to the boolean logic level of abstraction. The open areas of research, described in this chapter, range from system specification down to board layout. The overall challenge is a comprehensive solution for the system integration problem which requires and combines knowledge and techniques from many disciplines within electrical engineering and computer science.

BIBLIOGRAPHY

1. [Amon91a] T. Amon and G. Borriello. *OEsim: A Simulator for Timing Behavior.* Proceedings of the 28th ACM/IEEE DAC, June 1991.
2. [Amon91b] T. Amon and G. Borriello. *Sizing Synchronization Queues: A Case Study in Higher Level Synthesis.* Proceedings of the 28th ACM/IEEE DAC, June 1991.
3. [Barbacci81] M.R. Barbacci. *Instruction Set Processor Specifications (ISPS): The Notation and Its Applications.* IEEE Transactions on Computers, Vol. C-30, No. 1, January 1981.
4. [Berkel91] K. van Berkel, J. Kessels, M Roncken, R. Saeijs and F. Schalij. *The VLSI-programming language Tangram and its translation into handshake circuits.* Proceedings of EDAC, March 1991.
5. [Bochman83] G.V. Bochman and M. Raynal. *Structured Specification of Communicating Systems.* IEEE Transactions on Computers, Vol. C-32, No. 2, February 1983.
6. [Borriello87] G. Borriello and R.H. Katz. *Synthesis and Optimization of Interface Transducer Logic.* Proceedings of IEEE ICCAD, November 1987.
7. [Borriello88a] G. Borriello. *Combining Event and Data-Flow Graphs in Behavioral Synthesis.* Proceedings of IEEE ICCAD, November 1988.
8. [Borriello88b] G. Borriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis.* Ph. D. Thesis, Report No. UCB/CSD 88/430, EECS Department, U.C. Berkeley, May 1988.
9. [Camposano86] R. Camposano and A. Kunzmann. *Considering Timing Constraints in Synthesis from a Behavioral Description.* Proceedings of ICCD, October 1986.
10. [Chandrakasan91] A.P. Chandrakasan. *Design of a Real-Time Flexible Image Processing System.* M.S. Report, Report No. UCB/ERL M91/27, EECS Department, U.C. Berkeley, April 1991.
11. [Chu86] T.A. Chu. *On Models for Designing VLSI Asynchronous Digital Systems.* Integration - the VLSI Journal, Vol. 4, August 1986.
12. [Chu87a] T.A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* Proceedings of IEEE ICCD, October 1987.
13. [Chu87b] T.A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications.* Ph. D. Thesis, MIT/LCS/TR-393, Laboratory for Computer Science, MIT, June 1987.
14. [Chu89] C.M. Chu, M. Potkonjak, M. Thaler and J. Rabaey. *HYPER: An Interactive Synthesis Environ-*

ment for Real Time Applications. Proceedings of IEEE ICCD 1989, October 1989.

15. [Chuang73] H. Chuang and S. Das. *Synthesis of Multiple-Input Change Asynchronous Machines Using Controlled Excitation and Flip-Flops.* IEEE Transactions on Computers, Vol. C-22, No. 12, December 1973.
16. [Clements87] A. Clements. *Microprocessor Systems Design: 68000 hardware, Software, and Interfacing.* PWS-KENT Publishing Company, 1987.
17. [Cloutier90] R. Cloutier and D. Thomas. *The Combination of Scheduling, Allocation and Mapping in a Single Algorithm.* Proceedings of the 27th ACM/IEEE DAC, June 1990.
18. [Davis82] A.L. Davis and R.M. Keller. *Data Flow Program Graphs.* Computer, February 1982.
19. [DeMicheli88] G. DeMicheli and D.C. Ku. *HERCULES: A System for High-Level Synthesis.* Proceedings of the 25th ACM/IEEE DAC, June 1988.
20. [Dijkstra75] E.W. Dijkstra. *Guarded Commands, Non-determinacy and Formal Derivation of Programs.* Communications of the ACM, Vol. 18, No.8, August 1975.
21. [Doukas91] D. Doukas and A.S. LaPaugh. *CLOVER: A Timing Constraints Verification System.* Proceedings of the 28th ACM/IEEE DAC, June 1991.
22. [Eichelberger65] E.B. Eichelberger. *Hazard Detection in Combinational and Sequential Switching Circuits.* IBM Journal, March 1965.
23. [Futurebusa] P.L. Borrill. *The Futurebus Project.* Proceedings of COMPCON, Spring 1984.
24. [Futurebusb] *Futurebus Specifications.* P896.1. Public Comment Draft 7.1. IEEE Computer Society.
25. [Girczyc84] E.F. Girczyc. *Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Descriptions.* Ph. D. Thesis, Carleton University, Ottowa, Canada, July 1984.
26. [Guibaly89] F. E. Guibaly. *Design and Analysis of Arbitration Protocols.* IEEE Transactions on Computers, Vol. 38, No. 2, February 1989.
27. [Harrison86] D.S. Harrison, P. Moore, R.L. Spickelmier and A.R. Newton. *Data Management and Graphics Editing in the Berkeley Design Environment.* Proceedings of IEEE ICCAD, November 1986.
28. [Hartenstein87] R.W. Hartenstein. *Hardware Description Languages, The Classification of Hardware Description Languages.* Chapter 2, Elsevier Science Publishers B.V. (North-Holland), 1987.
29. [Har'El88] Z. Har'El and R.P. Kurshan. *Software for Analysis of Coordination.* Proceedings of the International Conference on System Science, 1988.
30. [Hayati89] S. Hayati and A.C. Parker. *Automatic Production of Controller Specifications From Control and Timing Behavioral Descriptions.* Proceedings of the 26th ACM/IEEE DAC, June 1989.
31. [Hayes81] A.B. Hayes. *Stored State Asynchronous Sequential Circuits.* IEEE Transactions on Computers, Vol. c-30, No. 8, August 1981.
32. [Hilfinger85] P. Hilfinger. SILAGE, *A High Level Language and Silicon Compiler for Digital Signal Processing.* Proceedings of IEEE CICC, May 1985.
33. [Hill78] F.J. Hill and G.R. Peterson. *Digital Systems: Hardware Organization and Design.* John Wiley & Sons, 1978.
34. [Jones90] G. Jones. *ASYNC Man Page.* EECS Department, U. C. Berkeley, 1990.
35. [Kleeman87] L. Kleeman and A. Cantoni. *Metastable Behavior in Digital Systems.* IEEE Design & Test of Computers, December 1987.
36. [Knapp84] D. Knapp, J. Granacki and A.C. Parker. *An Expert Synthesis System.* Proceedings of IEEE ICCAD, September 1984.
37. [Kramer90] H. Kramer and W. Rosenstiel. *System Synthesis using Behavioral Descriptions.* Proceedings of EDAC, March 1990.
38. [Lavagno91] L. Lavagno, K. Keutzer and A. Sangiovanni-Vincentelli. *Algorithms for Synthesis of Hazard-free Asynchronous Circuits.* Proceedings of the 28th ACM/IEEE DAC, June 1991.
39. [Lipsett89] R. Lipsett, C.F. Schaefer and C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1989.
40. [Martin86] A.J. Martin. *Compiling Communicating Processes Into Delay-Insensitive VLSI Circuits.* Distributed Computing, No. 1, 1986.
41. [MCC91] Microelectronics and Computer Technology Corporation. *VHDL System Release 3.2.2 Docu-*

mentation, 1990.

42. [McFarland78] M.C. McFarland. *The VT: A Database for Automated Digital Design.* Design Research Center, DRC-01-4-80, Carnegie-Mellon University, December 1978.
43. [McFarland86] M.C. McFarland. *Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions.* Proceedings of the 23rd ACM/IEEE DAC, June 1986.
44. [McFarland90] M.C. McFarland, A.C. Parker and R. Camposano. *The High-Level Synthesis of Digital Systems.* Proceedings of the IEEE, Vol. 78, No. 2, February 1990.
45. [McWilliams80] T. McWilliams. *Verification of Timing Constraints on Large Digital Systems.* Ph. D. Thesis, Lawrence Livermore Laboratory, May 1980.
46. [Meng88] T.H.Y. Meng. *Asynchronous Design for Digital Signal Processing Architectures.* Ph. D. Thesis, EECS Department, U.C. Berkeley, 1988.
47. [Meng89] T.H.Y. Meng, R.W. Brodersen and D.G. Messerschmitt. *Automatic Synthesis of Asynchronous Circuits from High-Level Specifications.* IEEE Transactions on CAD, Vol. 8, No. 11, November 1989.
48. [Moon91] C.W. Moon, P.R. Stephan and R. K. Brayton. *Synthesis of Hazard-free Asynchronous Circuits from Graphical Specifications.* Proceedings of IEEE ICCAD, November 91.
49. [Moon92] C.W. Moon, P.R. Stephan and R. K. Brayton. *Synthesis and Verification of Asynchronous Circuits from Graphical Specifications.* To appear in Journal of VLSI Signal Processing. Kluwer Academic Publishers, 1992.
50. [Multibus] Intel Corporation. *Intel Multibus Specification.* Order No. 9800683-04, 1982.
51. [Nestor86] J.A. Nestor and D.E. Thomas. *Behavioral Synthesis with Interfaces.* Proceedings of IEEE ICCAD, 1986.
52. [Nowick91] S.M. Nowick and D.L. Dill. *Synthesis of Asynchronous State Machines Using a Local Clock.* Proceedings of IEEE ICCD, October 1991.
53. [NuBus] P1196 Working Group of the Microprocessor Standards Committee. *NuBus - a Simple 32-Bit Backplane Bus.* Draft 2.0, December 1986.
54. [Octtools91] Berkeley CAD Group. *OCTTOOLS Reference Manuals.* EECS Department, U.C. Berkeley, 1991.
55. [Oppenheim89] A.V. Oppenheim and R.W. Schafer. *Discrete-Time Signal Processing.* Prentice Hall, 1989.
56. [Orailaglu86] A. Orailaglu and D.D. Gajski. *Flow Graph Representation.* Proceedings of the 23rd ACM/IEEE DAC, June 1986.
57. [Pangrle87] B.M. Pangrle and D.D. Gajski. *Design Tools for Intelligent Silicon Compilation.* IEEE Transactions on CAD, Vol. 6, No. 6, November 1987.
58. [Parker81] A.C. Parker and J. Wallace. *SLIDE: An I/O Hardware Descriptive Language.* IEEE Transactions on Computers, Vol. C-30, No. 6, June 1981.
59. [Parker86] A.C. Parker, J. Pizarro and M. Mlinar. *MAHA: A Program for Datapath Synthesis.* Proceedings of the 23rd ACM/IEEE DAC, June 1986.
60. [Paulin89] P.G. Paulin and J.P. Knight. *Force-Directed Scheduling for the Behavioral Synthesis of ASIC's.* IEEE Transactions on CAD, Vol. 8, No. 6, June 1989.
61. [Rabaey90] J. Rabaey and P. Hoang. *HYPER Flowgraph Policy.* EECS Department, U.C. Berkeley, March 1990.
62. [Racal] Racal-Redac Inc. *VISULA-PLUS User's Guide.*
63. [Ramamoorthy80] C.V. Ramamoorthy and G.S. Ho. *Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets.* IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, September 1980.
64. [Rey74] C.A. Rey and J. Vaucher. *Self-Synchronized Asynchronous Sequential Machines.* IEEE Transactions on Computers, December 1974.
65. [Richards] B. Richards. *SDL Language Syntax.* LAGER-IV Manuals, Vol. 2, EECS Department, U.C. Berkeley.
66. [Rudell88] R. Rudell, A. Wang and R. Spickelmier. *MIS User's Manual.* Oct Tools Distribution, EECS Department, U.C. Berkeley, March 1988.

67. [Segal88] R.B. Segal. *BDSYN User's Manual.* Oct Tools Distribution, EECS Department, U.C. Berkeley, March 1988.
68. [Sentovich88] E. Sentovich and K.J. Singh. *SIS Reference Manual Pages.* EECS Department, U.C. Berkeley, March 1988.
69. [Shung91] C. S. Shung, R. Jain, K. Rimey, E. Wang, M. B. Srivastava, E. Lettang, S. K. Azim, L. Thon, P. N. Hilfinger, J. M. Rabaey, and R. W. Brodersen. *An Integrated CAD System for Algorithm-Specific IC Design.* IEEE Transactions on CAD, April 1991.
70. [Srivastava91a] M. B. Srivastava, and R. W. Brodersen. *Rapid-Prototyping of Hardware and Software in a Unified Framework.* Proceedings of IEEE ICCAD, November 1991.
71. [Srivastava91b] M.B. Srivastava. *Private Communications.* EECS Department, U.C. Berkeley, 1991.
72. [Srivastava92] M.B. Srivastava. *Rapid-Prototyping of Hardware and Software in a Unified Framework.* Ph. D. Thesis. EECS Department, U.C. Berkeley, 1992.
73. [Stone82] H. S. Stone. *Microcomputer Interfacing.* Addison-Wesley Publishing Company, 1982.
74. [Sun91] J. S. Sun, M. B. Srivastava and R. W. Brodersen. *SIERA: A CAD Environment for Real-TIme Systems.* 3rd IEEE/ACM Physical Design Workshop, May 1991.
75. [TAXIxmt] Advanced Micro Devices. *Am7968/Am7969 TAXIchip Integrated Circuits Specifications.*
76. [Thomas83] D.E. Thomas, C.Y. Hitchcock III, T.J. Kowalski, J.V. Rajan and R.A. Walker. *Methods of Automatic Data Path Synthesis.* IEEE Computer, December 1983.
77. [Thomas90] D.E. Thomas, E.D. Lagnese, R.A. Walker, J.A. Nestor, J.V. Rajan and R.L. Blackburn. *Algorithmic and Register Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, 1990.
78. [Thon89] L. Thon, K. Rimey, B. Richards and L. Svensson. *From C to Silicon with LagerIV.* IEEE/ACM Physical Design Workshop, May 1989.
79. [THOR] *THOR Simulator Reference Manual.* Electrical Engineering Department, Stanford University.
80. [Thurber72] K.J. Thurber, E.D. Jensen, L.A. Jack, L.L. Kinney, P.C. Patton and L.C. Anderson. *A Systematic Approach to the Design of Digital Bussing Structures.* AFIPS, Proceedings of FJCC, 1972.
81. [Torng72] H.C. Torng. *Switching Circuits: Theory and Logic Design.* Addison-Wesley Publishing Company, 1972.
82. [Trickey87] H. Trickey. *Flamel: A High-Level Hardware Compiler.* IEEE Transactions on CAD. March 1987.
83. [TMS320] Texas Instruments. *Third Generation TMS320 User's Guide.*
84. [Treleaven82] P.C. Treleaven. T.P. Hopkins and P.W. Rautenbach. *Combining Data Flow and Control Flow Computing.* Computer Journal, Vol. 25, No. 2, 1982.
85. [Vanbekbergen90] P. Vanbekbergen, F. Catthoor, G. Goossens and H. DeMan. *Optimized Synthesis of Asynchronous Control Circuits from Graph-theoretic Specifications.* Proceedings of the IEEE ICCAD, November 1990.
86. [Villa88] T. Villa. *NOVA User's Manual.* Oct Tools Distribution, EECS Department, U.C. Berkeley, March 1988.
87. [VMEbus] VITA. *VMEbus Specification Manual.* PRINTEX Publishing, 1985.
88. [Walker91] R.A. walker and R. Camposano. *A Survey of High-Level Synthesis Systems.* Kluwer Academic Publishers, 1991.
89. [Whitcomb92] G. Whitcomb. *BLIS Reference Manual.* EECS Department, University of California, Berkeley, 1992.
90. [Yenersoy79] O. Yenersoy. *Synthesis of Asynchronous Machines Using Mixed-Operation Mode.* IEEE Transactions on Computers, Vol. C-28, No. 4, April 1979.
91. [Yu91] R.K. Yu. *PLDS: Prototyping in Lager Using Decomposition and Synthesis.* Memorandum No. UCB/ERL M91/53, U.C. Berkeley, May 1991.

APPENDIX A: EVENT GRAPH FORMAT DESCRIPTION

A digital component or module has input and output terminals which convey signals. Some of these signals carry the information of interest, such as data and address. Other signals, such as control signals, synchronize the transmission of information. For proper transmission, events (or signal transitions) on the information and control signals are sequenced according to a signaling convention, or *I/O protocol*. The protocol may also define data formatting and representation rules.

The protocol is characterized by events and the precedences between events, including timing relationships, and *event graphs* are used to model this behavior. Nodes in the graph represent events, and the directed edges represent the precedences. Weights on the edges represent timing relationships. A module may use one or a set of protocols. For example, memory components use different sequences of events for the read and the write cycle. This requires two event graphs to describe memory access, one for the read protocol and the other for the write protocol.

Figure A shows the block diagram for a static memory, and Figure B shows the timing diagram for the protocol exercised during a write cycle. The equivalent event graph is in Figure C. Data,

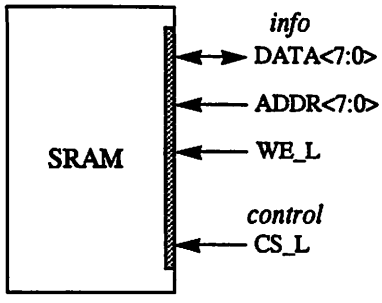


Fig. A

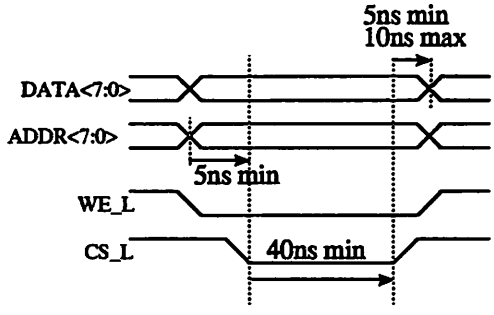


Fig. B

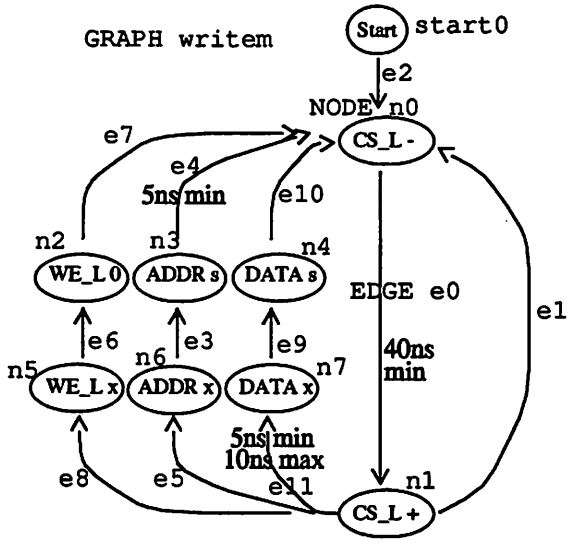


Fig. C

Table 1: Transition Value System

Value	Meaning
r	rising(+), signal changing from low to high
f	falling(-), signal changing from high to low
s	stable, signal changing to stable at some bit value
bit vector	signal is changing to stable at a specific binary value
x	unknown, signal value is unknown or don't care
z	high-impedance, signal is changing to high-Z state

address and write enable are considered information type of signals, while chip select is regarded as a control signal. Each node in the graph has a corresponding transition in the timing diagram. The edges show the basic precedence of events: data, address and write enable signals stabilize, then an active-low chip select pulse occurs, followed by data, address and write enable signals becoming unknown, and finally cycle is allowed to repeat. The 5ns minimum address set time (relative to chip select falling), the 40ns minimum chip select strobe width, and 5ns minimum/10ns maximum data hold time (relative to chip select rising) are all timing constraints expressed as weights on the appropriate edge. The start node marks the first event of the cycle.

In the event graph model used by ALOHA, signal events are allowed to take on transition values listed in Table 1. Control type signals use the rise, fall or high-impedance values; information type signals use the stable, unknown, binary and high-impedance values. The model also requires that events occurring on control type signals must be strongly connected, and there must not be any redundant edges. The precedences between control events thus forms the skeleton of the event graph, and makes it cyclic. In Figure C, node n0, edge e0, node n1 and edge e1 form the strongly connected skeleton of the event graph.

The following describes the policy for the textual specification of event graphs for use in the ALOHA synthesis system. The format is based on the AFL language [Rabaey90]. The AFL specificaiton for the event graph in Figure C is shown at the end of this appendix. It should be used when reading the policy.

Policy for the Event Graph Format

Once again, an event graph is a composition of nodes and edges. A node represents a single event or a collection of simultaneously occurring events. In the second case, the node is actually a subgraph which in turn contains nodes representing the events. AFL is a language from the HYPER synthesis system that describes directed graphs as a list of nodes, edges and subgraphs. Presented below is the policy for using AFL to describe event graphs for ALOHA.

Event Graph Description Syntax

The format uses rounded brackets as delimiters, such that a lisp-like syntax is obtained(only in resemblance). The basic statement has the following format:

statement := (keyword definition)

where

definition := atom || statement || list
atom := integer || string
list := (definition*)

A string is either of the following format:

string := [A-Za-z][_A-Za-z0-9]*
or any double quoted entity "..."

The format supports three basic structures: the graph, the node and the edge. An event graph is defined as a collection of graphs:

event graph := graph-definition graph-definition ...

where the main event graph must be the first graph-definition, followed by the subgraphs.

Since the AFL-parser first passes the input description through the C-preprocessor *cpp*, the language recognizes C-style comments (surrounded by "/*" and "*/") and the C-directives #include and #define. The parser is also case sensitive to string atoms.

Graph-Structure

A graph is a named entity, which consists of a number of edges and nodes.

graph-definition :=
(graph
(name name-string)
(class "MODULE")
(model ((model_name model-string)))
(arguments ((port port-string) (timeunit timeunit-string)))
(attributes ((mode validated)))*
(nodelist node-definitions)
(controllist edge-definitions)

)

For the main event graph, **name-string** is a name assigned to the protocol being described, such as "read" or "write". For subgraphs, name-string is "multievent#", where # is a unique integer tag.

The class of an event graph or subgraph is always, "MODULE".

For the main event graph, the **model-string** specifies whether the module port using the described protocol is a master or slave port. Master ports initiate data transactions while slave port can only respond. model-string is either "masterp" or "slavep". For subgraphs, model-string must be "multievent".

The **port-string** is the name assigned to the port which exercises the describes protocol. **timeunit-string** is either "ms", "ns", or "us". Only one may be specified, and timing constraints in the event graph are expressed in these units. The **mode** attribute "validated" is only used to show that the event graph has been checked for consistency.

nodelist contains a list of all nodes in the graph, and **controllist** is a list of all event precedence edges. **node-definitions** and **edge-definitions** respectively stand for a list of node of edge-structures definitions, as defined below.

Node-Structure

A node is a named entity, embedded in a graph. Again, a node represents an event, or a collection of simultaneous event via a subgraph. The label for the model is **name-string**, and the **class** field specifies whether the node represents a single event or multiple event. In the multiple event case, the **master** field identifies the subgraph that contains the event nodes. Only the single event nodes use the **arguments** field.

node-definition :=
(node
(name name-string)

(class class-string)
(master master-string)
(arguments argument-list)
(in_control edge-list)
(out_control edge-list)
)

For single event nodes, **name-string** is "n#", **class-string** is "event", and **master-string** is "event", where # is an unique integer tag. The **argument-list** is a list of properties that describes the actual signal transition. **arguments** is specified as:

(arguments (
(signal signal-string)
(bitvectwidth width-integer)
(bitvectbase base-integer)
(value transition-value-string)
(direction direction-string)
(valid valid-string)*
(invalid invalid-string)*
(phase phase-string)
))

where:

signal-string is the name of the signal on which the event occurs.
width-integer for bus signals, >= 1; optional, default=1 if not specified.
base-integer is the l.s.b. of the bus, >= 0; default=0 if not specified.
transition-value-string is "r", "f", "z", "s", "x", or "bitvector" (Table 1).
direction-string is "in", "out", or "tri"(tri-state output).

and for describing events on control signals only:

valid-string is a name of an information signal; optional.
Any information signal that is set-up before this event (being described) is named in a **valid** field.
invalid-string is a name of an information signal; optional.
Any information signal that must be held past this event (being described) is named in a **invalid** field.
This argument is the counterpart of the **valid** argument.
phase of the signal is "set" or "reset". For an active high signal, phase is "set" for the rising event and "reset" for the falling event, and the opposite for active low signals.

For multiple event nodes, **name-string** is "nn#", **class-string** is "Hierarchy", and **master-string** is

the name of the appropriate subgraph. Again ,# is an unique integer tag. This type of node does not use the arguments field.

The class, master and arguments fields of a node describes the event represented by the node. In contrast, it is the **in_control** and **out_control** fields where connectivity between nodes and edges in the main event graph is specified. The **edge-list** for **in_control** is a list of names of the input edges to the node, whereas **out_control** specifies the output edges. The input or output edges must be specified int the **controllist** of the event graph, as described below. Note that subgraphs do not use in_control and out_control fields, since the events within them occur simultaneously.

Edge-Structure

An edge is a named entity, embedded in a graph. Again, an edge represents precedence and can have timing constraints associated to it. Three classes of edges can be defined. "control" class edges represent precedence between two control events, whereas "ctrlinfo" class edges represent precedence between a control and information event, and vice versa. Lastly, "timing" class edges are used to specify timing constraints between two events that do not have a causal relationship.

edge-definition :=
(edge
(name name-string)
(class class-string)
(arguments argument-list)
(in_nodes node-list)
(out_nodes node-list)
)

name-string is a unique string.

class-string is "control, "ctrlinfo" or "timing" as explained.

Timing constraints are expressed in the **arguments** field. Currently, synthesis supports the min-max-avg model of delays, and delays must be specified as integer constants. Non-constant delays depend on a parameter(s), such as clock period), can be specified as a string but are not recognized

by synthesis. The **arguments** format is:

(arguments (

(min min-atom)

(max max-atom)
(avg avg-atom)
))

where:

min-atom is an integer >= 0, or string; optional, default=0.
max-atom is an integer >= 0, or string; optional, default=+infinity.
avg-atom is an integer >= 0, or string; optional.

The **node-list in_nodes** and the **out_nodes** fields name the input and output node connected to the edge under consideration. An edge has one and only one in_node, and one and only one out_node.

Example AFL Specification

The AFL specification for the event graph in Figure C is shown below:

```
/* Event graph for
 * module   : Static RAM
 * port     : 8-bit data, 8-bit address
 * protocol: write access
 */

(GRAPH
   (NAME writem)
   (CLASS MODULE)
   (MODEL ( (model_name slavep) ))
   (ARGUMENTS ( (port sram) (timeunit ns) ))
   (NODELIST
      (NODE
         (NAME n0)
         (CLASS event)
         (MASTER event)
         (ARGUMENTS (
            (signal CS_L)
            (value f)
            (direction in)
            (valid DATA)
            (valid ADDR)
            (valid WE_L)
            (phase set) ))
        (IN_CONTROL (e1 e2 e4 e7 e10) )
        (OUT_CONTROL (e0) )
     )
```

```
(NODE
   (NAME n1)
   (CLASS event)
   (MASTER event)
   (ARGUMENTS (
      (signal CS_L)
      (value r)
      (direction in)
      (invalid DATA)
      (invalid ADDR)
      (invalid WE_L)
      (phase reset) ))
   (IN_CONTROL (e0) )
   (OUT_CONTROL (e1 e5 e8 e11) )
)
(NODE
   (NAME n2)
   (CLASS event)
   (MASTER event)
   (ARGUMENTS (
      (signal WE_L)
      (value "0")
      (direction in)) )
   (IN_CONTROL (e6) )
   (OUT_CONTROL (e7) )
)
(NODE
   (NAME n3)
   (CLASS event)
   (MASTER event)
   (ARGUMENTS (
      (signal ADDR)
      (bitvectwidth 8)
      (bitvectbase 0)
      (value s)
      (direction in) ))
   (IN_CONTROL (e3) )
   (OUT_CONTROL (e4) )
)
(NODE
   (NAME n4)
   (CLASS event)
   (MASTER event)
   (ARGUMENTS (
      (signal DATA)
      (bitvectwidth 8)
      (bitvectbase 0)
      (value s)
      (direction in) ))
   (IN_CONTROL (e9) )
   (OUT_CONTROL (e10) )
)
(NODE
   (NAME n5)
   (CLASS event)
   (MASTER event)
   (ARGUMENTS (
      (signal WE_L)
      (value x)
      (direction in)) )
   (IN_CONTROL (e8) )
```

```
        (OUT_CONTROL (e6) )
    )
    (NODE
       (NAME n6)
       (CLASS event)
       (MASTER event)
       (ARGUMENTS (
          (signal ADDR)
          (bitvectwidth 8)
          (bitvectbase 0)
          (value x)
          (direction in) ))
       (IN_CONTROL (e5) )
       (OUT_CONTROL (e3) )
    )
    (NODE
       (NAME n7)
       (CLASS event)
       (MASTER event)
       (ARGUMENTS (
          (signal DATA)
          (bitvectwidth 8)
          (bitvectbase 0)
          (value x)
          (direction tri) ))
       (IN_CONTROL (e11) )
       (OUT_CONTROL (e9) )
    )
    (NODE
       (NAME start0)
       (CLASS header)
       (MASTER start)
       (OUT_CONTROL (e2) )
    )
)
(CONTROLLIST
    (EDGE
       (NAME e0)
       (CLASS control)
       (ARGUMENTS ( (min 40) ))
       (IN_NODES (n0) )
       (OUT_NODES (n1) )
    )
    (EDGE
       (NAME e1)
       (CLASS control)
       (IN_NODES (n1) )
       (OUT_NODES (n0) )
    )
    (EDGE
       (NAME e2)
       (CLASS control)
       (IN_NODES (start0) )
       (OUT_NODES (n0) )
    )
    (EDGE
       (NAME e3)
       (CLASS ctrlinfo)
       (IN_NODES (n6) )
       (OUT_NODES (n3) )
    )
```

```
      (EDGE
         (NAME e4)
         (CLASS ctrlinfo)
         (ARGUMENTS ( (min 5) ))
         (IN_NODES (n3) )
         (OUT_NODES (n0) )
      )
      (EDGE
         (NAME e5)
         (CLASS ctrlinfo)
         (IN_NODES (n1) )
         (OUT_NODES (n6) )
      )
      (EDGE
         (NAME e6)
         (CLASS ctrlinfo)
         (IN_NODES (n5) )
         (OUT_NODES (n2) )
      )
      (EDGE
         (NAME e7)
         (CLASS ctrlinfo)
         (IN_NODES (n2) )
         (OUT_NODES (n0) )
      )
      (EDGE
         (NAME e8)
         (CLASS ctrlinfo)
         (IN_NODES (n1) )
         (OUT_NODES (n5) )
      )
      (EDGE
         (NAME e9)
         (CLASS ctrlinfo)
         (IN_NODES (n7) )
         (OUT_NODES (n4) )
      )
      (EDGE
         (NAME e10)
         (CLASS ctrlinfo)
         (IN_NODES (n4) )
         (OUT_NODES (n0) )
      )
      (EDGE
         (NAME e11)
         (CLASS ctrlinfo)
         (ARGUMENTS ( (min 5) (max 10) ))
         (IN_NODES (n1) )
         (OUT_NODES (n7) )
      )
   )
)

/* End of AFL */
```

APPENDIX B: IDL USER'S GUIDE

The IDL hardware description language was developed especially to describe inter-module communication behavior at the system level. It is a procedural language and serves as the input into the ALOHA interface generation system. For use with ALOHA, the language is coupled to the SIERA module library, although it can be used in a stand-alone fashion for documentation. The specification model is based on the SSCS specification for distributed systems, but it has an appearance similar to the C programming language and the BDS language for combinational logic. The IDL grammar is implemented with a parser that transforms the text description into a parse tree data structure, with syntax error-reporting capabilities. A translator provides the front-end into synthesis by constructing the flow graph data structure from the parse tree. The parser and translator are described in Appendix C.

The behavioral model and semantics for the IDL language are treated in Chapter 4. Basically, behavior is specified as a network of modules that transmit and receive data through ports. Inter-module transfers are a temporal and spatial mapping of source information streams to destination streams. Describing a transfer includes the naming the I/O protocol each module uses to

synchronize the transfer. The modules and protocols come from the SIERA module library. The specification also includes standard data flow and control flow features found in many hardware description languages. This appendix describes the complete IDL format, consisting of syntax rules and use restrictions. The current ALOHA synthesis implementation supports most of the features. The features remaining to be implemented will be pointed out. The end of this appendix presents full specification examples.

Language Constructs

Character Set

The input text of an IDL description consists of names, numbers and reserved keywords separated by a space(s) or carriage return. Comments are preceded by a '!' exclamation point and ended by a carriage return.

A legal name is a any double quoted entity "..." or a string formed from the character set: A-Z, a-z, 0-9, _ (underscore), and [ ] (square brackets). The first character of a string must not be a number.

Numbers are either a *non-negative* integers, such as 0 or 25, or a bit vector surrounded by single quotes, such as '1' or '01101'. Integers must be non-negative too. When an integer is used in an assignment statement (described later), it is converted into the equivalent binary form.

Reserved Keywords

The following are reserved words and symbols in IDL. They must not be used as names.

ALL AND BDS BDSFILE BDSYN BIDIRECT BLOCK BUFFER
BVE CONSTANT DATAONLY DESIGN DEST DO ELSE
ENDBLOCK ENDDESIGN ENDFUNCTION ENDIF ENDITERATE
ENDPORT ENDPROCEDURE ENDRESETPROC ENDROUTINE ENDWHILE
EQL FALL FROM FUNCTION GEQ GTR IF INPUT ITERATE
LEQ LSS NAND NEQ NEXT NOR NOT OR OUTPUT
PACKL PACKR PORT PROCEDURE RESET RESETPROC RESTART
RETURN RISE ROUTINE SHL SHR SOURCE STATE THEN TO

WHILE XOR PLUSEQ RIGHTARROW LEFTARROW

! + - * / ^ ( ) { }

< > = @ . : ; , “

Signals and Variables

Signals are inputs and outputs or internal vectors of the described behavior. They can be thought of as physical wires, and synthesis will allocate hardware to realize them. Signals are specified as a name and optionally with bit subscripts that define their width. The three formats for a signal are:

Signal_name < msb : lsb > Signal_name <bit> Signal_name

Signal_name is a legal name. The bit subscripts msb, lsb and bit are non-negative integers or a symbolic name for an integer constant (discussed later). If the subscripts are omitted, then the signal is one bit wide.

Variables serve as internal variables in a BLOCK declaration or indices in ITERATE and WHILE loop statements. Specifying them does not necessarily imply a hardware implementation. A variable is specified simply using a legal name.

I/O Transactions

As explained in Chapter 4, an I/O transaction defined as the triplet {module_instance, protocol, signal}. It can also be the instantaneous signal value, so the pair {module_instance, signal} is sufficient. The signal is defined as above. The protocol is the name of an event graph from the module library. The module_instance is an instance of a module from the library. The transaction is specified as:

module_instance_name/protocol_name.signal module_instance_name.signal

Input Format

The IDL format is defined below using the BNF description. Keywords are shown in upper-case letters. User-defined names and numbers are shown in lower case italic letters, while productions are shown in lower-case plain letters. Features in the "{ }" brackets may be repeated zero or more times, and those in "{ }+" may be repeated one or more times. An optional feature is surrounded by square brackets "[ ]". The vertical bar "|" represents a choice of items.

Any IDL description has a header that *declares* ports, input and output signals, and symbolic constants used in the body. Then the behavior is actually specified in the body with *local* variable declarations, transfer (assignment) and control flow *statements*, and data flow *expressions*.

An IDL description is contained in a DESIGN declaration:

DESIGN *design_name*
    {constant_declaration}
    {port_declaration}+
    {block_declaration}+
ENDDESIGN [*design_name*];

A constant_declaration is a symbolic name for a non-negative integer or a bit vector:

CONSTANT *constant_name* = *number*;

A port_declaration declares instances of a library module and the I/O signals used in the body:

PORT *library_module_name instance_name* {,*instance_name*};
    {SOURCE *signal* {,*signal*} ;}
    {DEST *signal* {,*signal*} ;}
    {BIDIRECT *signal* {,*signal*} ;}
ENDPORT [*library_module_name*];

A block_declaration contains the behavioral description. It consists of three types of local variables (buffer, state and input) and four types of behavioral subblocks (routine, function, procedure and resetproc). Currently, synthesis does not recognize the local variable declarations. The semantics of the subblocks are explained in Chapter 4. They must not be recursive (calls itself). Only the routine subblock may call the another subblock. The block declaration is defined as:

```
BLOCK block_name
    {buffer_declaration}
    {state_declaration}
    {input_declaration}

    ROUTINE routine_name;
        {statement}
    ENDROUTINE;

    {FUNCTION function_name<return_bit_width>(parameter1{, parameter2});
        {statement}| BDS "file_name";
     ENDFUNCTION;}

    {PROCEDURE procedure_name;
        {statement}
     ENDPROCEDURE;}

    [ RESETPROC reset_name;
        {statement}
     ENDRESETPROC;]
ENDBLOCK [block_name];
```

Inter-module transfers may be buffered with memory called a queue, FIFO or pipeline buffer. The buffer's depth is the number of signal values it can hold, and the width is the bit-width of the signal. The depth is a system design parameter. A buffer declaration explicitly specifies local memory, implying hardware within the BLOCK construct:

BUFFER *buffer_name(source_signal, destination_signal, width, depth)*;

Memoryless internal variables are declared in the following format. They must not be a destination signal declared in the PORT declaration. But, they may also be outputs of the block that they are declared in.

STATE *signal = initial_constant_value;*

An input signal to a block can be from the output of another block. This input signal is declared as:

INPUT *signal* {,*signal* };

A statement in a routine, function, procedure or reset-procedure can be one of two data flow statements or nine control flow statements. The data flow statements are the assignment, which represents an inter-module transfer, and the BDS statement, which names a BDS file that describes combinational logic. Control flow statements specifies how transfers are sequenced from one to the next. They consist of the reset, restart, procedure call, return, next, if, iterate, while and

concurrent statements. The bold curly brackets "{" and "}" below are actually brackets in the specification. The formats for the statements are:

```
| BDS "file_name ";
| transaction = expression;
| RESET;   | RESTART;
| procedure_name();
| RETURN expression;
| NEXT ( );
| NEXT ( (mod_inst_name[,integer]) {,(module_inst_name[,integer])} );
| IF expression THEN {statement} [ ELSE {statement} ] ENDIF;
| ITERATE index_variable FROM integer TO integer DO {statement} ENDITERATE;
| WHILE expression DO {statement} ENDWHILE;
| { statement(s) }
```

Some statements above have restrictions regarding their use. The BDS and RETURN statements are only used within a FUNCTION declaration. The RETURN statement causes a function to exit and return a value to the ROUTINE subblock. The RESET statement is only used in a RESETPROC declaration. When it is encountered in the RESETPROC procedure, control flow in the ROUTINE is interrupted and returned to the beginning. The RESTART statement is only used in the ROUTINE declaration. Similar to the RESET statement, a RESTART control in the ROUTINE to return to the beginning. Also, only the ROUTINE can make a procedure call, and use the NEXT, IF, ITERATE and WHILE statements. The WHILE construct is not supported by the current ALOHA synthesis implementation.

IDL expressions describe data flow behavior. An expression can be embedded within another by surrounding it with parentheses, "(expression)". The innermost expressions are evaluated first. The expressions listed below from highest to lowest precedence; the highest is evaluated first. The precedence can be overridden by using the parentheses. Expression have the formats:

```
| number    | constant
| signal    | transaction
| ( expression )
| expression@integer   | expression@constant
| function_name( argument{,argument} )          ! argument is a signal or @ expression
| expression + expression
| expression EQL expression
| expression NEQ expression
| NOT expression
| expression AND expression
| expression NAND expression
| expression OR expression
```

| expression NOR expression
| expression XOR expression

Examples

The following four IDL descriptions demonstrate how various language declarations, expressions and statements are used to describe inter-module communication behavior. The four examples presented in Section 4.4 of Chapter4 are actually simplified versions of the ones shown here.

The first specifies the behavior of a VMEbus interface to a single static RAM. It highlights conditional behavior using the IF statement, which is controlled by the decode FUNCTION. The actual function description is in a separate BDS file.

The second example describes multiplexed transfers between a TMS320 processor and a TAXI optical link transmitter module. The TMS provides address and data on parallel lines in one transaction. The interface sends the address first and then the data to the signal TAXI data line. It illustrates the BUFFER declaration, the NEXT control statement and also a combinational logic description in the FUNCTION.

Demultiplexed transfers between a TAXI optical receiver and a D/A module are described in the third example. Here, words from two consecutive TAXI transactions are collected and sent to the D/A module; the first word serves as the address to the module and the second is the data to be converted. It illustrates a transfer that involves the @ delay and function call expression, and also a RESETPROC declaration that involves the logical OR expression and the RESET control statement.

The final example shows the IDL description for an interface between a bank of A/D converters and the TAXI optical transmitter. It illustrates the CONSTANT declaration, the PROCEDURE declaration, and an application of the ITERATE statement toward describing block transfers.

VME System Bus Interface

```
!IDL description: VMEbus read/write access to a SRAM module.

DESIGN vme_interface

   ! Interface inputs and outputs
   PORT VMEbus vme;
      SOURCE vme.WRITE_L, vme.A<14:1>;
      BIDIRECT vme.D<15:0>;
   ENDPORT;

   PORT m1621 mem; ! static RAM
      DEST mem.WE_L, mem.ADDR<10:1>;
      BIDIRECT mem.DATA<15:0>;
   ENDPORT;

   ! communication behavior
   BLOCK vme_interface

      ! main routine
      ROUTINE main;
         IF (decode(vme/all.A<14:11>) EQL '1') THEN !sram decode
            IF (vme/all.WRITE_L EQL '0' ) THEN  !write access
               {mem/writem2.WE_L = '0';
                mem/writem2.ADDR<10:1> = vme/dtb_write.A<10:1>;
                mem/writem2.DATA<15:0> = vme/dtb_write.D<15:0>;}
            ELSE                                    ! read access
               {mem/readm.WE_L = '1';
                mem/readm.ADDR<10:1> = vme/dtb_read.A<10:1>;
                vme/dtb_read.D<7:0> = mem/readm.DATA<7:0>; }
            ENDIF;
         ENDIF;
      ENDROUTINE;

      !decode function
      FUNCTION decode<0>(x<3:0>);
         BDS "decoder.bds";
      ENDFUNCTION;

   ENDBLOCK;

ENDDESIGN vme_interface;
```

TMS320 Processor to Optical Link Multiplexing

```
! IDL description: TMS320C30 DSP to AMD AM7968 TAXI module.

DESIGN tms2taxi_interface

   !Interface inputs and outputs
   PORT tms_iobus tms;
      SOURCE tms.XR_W_L,tms.XA<4:0>,tms.XD<3:0>;
   ENDPORT;

   PORT taxixmt_bus taxi;
      DEST taxi.DI<3:0>,taxi.CI<1:0>;
   ENDPORT;

   !communication behavior
   BLOCK tms2taxi_interface

      BUFFER buf1(tms.XA<3:0>,taxi.DI<3:0>,4,6); ! depth=6
      BUFFER buf2(tms.XD<3:0>,taxi.DI<3:0>,4,6);

      ROUTINE main;
         IF (tms/all.XR_W_L EQL '0') THEN    ! write access

            !multiplex tms address and data to taxi DI line.
            IF (tms/all.XA<4> EQL '1') THEN
               ! send tms address and header
               {taxi/xmt.DI<3:0> = tms/exb_write.XA<3:0>;
                taxi/xmt.CI<1:0> = '00';}
               NEXT();
               ! send tms data and header
               {taxi/xmt.DI<3:0> = tms/exb_write.XD<3:0>;
                taxi/xmt.CI<1:0> = '01';}
            ENDIF;

            ! no multiplexing
            IF ( decode(tms/all.XA<4:0>) EQL '1' ) THEN
               {taxi/xmt.CI<1:0> = tms/exb_write.XD<1:0>;}
            ENDIF;

         ENDIF;
      ENDROUTINE;

      FUNCTION decode<0>(x<4:0>);
         IF (x<4:0> EQL '00000') THEN RETURN '1';
         ELSE RETURN '0';
         ENDIF;
      ENDFUNCTION;

   ENDBLOCK;

ENDDESIGN tms2taxi_interface;
```

Optical Link to D/A Converter Module Demultiplexing

```
! IDL description:AM7969 TAXI module to D/A converter module.

DESIGN  taxi2dac_interface

   !Interface inputs and outputs

   PORT taxircvr_dbus taxid;
      SOURCE taxid.Do<9:0>;
   ENDPORT;

   PORT taxircvr_cbus taxic;
      SOURCE taxic.Co<1:0>;
   ENDPORT;

   PORT dac_bank dac;     ! bank of 9 dacs
      DEST dac.D<9:0>,dac.BankSel<8:0>;
   ENDPORT;

   PORT DATAONLY rst;
      SOURCE rst.powerup_reset;
   ENDPORT;

   ! communication behavior
   BLOCK demux

      !collect two consecutive words from taxid and send to dac.
      ROUTINE main;
         NEXT( (taxid) );
         {dac/dacb_write.BankSel<8:0> = decode(taxid/rcvD.Do<9:0>@1);
          dac/dacb_write.D<9:0> = taxid/rcvD.Do<9:0>;}
      ENDROUTINE;

      FUNCTION decode<8:0>(x<9:0>);
         BDS "decoder.bds";
      ENDFUNCTION;

      RESETPROC reset;
         IF (rst.powerup_reset EQL '1'
               OR taxic/rcvC.Co<1:0> EQL '11')  THEN
            RESET;
         ENDIF;
      ENDRESETPROC;

   ENDBLOCK demux;

ENDDESIGN taxi2dac_interface;
```

A/D Converter Module to Optical Link Block Transfers

```
! IDL description: AD7870 A/D converter bank to AM7968 TAXI module.

DESIGN adc2taxi_interface

   CONSTANT Iportaddr='1000'; ! address of A/D Iport

   ! Interface inputs and outputs
   PORT DATAONLY status;    ! external command signals
      SOURCE status.reset,status.err,status.int;
   ENDPORT;

   PORT taxi_xmt taxi;
      DEST taxi.DI<9:0>,taxi.CI<1:0>;
   ENDPORT;

   PORT adc_bank adc;        ! bank of 6 A/Ds and control port.
      SOURCE adc.DB<9:0>;
      DEST adc.CS_L<3:0>;   ! CS_L is the bank select.
   ENDPORT;

   BLOCK adc2taxi    ! communication behavior

      ROUTINE main;
         IF (status.err EQL '1') THEN          ! err=1
            {taxi/xmt.CI<1:0> = '11';}
            NEXT();
            xmit();    ! procedure call
         ELSE
            IF (status.int EQL '0') THEN       ! err=0, int=0
               xmit();
            ELSE                               ! err=0, int=1
               {taxi/xmt.CI<1:0> = '10';}  ! transfer header
               NEXT();
               ITERATE j FROM 0 TO 5 DO    !send block of 6 words
                  {adc/b_read1.CS_L<3:0> = j;
                   taxi/xmt.DI<9:0> = adc/b_read1.DB<9:0>;}
                  NEXT( (adc) );
               ENDITERATE;
            ENDIF;
         ENDIF;
      ENDROUTINE;

      PROCEDURE xmit;
         {taxi/xmt.CI<1:0> = '01';}
         NEXT();
         {adc/b_read1.CS_L<3:0> = Iportaddr; ! access A/D Iport
          taxi/xmt.DI<9:0> = adc/b_read1.DB<9:0>;}
      ENDPROCEDURE;

      RESETPROC reset;
         IF status.reset EQL '1' THEN
            RESET;
         ENDIF;
      ENDRESETPROC;

   ENDBLOCK;
ENDDESIGN;
```

APPENDIX C: ALOHA SOFTWARE IMPLEMENTATION

The ALOHA synthesis system consists of libraries, tools and the design examples used throughout this thesis. It can be a stand-alone system and also a point tool within the SIERA CAD environment for system design [Srivastava92].

This appendix serves as a guide to the ALOHA software. It should be read before using the synthesis tools or modifying/compiling the source code. The first section shows how to compile and run the software. The second shows the organization of the ALOHA software located in the ~siera home directory, and it explains what each subdirectory contains.

Using ALOHA

The software is installed on the *BroderSuns* workstation cluster in the EECS Department, currently located in the directory ~siera/shared/aloha. It runs in the UNIX operating system.

ALOHA provides two main tools located in ~siera/shared/aloha/bin. The validation tool, validateEG*, performs consistency checks on I/O protocol event graphs which are installed into

the library ~siera/shared/aloha/cellib (part of module library discussed in Chapter 3). All event graphs are described in the AFL format (see Appendix A) and must be in a file named "root_name.afl". The synthesis tool, aloha*, generates the design RTL structure and event graphs from a given IDL description (discussed in Chapter 4 - Chapter 8). The IDL file must be names "root_name.idl". To run either of these two tools:

1. Deposit a copy of the file ~siera/shared/aloha/hyper and ~siera/shared/aloha/lager into the working directory or home directory. This file contains paths to ALOHA libraries.
2. Invoke "validateEG [-n] AFL_file",

AFL_file: root name of event graph AFL file

-n : do not dump new AFL file

or invoke "aloha [-I -A -O -i -a -n -v] IDL_file",

IDL_file: root name of IDL file

-I: reads IDL file

-A: reads flow graph in AFL file (currently not supported)

-O: reads flow graph from OCT (currently not supported)

-i: debug information

-a: dumps final flow graph in AFL format

-n: dumps RTL netlist in AFL or SDL format,
combinational logic descriptions in BDS format,
interlocked event graphs in ASTG format,
time constraints in CLOVER format

-v: verbose mode used with -A flag

Currently, the typical command line will be "aloha -I -a -n IDL_file" or "aloha -I -n IDL_file".

The -i option provides error and warning messages related to synthesis. It is useful to the tool developer rather than the tool user (usually the system designer).Currently, the error reporting capabilities to the user is primitive.

The ASTG output produced by ALOHA drives the ASTG logic synthesis tool. Currently, only the SIS alpha version in /usr/tools/async/sis/sis contains the ASTG package. The next beta release installed into ~octtools should be used when made available.

ALOHA Software Organization

The ALOHA synthesis tools are implemented in the C programming language, and uses the graph data structures and utility libraries provided by the HYPER system in the ~hyper home directory [Rabaey90]. Copies of the relevant HYPER include (.h) files and archived libraries (.a files) have been made and deposited into the ALOHA source directory (/src/hyperLib). Currently, the HYPER archives are compiled for the Sun4 workstations. When installing ALOHA on another machine, the HYPER archives must be recompiled from source code in ~hyper. A second caveat is that the HYPER software uses data structures and utility libraries from ~octtools. So, it should always be kept in mind that changes in ~octtools can affect the state of the ALOHA software.

The file organization of ALOHA is listed below. Many subdirectories have accompanying on-line documentation in ascii and FrameMaker files. To compile any of the ALOHA source code, use the provided makefile in the directory containing the code.

The root directory of ALOHA located in ~siera/shared/aloha is organized as follows:

a. /src
   Source code and header files for main programs and utility routine libraries.

b. /lib
   Libraries that support the ALOHA flow graph and event graph data structure. Keep in mind that these two structures are based on the HYPER graph data structure. /lib supplements the generic libraries provided by HYPER.

c. /bin
   Binaries compiled from the source code in /src. Contains validateEG* and aloha*. The aloha* tool is actually made up of routines in Idl2Flow*, Flow2EG*, Flow2DP* and Flow2IC* programs also in the same directory.

d. /cellib
   Library of I/O protocol event graphs for various hardware modules, each contained in a subdirectory. A subdirectory has the AFL event graph files and FrameMaker pictorial documentation. See Appendix A for details on installing event graphs.

e. /demo
   Five design examples, each contained in a subdirectory. The four standard examples used to illustrate design methods throughout this thesis are in /demo. Subdirectories contain a README file explaining how aloha* was invoked to generate the output files.

f. /doc
Ascii and FrameMaker documentation related to ALOHA software implementation and use.

The /lib and /src directories listed above contain subdirectories and files described below.

First, the /lib directory holds the following:

a. /ENODELIB
Library supporting the event graph data structure. Also has the event graph for the Ictrl handshake protocol described in Section 7.1.2 and Figure 7-6.

b. /FNODELIB
Library of primitive flow graph nodes that support the flow graph data structure. Supplements the ones found in ~hyper/COMMON/lib/NODELIB.

the /src directory contains the following subdirectories of source code and README files:

a. /system
Holds source code for the two main ALOHA tools, validateEG* and aloha*.

b. /hyperLib
Copies of include files from ~hyper/COMMON/include; copies of archived libraries from ~hyper/COMMON/src/flowLib, ~hyper/COMMON/src/transform and ~hyper/COMMON/src/hyperUtil.

c. /flowLibAux
Library of utility routines that support the ALOHA flow graph data structure. Supplements the generic HYPER library in ~hyper/COMMON/src/flowLib. The ascii README file documents the prototypes and functionality of the flowLibAux routines.

d. /idlLib
IDL parser and pass routines. Implements the IDL grammar described in Chapter 4 and Appendix B, using the UNIX Lex and Yacc facilities. Produces a parse tree called AST (abstract syntax tree) and is thoroughly documented in README.ast.frame and README.ast_attributes. Look at the documentation before modifying this source code!

e. /i2f
Translator that constructs an ALOHA flow graph data structure from an IDL parse tree. See section 5.2.1 of Chapter 5 for details.

f. /timeconvert
First part of scheduler and produces initial flow graph schedule. Creates control steps, schedules inputs and outputs, and storage allocation. Implements scheduling techniques described in Section 6.2 of Chapter 6.

g. /schedule
Second part of scheduler. Implementing various scheduling details described in Sections 6.2 and 6.3, including some optimizations.

h. /cluster
Programs to insert sync nodes into the flow graph and cluster data flow nodes. Corresponds to the synthesis techniques discussed in 5.1.1 and 5.2.3.

i. /eventLib
Library of utility routines that support the event graph data structure. The event graph data structure is based on the HYPER graph data structure, just as the flow graph is. The ascii README file documents the prototypes and functionality of the /eventLib routines.

j. /f2e
Programs to generate interlocked event graphs from flow graph and I/O protocol event graphs in ~siera/shared/aloha/cellib. Implements the techniques described in Chapter 7.

k. /f2d
Programs to generate the RTL datapath netlist from the flow graph. Corresponds the techniques shown in Section 8.1 of Chapter 8.

l. /f2i
Programs to generate the interface controller netlist and FSM core description (in BDS format). Details described in Section 8.3.

m. /transLib
Library of utility routines to support BDS and CLOVER and SDL translator source code in /translators.

n. /translators
Currently contains two low-level translators. Ast2Bds converts an AST parse tree representation of combinational logic equations into the BDS format. AflToClover converts time constraints in a event graph from AFL format to CLOVER format. ALOHA internally uses the AFL format to represent structural netlists. Afl2Sdl converts an Afl netlist representation into the Sdl netlist format.