

Copyright © 1992, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**RAPID-PROTOTYPING OF HARDWARE AND
SOFTWARE IN A UNIFIED FRAMEWORK**

by

Mani Bhushan Srivastava

Memorandum No. UCB/ERL M92/67

15 June 1992

COVER PAGE

**RAPID-PROTOTYPING OF HARDWARE AND
SOFTWARE IN A UNIFIED FRAMEWORK**

by

Mani Bhushan Srivastava

Memorandum No. UCB/ERL M92/67

15 June 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

Rapid-Prototyping of Hardware and Software in a Unified Framework

by

Ph.D.

Mani Bhushan Srivastava

Department of EECS

Abstract

Modern electronic systems contain a mix of software running on general-purpose programmable processors, algorithms hardwired into dedicated hardware such as custom boards and chips, electromechanical components, and mechanical interconnect and packaging. Therefore a systems perspective of the design process is essential, as opposed to the conventional "chip-focussed" approach. A design framework for application-specific systems is described in which higher level aspects of system design, including software, multi-chip design issues present at the board level, and hardware-software integration are addressed, in addition to the design of individual custom chips. A high-level description of the system as a network of processes is mapped to a system architecture template consisting of multiple boards using dedicated hardware modules and ASICs as well as software processes running on programmable hardware modules. Use of the framework to applications in multi-sensory robotics and real-time speech recognition is also described.



Robert W. Brodersen

Chairman of Committee

THE UNIVERSITY OF MICHIGAN LIBRARY

1964

TO THE UNIVERSITY OF MICHIGAN LIBRARY
FROM THE UNIVERSITY OF MICHIGAN LIBRARY

To Mom and Dad.

TO THE UNIVERSITY OF MICHIGAN LIBRARY
FROM THE UNIVERSITY OF MICHIGAN LIBRARY



UNIVERSITY OF MICHIGAN LIBRARY
ANN ARBOR, MICHIGAN

Table of Contents

INTRODUCTION	1
1.1 Motivation and Objective	1
1.2 Problem Formulation.....	3
1.2.1 System Examples.....	3
1.2.2 Problems in System Design	9
1.3 Previous Work	10
1.3.1 Computer Aids for Hardware Development.....	11
1.3.2 Computer Aids for Software Development.....	12
1.3.3 Computer Aids for Concurrent Development of Hardware and Software.....	13
1.4 Summary	18
SIERA SYSTEM DESIGN FRAMEWORK	19
2.1 Overview of SIERA	20
2.1.1 System Design Methodology in SIERA	23
2.2 The Philosophy Behind SIERA.....	32
2.3 Summary	34
GENERATION OF HARDWARE MODULES	37
3.1 Overview of Hardware Module Generation Environment.....	38
3.2 Design Management Framework	39
3.2.1 OCT Database	40
3.2.2 Policies for Design Representation in OCT	41
3.2.3 Design Manager DMoct.....	47
3.3 Board Layout Generation Tools	50
3.3.1 psg: Package Symbol Generator	51
3.3.2 pfp: PCB Floor-Planner	51
3.3.3 oct2rinf: Interface to Foreign Router	55
3.4 Module Generation from Behavioral Specifications	56
3.4.1 Mapping Random Logic to PLDs and FPGAs.....	57
3.4.2 Generation of ASICs from Behavioral Description	62
3.4.3 Synthesis of Interface Logic	63

3.5	Libraries	65
3.5.1	Package Library	65
3.5.2	Primitive Component Library	66
3.5.3	Subsystem Module Library	69
3.6	Simulation and Netlist Checking	80
3.7	Board Example	83
3.7.1	DSP Multiprocessor Board	84
3.8	Summary	87
SOFTWARE MODULES		89
4.1	Software Issues in Application-Specific Systems	90
4.2	Software Module Organization.....	91
4.2.1	Existing Control Mechanisms for Software Modules.....	92
4.2.2	Using Processes as Software Modules	96
4.2.3	Implementation of Process Modules	97
4.3	Communication and Synchronization of Software Modules.....	101
4.4	Run-Time and Development Environment.....	103
4.4.1	Software Module Loaders and Initialization Utilities.....	103
4.4.2	Client-Server Model Using RPC.....	105
4.4.3	File and Terminal I/O, and Other System Services	107
4.5	Generation of Software Modules.....	111
4.6	Summary.....	112
SYSTEM REPRESENTATION AND SIMULATION.....		113
5.1	High-Level System Modelling	114
5.1.1	Alternative Models of Inter-Process Communication	116
5.1.2	Model Used in this work	118
5.2	VHDL-based High-Level System Simulation	122
5.2.1	VHDL Package for Simulating the Process Network Model	123
5.2.2	Modelling Continuous-Time Subsystems using VHDL.....	135
5.3	Summary.....	140
ARCHITECTURE GENERATION		143
6.1	Approaches to Architecture Generation	144
6.1.1	Manually Specified Architecture.....	145

6.1.2 Fixed Architecture	145
6.1.3 Architecture Template	147
6.1.4 Automatic Architecture Generation.....	149
6.2 Co-Design of Hardware and Software	150
6.3 Template Mapping Based Approach to Architecture Definition.....	152
6.4 The Layered System Architecture Template	154
6.4.1 Alternative Models of System Architecture	154
6.4.2 Details of the Layered Architecture Template	160
6.5 Communication and Synchronization Mechanisms for the Layered Architecture Template	162
6.5.1 Communication and Synchronization in Software	164
6.5.2 Applicability of Communication and Synchronization Techniques in Software to Dedicated Hardware	166
6.5.3 Implementation of the Channels in the Architecture Template	168
6.6 Partitioning of the System Specification	176
6.6.1 Manual Partitioning.....	176
6.6.2 A Strategy for Automated Partitioning	180
6.6.3 Role of Process Network Transformation in Partitioning	181
6.7 Implementation of the Architecture Template.....	183
6.7.1 Implementation Restrictions and Guidelines.....	184
6.8 Summary	189
USING SIERA FOR DESIGNING SYSTEMS.....	191
7.1 The Three Entry Points to SIERA.....	191
7.2 Designing a Custom Board with Dedicated Hardware	192
7.3 Designing a Custom Board with Dedicated Hardware and Software Programmable Processors.....	193
7.4 Designing a System According to the Architecture Template	195
7.5 Summary	200
MULTI-SENSORY ROBOT CONTROL SYSTEM.....	201
8.1 System Requirements	201
8.1.1 Environmental Constraints	202
8.1.2 Goal.....	205
8.2 Algorithms.....	205

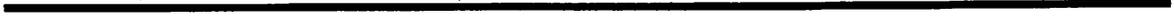
8.3	System Architecture	207
8.4	Hardware Organization.....	210
8.4.1	Controller Board	210
8.4.2	Peripheral Board	215
8.5	Software Organization.....	218
8.6	User's View of the Robot System.....	219
8.7	Summary.....	220
	GRAMMAR SUBSYSTEM FOR SPEECH RECOGNITION.....	221
9.1	Global Functionality of the Speech System	221
9.2	Old Implementation of the Speech System	225
9.3	Re-Implementation of Front-End Processing, Successor Computation, and Back-Track Processing	228
9.4	Board Requirements and Constraints	229
9.4.1	Front-End Processing.....	229
9.4.2	Successor Computation	232
9.4.3	Back-Track Processing.....	234
9.5	Hardware Architecture and Implementation	234
9.6	Software Implementation	240
9.7	Summary.....	240
	CONCLUSIONS AND FUTURE WORK.....	241
10.1	Conclusions from this Work	241
10.2	Open and Unsolved Problems	243
10.3	Future Directions in Systems Design	243
	BIBLIOGRAPHY	247
	APPENDIX A: VHDL PROCESS N/W PACKAGE	253
	APPENDIX B: THE ASSYS SOFTWARE UTILITIES.....	265
	APPENDIX C: SIERA SOFTWARE ORGANIZATION.....	267

List of Figures

Figure 1-1 : Block Diagram of a CD-I System.....	4
Figure 1-2 : Architecture of CMU Direct Drive Arm II Controller	6
Figure 1-3 : System Architecture Model of Vulcan-II (Figure 3 of [Gupta92a]).....	15
Figure 2-1 : Simplified Top Level Views of LAGER and SIERA	22
Figure 2-2 : System Design Using SIERA	24
Figure 2-3 : Multi-Layered Architecture Template for System Architecture Generation.....	28
Figure 2-4 : User's View of System Design in SIERA	31
Figure 3-1 : Design Flow using DMoct.....	48
Figure 3-2 : Board-Level Module Generation Using FPGAs	60
Figure 3-3 : Board-Level Module Generation Using PALs and PLDs.....	61
Figure 3-4 : Interconnect Module Generation Using ALOHA [Sun92a].....	64
Figure 3-5 : Sample SDL File for a xx00 (7400) Chip.....	67
Figure 3-6 : Layout of an Instance of procC30, a TMS320C30 based processor module	73
Figure 3-7 : Architecture of the TMS320C30 based procC30 Processor Module	74
Figure 3-8 : Hierarchy of SDL files for the procC30 Module.....	76
Figure 3-9 : Black-Box Picture of the opticalR Module	78
Figure 3-10 : Event Graphs for opticalR Sub-system Module.....	81
Figure 3-11 : Event Graph for opticalR Module in afl Text Format.....	82
Figure 3-12 : Board Architecture of the MC96002 Based Shared Memory Multi-Processor with Ordered Memory Access [Sriram92]	85
Figure 3-13 : SDL Hierarchy for the MC96002 Based Shared Memory Multi-Processor with Ordered Memory Access [Sriram92]	86
Figure 3-14 : Physical Layout of the MC96002 Based Ordered Memory Access Shared Memory Multi-Processor Board.....	88
Figure 4-1 : Organization of a Simple System Using Off-Shelf Hardware and Module Generators and Libraries from Chapter 3.....	90
Figure 4-2 : Control Mechanisms for Software Modules.....	92
Figure 4-3 : Implementation of Process Modules by Multiple Autonomous Kernels	99
Figure 4-4 : Similarity between Organization of Hardware and Software Modules in a System	101
Figure 4-5 : The Pseudo-Code for Boot-Strapping a Processor Module.....	105
Figure 4-6 : The SQ Package for Simple Low-Level Message Passing.....	106
Figure 4-7 : Client-Server Model for User-Interface to a Dedicated System.....	108
Figure 4-8 : Implementation of UNIX-like File and Terminal I/O for Programmable	

Processor Modules on Custom Boards	110
Figure 5-1 : Simple Example Systems Described as Process Networks	121
Figure 5-2 : VHDL Implementations of Channels for the Process Network Model	125
Figure 5-3 : A Simple Example Demonstrating the Use of the SHM and SEM VHDL Packages	134
Figure 5-5 : A Simple Example of a Mixed-Mode System	140
Figure 5-4 : Modelling a D. C. Motor in VHDL Using the Continuous-Time LTI Entity	141
Figure 6-1 : Typical System Based on General-Purpose Multi-Processor Architecture	146
Figure 6-2 : System Architecture Using Application-Specific Boards	148
Figure 6-3 : Template Based Approach to System Architecture Definition	153
Figure 6-4 : A Layered Architecture Template for Systems	155
Figure 6-5 : Synchronization Techniques in Software Systems and Their Relationships	166
Figure 6-6 : Required Functionality of the Layer 2 \leftrightarrow Layer 3 Memory Mapped Communication and Synchronization Interface	171
Figure 6-7 : Abstract View of the Communication and Synchronization Interface between Processing Modules on Layers 3 and 4	175
Figure 6-8 : A Hypothetical System Using the Architecture Template to Demonstrate the Syntax of a SAIL File	178
Figure 6-9 : Coalescing a Server Process into a Client Process, and vice versa	182
Figure 6-10 : Currently Available Implementation Choices for the System Architecture Template	183
Figure 6-11 : Reference Implementation of the Interface between Layer 2 and Layer 3	187
Figure 7-1 : Currently Implemented Design Flow for a Custom Board with Dedicated Architecture without using the System Architecture Template	194
Figure 7-2 : Using SIERA for the Entire Top-Down Design of a System	196
Figure 7-3 : Top-Level SDL File for a Custom Board in a System following the Architecture Template	199
Figure 8-1 : Functional Decomposition of the Robot Control System	208
Figure 8-2 : Architecture of the Multi-Sensory Robot Control System following the Architecture Template of Chapter 6	209
Figure 8-3 : Architecture of the Robot Controller Board	211
Figure 8-4 : Photograph of the Robot Controller Board	215
Figure 8-5 : Block Diagram and Photograph of the Custom Robot Peripheral Board	217
Figure 9-1 : Functional Decomposition of the Speech Recognition System	222
Figure 9-2 : Old Implementation of the Speech Recognition System	226
Figure 9-3 : The Speech System Using the New Custom Board for Front-End Processing, Back-Track Processing, and Successor Computation	229
Figure 9-4 : Front-End Processing Block	230

Figure 9-5 : Architecture of the Custom Board for Front-End Processing, Back-Track Processing, and Successor Computation	236
Figure 9-7 : SDL Hierarchy for the Custom Board.....	237
Figure 9-6 : Alternate Ways of Structuring Successor Computation Using the Flexible Custom Interface to the HMM Board.....	238
Figure 9-8 : Photograph of the Custom Board	239



List of Tables

Table 3-1 : A Partial List of Reserved Attributes in structure_master Policy Relevant to Board Level Hardware Design	44
Table 3-2 : Partial Listing of the Board Level Primitive Component Library	68
Table 3-3 : Partial Listing of the Board Level Sub-system Module Library	70
Table 3-4 : Main Features of the MC96002 Multi-Processor Board	87
Table 4-1 : Currently Supported Processor Modules and Associated Kernels	99
Table 5-1 : Examples of Emulating Various IPC Models with the MSG Package.....	125
Table 8-1 : Main Features of the Robot Controller Board	214
Table 8-2 : Main Features of the Robot Peripheral Board	216
Table 9-1 : Salient Features of the Custom Board	239



Acknowledgments

The encouragement and contributions of friends, colleagues, and mentors is essential to most human endeavors, and this research was no exception. First of all I would like express my sincere appreciation to my advisor Bob Brodersen for his constant encouragement, help, advice, resources, and for plenty of disk space! Most importantly though I would like to thank him for giving me the freedom to explore whatever projects I wanted to - the result of which is that I found my years at Berkeley to be the most enjoyable and satisfying of my life.

Next I would like to thank everybody who contributed to my research work one way or the other. Without Bill Baringer and Gautam Doshi's immense help and energy the robot system, which was much more than just an example for this work - it really provided the motivation behind this research - would never have come up. Later Manish Arya and Trevor Blumenau wrote system and application software respectively for the first generation hardware to help make the robot do something more dramatic than just smashing into the table. In particular Trevor helped define what the second generation robot system described in this thesis ought to do, in addition to helping me debug the hardware as well as software. In fact, most of the credit for section 8.2 should go to Trevor. Tony Stölzle, although initially skeptical, graciously volunteered to be the guinea pig and try out the approach espoused in this thesis for the speech recognition example. Sriram, from Prof. Ed Lee's group, was my other guinea pig who cheerfully struggled with the ever-changing tools and designed the ordered-memory access multi-processor board example in Chapter 3. I thank both Tony and Sriram for providing me with invaluable feedback. Most of the hardware module generators and libraries would not have been feasible without Brian Richard's help in incorporating my quite dynamic feature wish-list into *DMoct* and also for explaining the mysteries of LAGER to me.

The theme of asynchrony that underlies the hardware and software module implementations in this thesis owes a lot to Jane Sun who helped clarify many aspects of the literature in this area. The subtleties of the formal techniques for designing asynchronous logic, which I picked up from discussions with Jane, definitely made me a better hardware designer - something I will always be

reminded of whenever I wear my hardware designer's hat in the future. Sam Sheng deserves special thanks for cheerfully explaining to me, even late at night, the subtleties of using Macintosh. Without his help the actual writing of this thesis would definitely have taken much longer. Of course, the games he gave me for the Mac also provided me with nice breaks from those long writing sessions in FrameMaker.

All the members of *bjgroup*, but most of all my cubicle mates Anantha Chandrakasan, Jane Sun and Kevin Komegay, made it fun to be in Cory Hall for all these years. Thanks Anantha, Jane and Kevin for being such nice friends, and for bearing with my mood swings and messy desk!

I am grateful to the other members of my thesis and qualifying examination committee besides my advisor - Professors Ron Fearing, David Culler, and Charles Stone - for providing me with valuable feedback during my qualifying examination and later. In particular I would like to thank Ron Fearing for explaining many aspects of robotics at various times. In addition Professor Jan Rabaey, although officially not on my committee, provided valuable feedback on numerous occasions - in fact he has been like a second advisor to me whenever Bob was away.

Finally, I would like to thank Mom and Dad for their love, encouragement, and patience while I stayed away from the real world - without which I would not be where I am today. I would also like to thank my sister Mukta and brother-in-law Gyanesh for making my transition to the American culture smooth, and for providing a home-away-from-home.

CHAPTER 1

INTRODUCTION

Computer systems are ubiquitous in the modern society, however most of these computers are not general-purpose computers such as PCs or the workstations. These specialized computer systems are embedded in communication devices, consumer electronics and appliances, industrial process controllers and automobiles. They perform dedicated tasks while interacting with the user and the physical world in real-time using a variety of sensors and actuators. Such *application-specific* computer systems are being applied to increasingly diverse and complex applications. Unlike general-purpose computer systems, the computational and I/O architecture of these application-specific computer systems are tailored specifically for the task to be performed. Further more, these application-specific computer systems are often very sensitive to non-recurring design cost and time. All this makes the design of such systems complex, and necessitates design methods that are different from those used for general-purpose computer systems.

1.1 Motivation and Objective

This thesis explores computer-aided methodologies for the design of these application-specific

computer systems (from now on the word system will be assumed to imply a computer system). Computer-aided design (CAD) tools are essential to making it cost effective for a designer to tailor the computational and I/O structure by easily exploring the design space, and for this reason much research has been done developing such tools, and design-methodologies based on them. However, this work has been concentrated on the design of individual application-specific ICs, or ASICs. Although these ASICs - or chips with dedicated architectures - are important, they nevertheless rarely constitute a complete system by themselves. Real-life systems are composed of a mix of software running on general-purpose programmable hardware, ASICs and other dedicated hardware, electromechanical components, and mechanical interconnect and packaging. They are distributed across multiple packaging levels including chips, MCMs, PCBs, backplanes, and even networks.

This "chip focussed" research in CAD has largely ignored taking a *systems* perspective of the design process. While the CAD tools for ASIC designs are in a fairly mature state - in some application domains it is even possible to completely synthesize an ASIC from a high-level behavioral description in a matter of hours [Rabaey91][Micheli90][Shung91] - CAD methodologies for dedicated systems have not kept pace.

A unified approach that encompasses these three key aspects of system design - *software*, *hardware*, and, *mechanical* - is essential. What makes this problem interesting is that these three domains are not isolated - design trade-offs need to be made across these domains. For example, a computation can be performed either by software running on programmable hardware or directly by dedicated hardware, dedicated hardware components require system software drivers to communicate and synchronize with them, and mechanical properties of interconnect affect the system partitioning and communication protocols.

Thus a need arises for tools that will enable a designer to explore this complex design space, and to rapidly prototype an application-specific system.

This thesis investigates one part of this problem by describing a unified CAD framework for the rapid-prototyping of software and hardware for application-specific systems that span multiple printed-circuit boards and are composed of off-the-shelf as well as custom, programmable as well as dedicated components. Instead of addressing the ASIC design problem (which we will consider “solved”), the higher level aspects of system design such as multi-chip design issues present at the board level, system software, representation and simulation of mixed software-hardware-mechanical systems, are investigated.

1.2 Problem Formulation

The word *system* is not a very precise term - one person’s system is another person’s component. This makes it difficult to state the exact scope of this thesis. This linguistic difficulty is bypassed here by describing some existing systems to exemplify the class of systems which are being targeted. These examples are then used to identify and abstract the key problems in rapid prototyping of such systems.

1.2.1 System Examples

Philips Compact Disc-Interactive System

The first-generation Compact Disk-Interactive (CD-I) player introduced by Philips in late 1991 is a good indicator of the type of devices people would routinely interact with in the near future. The CD-I integrates compact disk quality audio with full-motion video, still pictures, text, graphics and animation to create a multi-media environment through which a user can navigate using a remote control or a mouse. The required data and program is stored on a compact disk (CD).

Figure 1-1 shows a block diagram of the hardware specified by the CD-I standard. The system is based around Motorola’s 68000 family of microprocessors (the first CD-I player uses MC68070 and the future players will use MC68340 micro-controller which is based around a MC68020 core)

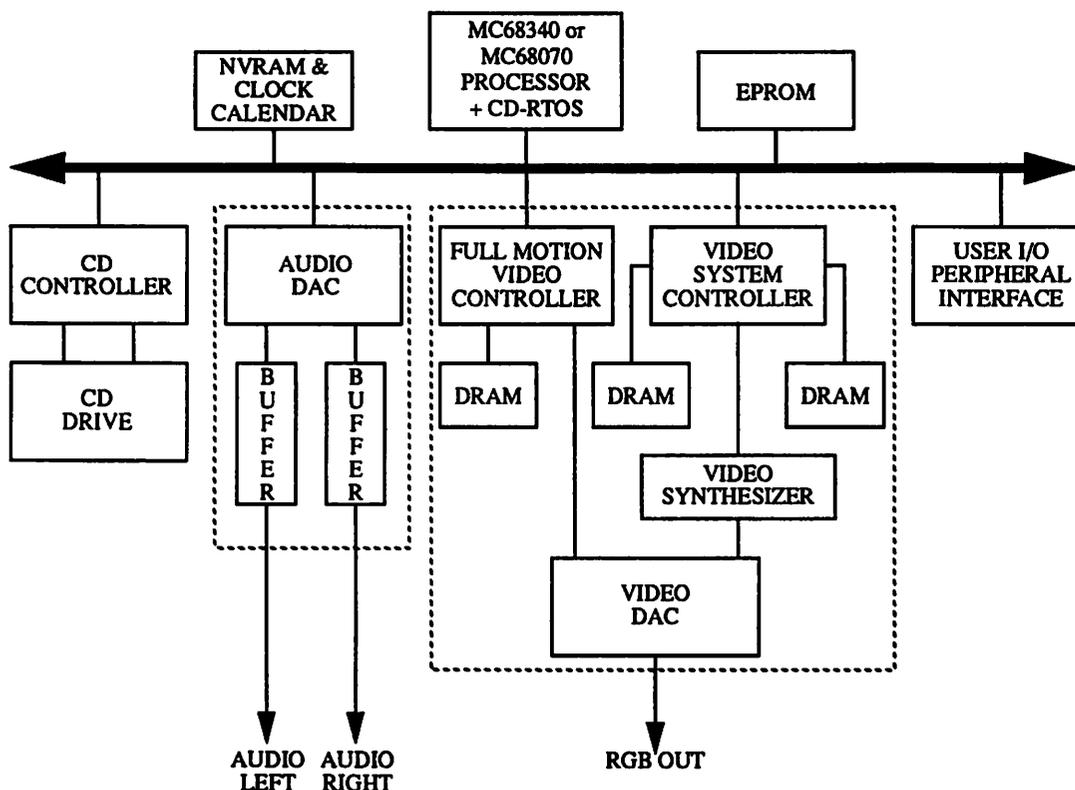


Figure 1-1 : Block Diagram of a CD-I System

running CD-RTOS (Compact Disc Real-Time Operating System), a special real-time operating system derived from OS-9 real-time operating system. The microprocessor is the data manager that controls the data flow around the system and interprets CD-I commands in real time.

Surrounding the microprocessor are dedicated hardware modules, each handling a different task. A CD driver interface and control module runs the CD mechanism and recognizes the different types of data from the disk. The control information is separated from the data information (audio, video and text), and the latter is routed to the audio and video modules. This is done under the control of the CD-RTOS which handles tasks such as the synchronization of the audio and video data. The audio module does data decompression and other signal processing before outputting it to an audio amplifier and speaker system. The video module is composed of two sub-modules. The first sub-

module (not available yet) produces full motion video from MPEG compressed video data. The second sub-module handles still pictures, text, graphics, and animation. The video output from the two sub-modules is combined in a video DAC before being sent to a display. The audio and video modules are based around ASICs. The last module handles user interaction through a variety of peripherals, such as infra-red remote controls, joysticks, and keyboards.

The CD-I system is characterized by a heterogeneity of computation models in the different parts of the system, with a variety of real-time constraints. At the top level the system is reactive, or event-driven, in nature. The software running on the microprocessor responds to user input and to the data coming from the disk. The interaction with the user has soft constraints on the latency of the response, whereas the interaction with the disk has hard constraints on latency and throughput. The audio and the video modules on the other hand do primarily signal processing computation and are best modelled in a data flow fashion with hard throughput and latency constraints, and high data rates requiring dedicated hardware.

The CD-I player currently marketed by Philips does not include the full motion video MPEG decompression unit. It is otherwise a complete implementation of the functionality described above, although the interactive response time of the player is distinctly slow. This version makes extensive use of off-shelf general-purpose chips resulting in a machine that is quite bulky. A set of ASICs for implementing the various modules in the CD-I standard architecture in a more efficient manner is said to be now under development. This suggests that a design philosophy was used where the board level design was done separately from the chip design, probably because of time to market reasons. Essentially, the current implementation appears to be a prototype using off-the-shelf components of the eventual implementation. The use of a system design environment where the board and chip level design are tightly coupled can result in a smaller and higher performance design while minimizing design time.

A goal for the work described in this thesis is to be able to rapidly design systems with such a mix

of software programmable and dedicated hardware, heterogenous computation models, and real-time constraints.

CMU Direct Drive Arm II Control System

The second example describes a research oriented robot control system from CMU for their Direct Drive Arm II project. The system, shown in Figure 1-2, is characterized by a reliance on

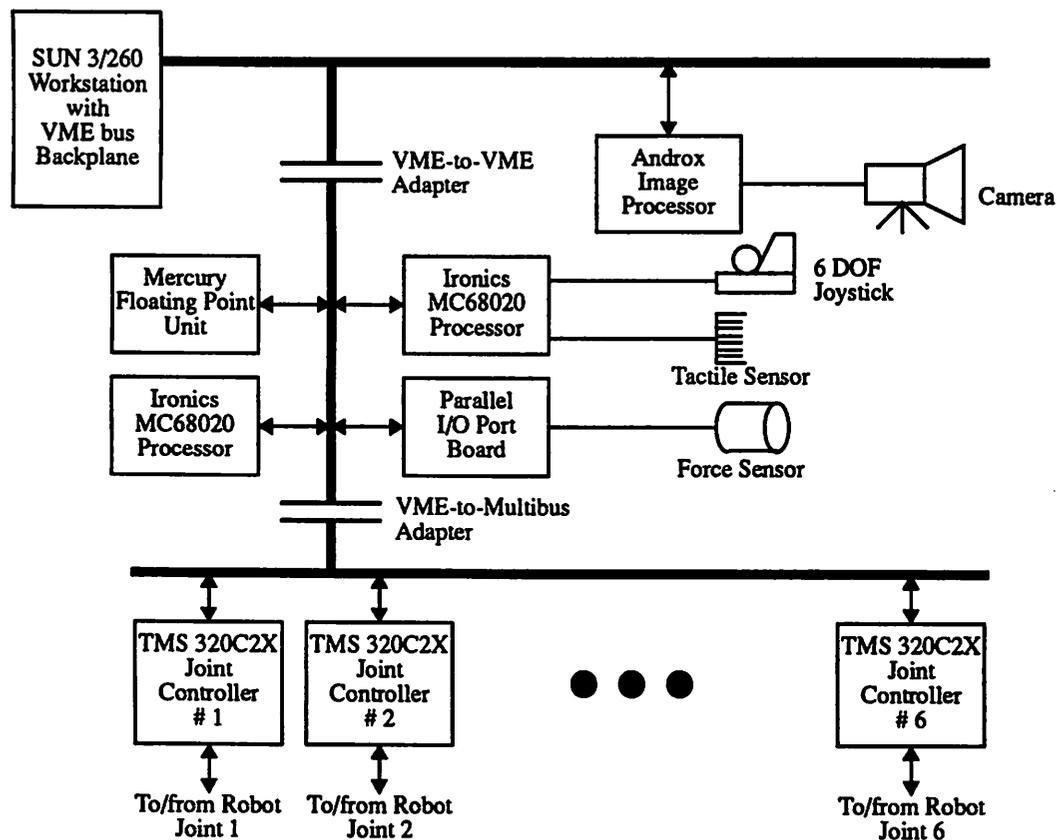


Figure 1-2 : Architecture of CMU Direct Drive Arm II Controller

off-the-shelf commercial boards. A VME bus based SUN 3/260 is the host with a VME-to-VME bus adapter to isolate the timesharing host from the real-time system. Multiple Ironics MC68020 boards reside on the second VME bus and are controlled by Chimer II, a real-time multiprocessing operating-system. A Mercury 3200 Floating Point Unit with a peak performance of 20 MFLOPS also resides on the second VME bus. Six cards based on Texas Instruments TMS320CX DSP

processor reside on a Multibus backplane, each controlling one joint of the CMU Direct Driver Arm II. The Multibus is connected to the VME bus through a VME-to-Multibus adapter. Multiple sensors are connected to corresponding I/O ports: a tactile sensor to an Ironics serial port, a six axes force sensor to a parallel I/O board, and a camera with monitor to a Androx Image Processing board. A six degree of freedom joystick is also connected to an Ironics serial port.

Although much more complex, this system has attributes similar to the previous example - heterogenous computation models and many real-time constraints. However, the emphasis on off-shelf hardware has resulted in an inelegant and complicated implementation. For example, both VME and Multibus backplanes are used because of the limitations imposed by the commercial boards when a single VME backplane would have sufficed. This results in an unnecessarily complicated electrical and mechanical hardware organization. The disadvantage of reliance on off-shelf boards is further demonstrated by the fact that an entire board is devoted to a simple function such as I/O port for the force sensor. On a custom board this functionality can be implemented in less than a few square inches. The use of off-the-shelf boards for I/O also results in a much looser coupling of I/O devices with the processors where the I/O data is consumed or produced, which of course has a negative effect on the performance besides complicating the software for controlling the I/O.

Other System Examples: Speech Recognition and Robot Control

Two other example systems which demonstrate the types of systems of interest in this thesis are a real-time speaker-independent continuous speech recognition system [Stölzle91] and a multi-sensory robot control system [Doshi89][Srivastava92]. Both these systems are complex real-time systems whose performance requirements precluded the use of off-shelf general-purpose computers.

The speech recognition system models speech generation as a hidden Markov model, and formulates the recognition process as a search for the most likely sequence of states of the Markov

model in a trellis of states using the Viterbi algorithm. For real-time recognition of a 60,000 word vocabulary this requires 200,000 states to be evaluated every 10 ms. This task is beyond the capability of general-purpose processors as demonstrated by the fact that the best available implementations using general-purpose processors are able to achieve speaker independent, continuous speech recognition in real-time for vocabularies of around 1000 words. As a result, the implementation of the system makes extensive use of custom boards as well as custom chips, with the overall system being coordinated by control software running on a general-purpose computer under the control of a real-time operating system.

The multi-sensory robot control system is required to control in real-time a six degree of freedom robot arm using data from a variety of sensors, such as position/velocity sensor, force/torque sensor, proximity sensor etc. Both the position of the robot arm, and the forces it applies to the physical environment, have to be controlled using hybrid force-position control algorithms. As evident from the example of the CMU robot controller presented earlier, the use of general-purpose hardware results in an unwieldy and inefficient design. A goal in this robot controller was to implement the entire controller and sensor I/O in just one or two boards making use of custom chips and boards. Given the complex event-driven nature of the controller many of the tasks need to be carried out in software whereas custom hardware is helpful in efficient interfacing to the diverse sensors and actuators.

The design of both these systems could be aided immensely by a methodology that encompassed the design of chips, boards, and software. In fact, a version of the robot control system, and a part of the speech recognition system were successfully implemented using the design framework described in this thesis. Both these designs are much more compact and higher performance than the existing designs that had been implemented using off-the-shelf boards and chips. Chapters 8 and 9 will describe these systems and their design using the tools presented in this thesis.

1.2.2 Problems in System Design

The above examples are representative of the application-specific systems that are the focus of this work. In the following discussion the primary characteristics of such systems are abstracted so as to define the key problems that occur in their design.

These systems continually interact with their environment - they are *reactive* systems. The environment can be a human user, an I/O device, a mechanical system such as the robot etc. This interaction is usually subject to timing constraints defined by the environment which requires *real-time* responses. These timing constraints can be either on the latency of a response or on the throughput. A heterogeneity of computation models is also exhibited by these systems. For example, at the top level they are to a large extent *event-driven*, continuously having to react to external and internal stimuli. But on the other hand they often contain signal processing subsystems that are best described in a *dataflow* model. Similarly parts of the system may have real-time constraints, while the other parts may be non-real-time. Being able to specify, simulate, implement, verify, and test such systems is the key issue in the design process. Some of these problems are studied in this thesis in an attempt to automate as much of the design process as possible using CAD tools, and as will become evident, modularity and reusability also play an important role.

• Most of these systems can naturally be decomposed into communicating asynchronous and concurrent components. The system architecture - both hardware and software - also reflects this decomposition. Therefore issues related to concurrency, communication, and synchronization, play an important role in system design at all levels from specification to hardware and software implementation.

Except for high performance systems where they do not have a choice, the designers of such systems shy away from the use of dedicated hardware in favor of general-purpose processors. The system is thus reduced to a software system - the CMU robot control system described above

being an example. However experience at chip level has amply demonstrated that ASICs with dedicated architecture often simplify the implementation of an algorithm, as opposed to software running on a processor, because they make it easier to meet real-time constraints on latency and throughput, as well as constraints on area and power. For example, real-time I/O interaction with the environment, which usually complicates the design of software systems, is often simplified with the use of dedicated hardware. Chip-level CAD tools such as LAGER [Shung91] have made ASIC implementations of algorithms competitive to software implementations in terms of design time. However tools to give system level designers a similar ability to easily explore and prototype alternative system implementations with different mixes of dedicated and general purpose hardware are not available. The problems that need to be solved for this include partitioning the functionality between software and dedicated hardware, generating the required hardware and software modules, and interconnecting the hardware and software modules.

Another set of problems in the design of these systems are related to their robustness and manufacturability. Problems in design for noise, mechanical packaging and interconnection, thermal analysis etc. fall under this category. Although this thesis makes no attempt to automate these aspects of the design process because they are still largely an art, these issues are discussed at various places in the thesis. Run-time environment and user-interface present some of the other problems in the design process of these systems.

1.3 Previous Work

This work draws on a rich history of separate work for computer aids for design of ICs and for software engineering (CASE). The previous work in these two distinct areas are presented in an attempt to isolate techniques that will also be useful at the system level. In addition results from the current research in the nascent area of Hardware-Software Co-Design will also be given.

1.3.1 Computer Aids for Hardware Development

Most of the work in computer aids for hardware design has been done at the chip level where CAD tools are available at all levels of design - layout tools for physical design, silicon assemblers and compilers for structural and register-transfer level (RTL) design, and synthesis tools for behavioral level design. In contrast board level design is still almost exclusively done in terms of low-level structure (schematic of chips) and physical layers.

MICON

One board-level CAD tool described in the literature that allows design at a higher level is the microprocessor "configurer", or Micon, system from CMU [Birmingham89][Birmingham92]. It is a CAD environment that uses an AI expert system to generate a single-board computer. The computers designed by Micon consist of a microprocessor, ROM, SRAM or DRAM, cache memory if supported by the processor, serial and parallel I/O, standard bus interface, and circuitry for support functions such as address decoding, clock generation etc. The user specifies the type of the microprocessor, amount and type of memory, and the number and type of I/O devices. The knowledge about the components and the microprocessor system structure in the rule base is then used to generate a design satisfying requirements on board size, cost, and system reliability. Although Micon raises the level of design, its use is restricted to single-board computers with a restricted architecture.

CAD Systems for ASIC Design

At the chip level many university as well as commercial integrated CAD systems are available that allow ASICs to be generated starting from high-level structural, RTL, or behavioral descriptions. LAGER [Brodersen92][Rabaey86][Shung89][Shung91], CATHEDRAL [Rabaey88], Bit Serial Silicon Compiler [Jassica85], System Architects' Workbench (SAW), BLIS [Whitcomb92] and Olympus [Micheli90] are some such CAD environments. Among these the LAGER CAD environment is one of the most versatile. The board-level hardware generation

aspects of this thesis are in fact based on LAGER. The core of LAGER is a silicon assembler that allows a high-level structural specification of an ASIC in terms of parameterized functional modules and modules specified behaviorally as a set of boolean equations. Such a structural specification is usually needed for very high data rate applications and for I/O interfacing applications. On top of this silicon assembly core several behavioral synthesis and mapping tools are built. HYPER [Rabaey91] takes a dataflow description of an algorithm in a language called Silage and uses datapath and control synthesis techniques to generate an ASIC with hardwired control. It is suited for medium data rate applications with simple control and low resource sharing. Many DSP applications fall in this category. C-to-Silicon [Thon92] is a behavioral mapper which takes a procedural description of an algorithm in a C-like language called RL and generates an ASIC with microprogram control using an architecture template (called KAPPA) with user specified datapath and a retargetable compiler. It is suited for control dominated, low data rate applications. Another tool built on top of the silicon assembler is a FIR filter generator called FIRGEN [Jain91]. It generates a FIR filter structure from a frequency domain specification. There are other such special-purpose behavioral mappers built on top of the LAGER silicon assembler. This notion of separating the ASIC design into two phases - one that generates an architecture (high-level structural description) from a behavioral specification, and the other that generates the physical layout from the architecture - with the two phases communicating via a well-defined structural description interface is a key feature of LAGER. The parameterized module library and the design manager are the other important features of LAGER. Some of these prove to be applicable at the board level too.

1.3.2 Computer Aids for Software Development

On the software side, many computer aids are available that enable structured development of a complex software system. Traditionally these computer-aided software engineering (CASE) tools automate structured or object oriented techniques, use diagrams and textual specifications to specify system requirements, and specify the architecture of the software implementation. While

some of these are just environments for structured code development and documentation, others allow the system to be specified using a formal textual or visual/graphical language according to an abstract computation model and then generate code for a programmable general-purpose uniprocessor or a homogeneous multi-processor computer. Examples of commercially available CASE packages include StateMate from i-Logix, Matrixx/SystemBuild from Integrated Systems Inc., SES/workbench from Scientific & Engineering Software Inc.

Some recent research work, such as GRAPE [Lauwereins90], GABRIEL [Lee89], and McDAS [Hoang92], has extended CASE to real-time applications in digital signal processing (DSP). Starting from a block diagram or language based description, software is generated for a homogeneous multiprocessor. Real-time constraints are met by using computation models and scheduling strategies that take advantage of the synchronous dataflow characteristic of DSP algorithms. The special characteristics of DSP algorithms allow these CASE tools to provide a much more vertically integrated and automated environment than general purpose CASE tools. GABRIEL and GRAPE use a library of primitive code segments whereas McDAS relies on translating data flow graph segments to C during the code generation process.

Some of the recent CASE tools have started to integrate hardware modelling and specification by trying to establish a link between tools that help automate software development to those that automate hardware development. This is typically being done by allowing modelling of hardware components, and by generating VHDL or other HDL code for simulation and/or synthesis. StateMate from i-Logix and SES/workbench 2.0 from Scientific and Engineering Software Inc. are commercial tools with such capabilities. Ptolemy, which is the successor to GABRIEL and discussed in the next section, also allows simulation of hardware components together with specification for the rest of the system.

1.3.3 Computer Aids for Concurrent Development of Hardware

and Software

As mentioned in the previous section some of the CASE tools provide a loose coupling between the development of hardware and software components of a system. However some recent research has attempted to explicitly address the problem of concurrent development of the hardware and software, although in a more limited context than in this thesis.

Ptolemy

Ptolemy [Ptolemy91] is a multi-paradigm simulator framework that uses object oriented software techniques to allow simulators with radically different computation models (such as synchronous dataflow, dynamic dataflow, discrete event) to be tightly coupled. This allows different parts of a system to be simulated according to computation models that are best suited for the respective part. These parts can then be connected or one hierarchically embedded in the other.

One of the supported computation models, called *domain* in Ptolemy, is the *THOR domain* which provides the ability to model digital hardware. This allows low level hardware implementation of part of the system to be simulated with rest of the system which may be modelled at a higher level of abstraction in a different domain for later mapping into software. Further more, the THOR model library contains interfaces to cycle-by-cycle simulators provided by manufacturers of some processors (such as MC5600 and MC96002 from Motorola). These simulators allow the simulation of actual execution of programs on hardware using those processors.

These features of Ptolemy provide an excellent framework for supporting the simulation phase of a system design that may contain both hardware and software components. Although in its present form it is still not versatile enough to handle all scenarios of mixed hardware and software simulation, some recent work has reported using Ptolemy for the simulation of an echo canceller where the execution of software running on a MC5600 based hardware is simulated together with an A/D converter [Kalavade92].

Although Ptolemy provides a code generation facility for parts of the system modelled in the Synchronous Dataflow domain, it neither supports real-time software as encountered in event-driven reactive systems, nor does it have any capability to support the actual implementation of the hardware.

Vulcan-II

This is a CAD tool which is currently under development at Stanford [Gupta92a][Gupta92b] and provides the ability to map an algorithm described in Hardware-C to multiple ASICs and one microprocessor such that part of the functionality is implemented in software. It uses the architecture model shown in Figure 1-3. The architecture model in Vulcan-II is based around one software programmable processor and multiple ASICs, all of which are masters on a shared bus which also has a common memory. The system is implemented by partitioning it into dedicated hardware on the ASICs and dynamically scheduled program fragments, or *threads*, executing on

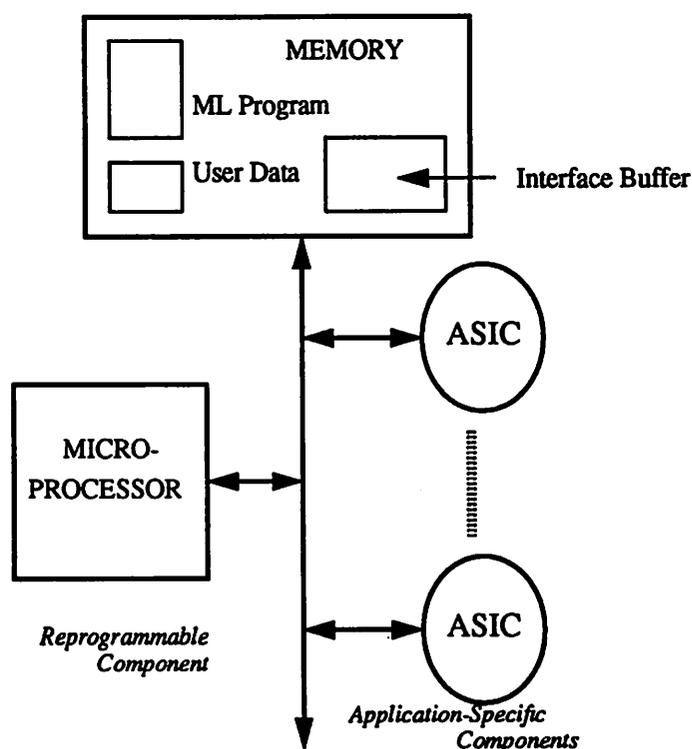


Figure 1-3 : System Architecture Model of Vulcan-II (Figure 3 of [Gupta92a])

the processor. The common memory not only contains the program and data memory for the processor but also an interface buffer memory using which the ASICs communicate with the software programmable processor. The interface buffer has two parts: a FIFO buffer for storing tags for scheduling the threads and an associative memory based data buffer for storing the actual data being transferred from the ASICs to the processor with the data being keyed according to the tag.

Instead of addressing the problem of implementing a complete system, Vulcan-II primarily addresses the problem of partitioning the control and data flow graph corresponding to the Hardware-C description on to this architecture model while meeting bus bandwidth constraints. The existing Olympus ASIC design system [Micheli90] can be used to implement the ASICs, and C code is generated for the software component. However many important problems that will occur in a real implementation of the architecture model have been ignored in Vulcan-II. These unaddressed problems include:

- a. implementation of the multi-master bus protocol and logic
- b. generation of ASICs which have the required bus-master interface logic
- c. implementation of the microprocessor sub-system hardware
- d. implementation of the communication memory sub-system which is required by the architecture model to have a complicated tag matching capability
- e. generation of the board to actually realize the multi-chip system

The architecture model itself is quite restrictive as it allows only one microprocessor, requires the ASICs to be clock synchronous, and does not allow any interaction between the software executing on the microprocessor with the rest of the world. A key weakness of the Vulcan-II architecture model is the existence of a single shared bus used not only for communication with the ASICs but also for program and memory access for the software running on the programmable processor which results in a severe bottle-neck. In addition, the architecture requires the processor as well as all the ASICs to be bus masters. Although [Gupta92a] and [Gupta92b] do not mention the implementation details, such multi-master busses invariably have complex protocols and

require complicated interface logic. Such multi-master busses also have access arbitration overhead which not only makes effective bandwidth low but also difficult to estimate. The interface buffer memory is also a shared resource and a bottle-neck. The implementation also requires an expensive associative memory based data buffer. All these factors appear to make the architecture model not only not scalable but also difficult to implement.

The Vulcan-II model also only allows for one programmable processor. While this makes the partitioning problem amenable to automation, it restricts the flexibility of the software programmable part of the system. Software programmable processors range from simple micro-controllers to general-purpose microprocessors to specialized digital signal processors, and they are suited for different kinds of applications. A single system might use general-purpose microprocessors to handle software for user interface, networking etc., signal processors for numerically intensive computation, and microcontrollers for control intensive software. An architecture model that allows a single software programmable processor does not allow the system to take advantage of these specialized programmable processors. In addition, the single programmable processor is shared between all the ASICs, and can easily become the performance bottle-neck.

This restrictive architecture model is needed for the partitioning algorithm used by Vulcan-II to work. The architecture model appears to be driven by the need to make an automated partitioning algorithm feasible as opposed to requirements of real systems. The partitioning algorithm itself is a greedy algorithm that first maps all nodes with data-dependent delays, such as a data-dependent *for* loop, into software and then greedily moves nodes from hardware to software until the bus bandwidth is exceeded. The nodes mapped in software are then implemented as a finite-state machine driven by data arrival from the ASICs. The bodies of nodes, such as the data-dependent *for* loop, are required to have statically known delays, and are scheduled cyclically. Using the statically known delays a maximum bound is found for the rate of invocation of each node from which the bus bandwidth required is calculated, Unfortunately, this algorithm will perform poorly

if the nodes are unbalanced in terms of the amount of data they produce per invocation. Further, the partitioning ignores the effect of synchronization delays on the available bus bandwidth. In a multi-master bus these delays can consume a large fraction of the bus bandwidth.

In summary, Vulcan-II models the system design problem as primarily one of partitioning, and uses a restricted simple system architecture model to make the partitioning problem tractable.

1.4 Summary

From the discussion in the previous section it is clear that little work has been directed towards treating custom hardware and software in an integrated fashion for system design. This is surprising in view of the obvious conceptual symmetries between hardware and software. For example, the notions of modularity, reusable parameterized libraries are present in both hardware and software. The behavior specification paradigms used in the two domains demonstrate remarkable similarities. Further, many techniques can easily be extended from one domain to the other. For example, techniques for inter-process communication and synchronization in software operating-systems prove to be applicable in hardware domain as well.

The following chapters describe a computer-aided design framework for dedicated board-level systems which uses a unified view for hardware and software components that make up a system.

Chapter 2 gives an overview of the design framework. Chapters 3 and 4 describe the generation of hardware and software modules respectively. The techniques and tools presented in these two chapters are not specific to any particular application domain. Chapters 5 and 6 describe techniques for architecture mapping that while not being universally applicable provide a complete vertical trajectory for designing a large class of systems. Chapter 7 gives a view of the tools, libraries, and models described in the earlier chapters from the perspective of a user. Chapter 8 and 9 describe the design of a robot control system and a part of a speech recognition system using the design framework described in the earlier chapters. Chapter 10 concludes the thesis by presenting ideas for future research in this area of system design that is still young and not very well defined.

CHAPTER 2

SIERA SYSTEM DESIGN FRAMEWORK

A system design can be represented according to three main views or levels of abstraction - *behavioral*, *structural*, and *physical*. The behavioral representation specifies the system functionality in some suitable textual or visual formalism, the structural representation specifies the system architecture in terms of the hardware/software modules and their logical connectivity, and the physical representation specifies the actual physical implementation as a spatial relationship between the component themselves and the environment. These levels of abstraction are also valid at the chip level where efficient design environments are available for transforming a behavioral or structural representation to a physical one. However, as discussed in the previous chapter, this is not true at the system level where CAD aids are relatively primitive.

A system design environment would allow a user to represent the behavior or structure of a system, together with constraints, and transform the initial representation to generate a physical implementation. For such a design environment to be realized the problems of how to represent a system at each of the three levels of abstraction, and how to transform a higher level representation to a lower one need to be solved. Simulation at each level of abstraction, and verification of one

level against another are also very important problems. These problems are explored in this thesis in context of the class of real-time reactive systems described in the previous chapter. As this thesis will show, solutions to some of these problems exist or are easily extrapolated from experience at chip level, others require new techniques, and still others remain to be solved. The results of this investigation have been integrated in a system design environment called SIERA which embodies a true system design methodology. SIERA also encompasses work done by co-researchers who are looking into other aspects of the system design problem. An overview of SIERA is presented in this chapter, and the specific contributions of this thesis are detailed in the following chapters.

Note that there is nothing fundamental about dividing the design process into the two phases of behavioral-to-structural and structural-to-physical transformation. The system design process is really one of continuous refinement from higher level representations to lower level ones. The particular division described here was found to be very useful in the LAGER chip design environment because of the relative independence of the two phases, and it appears to be useful at the system level too. It is possible to subdivide each of these phases further. For example the structural-to-physical transformation at the chip level is often viewed as a two step process of going from high-level structure (architecture or RTL description) to low-level-structure (gate level description) and then to physical.

2.1 Overview of SIERA

The overall organization of SIERA reflects its roots in LAGER [Rabaey86][Shung91][Brodersen92] - the custom chip design environment developed over the last six years. One of the key features of the organization of LAGER was the separation of the behavioral-to-structural transformation phase from the structural-to-physical transformation phase. Since the behavioral-to-structural transformation is a difficult problem even at the chip level, this decoupling allowed several special purpose tools to be developed for it while providing

a general-purpose silicon assembler for doing the structural-to-physical transformation. A structural interface, described in detail in Chapter 3, was established in which the architecture of a chip can be generated from a high-level tool or can be manually specified. SIERA has a similar top level organization with distinct subsystems for the behavioral-to-structural transformation, structural-to-physical transformation, and system testing. The scope of this thesis is restricted to the first two aspects of SIERA, and the reader is referred to [Kornegay91] for information on how support for system testing is integrated into SIERA. In addition [Sun92a] describes ALOHA, a specialized module generator available in SIERA for the synthesis of interface logic between the hardware modules in a system. Figure 2-1 shows the top-level view of SIERA and contrasts it to that of LAGER.

Based on Figure 2-1 one can identify several problems that need to be considered in SIERA:

- a. Behavioral representation of systems
- b. Simulation of behavioral representation
- c. Structural representation of systems
- d. Simulation of structural representation
- e. Physical representation of systems
- f. Physical simulation of systems
- g. Behavioral-to-structural transformation
- h. Structural-to-physical transformation

These problems in system design are set apart from the corresponding problems in chip design by two factors. First, systems are capable of exhibiting much more complex functional behavior than an ASIC. Second, the implementation medium is much more heterogeneous in case of systems than in the case of chips. For example, one has to trade-off between dedicated hardware and software programmable hardware, and between off-shelf chips and custom chips.

As a result of these differences, solutions to many of the above problems for chip design just do not apply or scale to system design. At the behavioral level the complexity and heterogeneity

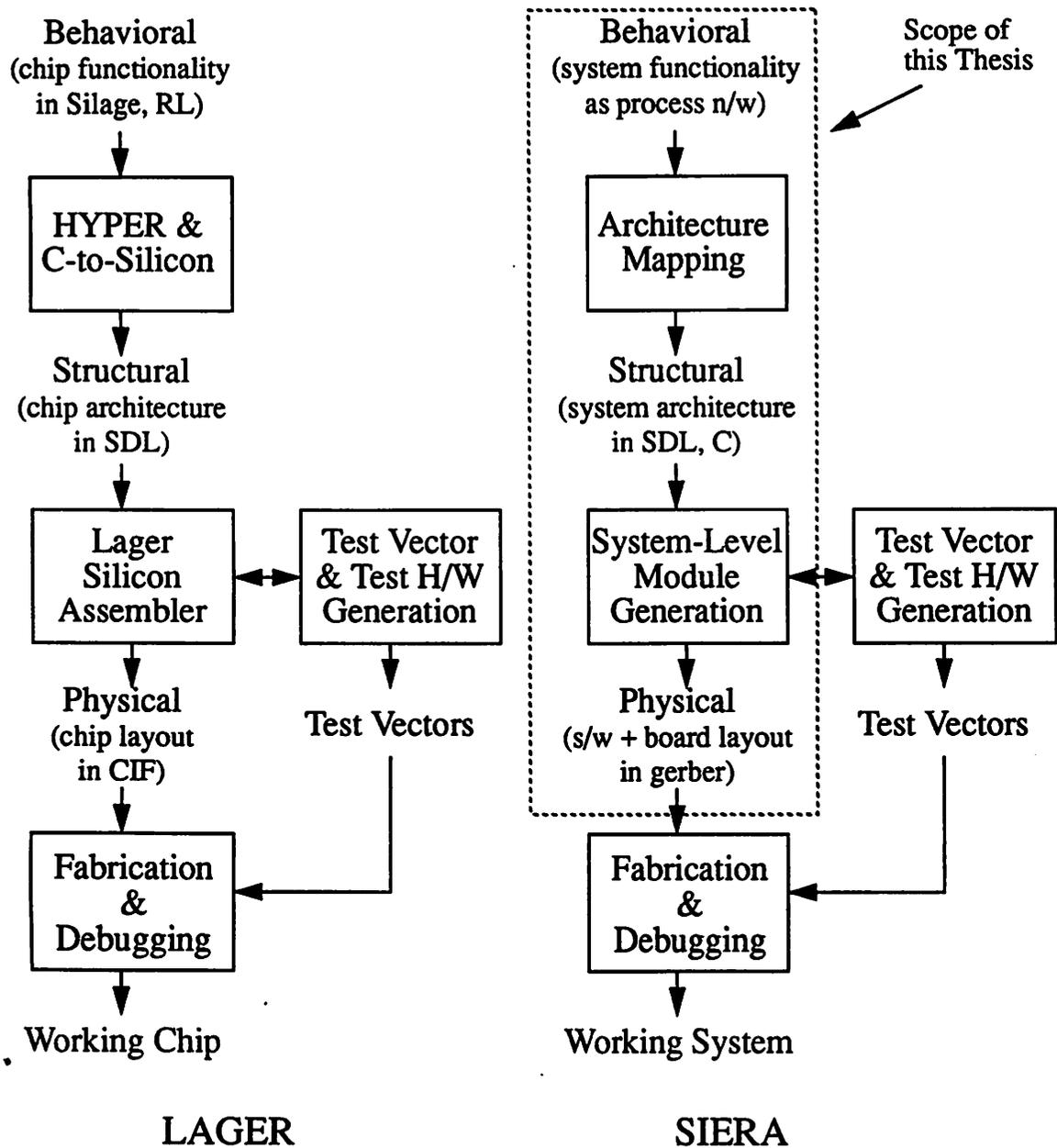


Figure 2-1 : Simplified Top Level Views of LAGER and SIERA

present in systems make it difficult and unnatural to represent and simulate an entire system according to a single computation model as is usually done in the case of chips. The simple architecture model of a single controller and a datapath as used for chips is not adequate for most board level systems. Hardware implementation that is synchronized to a single clock is usually

adequate for a single chip but not for an entire system. The software issues are not present in a chip design as ASICs mostly have hardwired controllers. These and some other similar issues are addressed later in the thesis.

On the other hand some of the system design problems can be adequately solved using techniques that are prevalent at the chip level. For example, as shown in chapter 3, the concept of a reusable parameterized module library together with a suite of module generators as originally used in LAGER for ASIC generation works well for board designs too.

Finally, some system design problems, such as physical representation, extraction and simulation, are not addressed here. This is not because adequate solutions to them always exist - in fact physical representation of boards and simulation using that information are in a primitive state compared to those for chips due to the prevalence of the use of prototyping as the typical strategy for system debugging.

2.1.1 System Design Methodology in SIERA

The research presented here describes the solutions provided in SIERA to the various problems in system design that were outlined above. The overall goal of the work was to develop within the SIERA framework a vertically integrated design methodology that supports all stages involved in the development of real application-specific systems from high level description to board and software generation. It was decided to adopt an “application-driven” approach for this, as opposed to the “tool-driven” approach adopted by many researchers, such as the Vulcan-II system described in Chapter 1. Critical importance was attached to the usability of the design methodology to actual applications. This necessitated an approach based on studying real-life example systems, developing an initial design methodology, and then gradually refining and automating the design methodology through experience obtained by actually implementing these example systems. By contrast, a “tool-driven” approach attacks the problem from the other end -

emphasis is placed on initially developing a set of automated tools, and then developing a design methodology around it.

Figure 2-2 shows a more detailed block diagram of the SIERA environment, emphasizing the design methodology provided by it for designing application-specific systems. The design of a system is divided into two distinct phases - *Module Generation* and *Architecture Generation*.

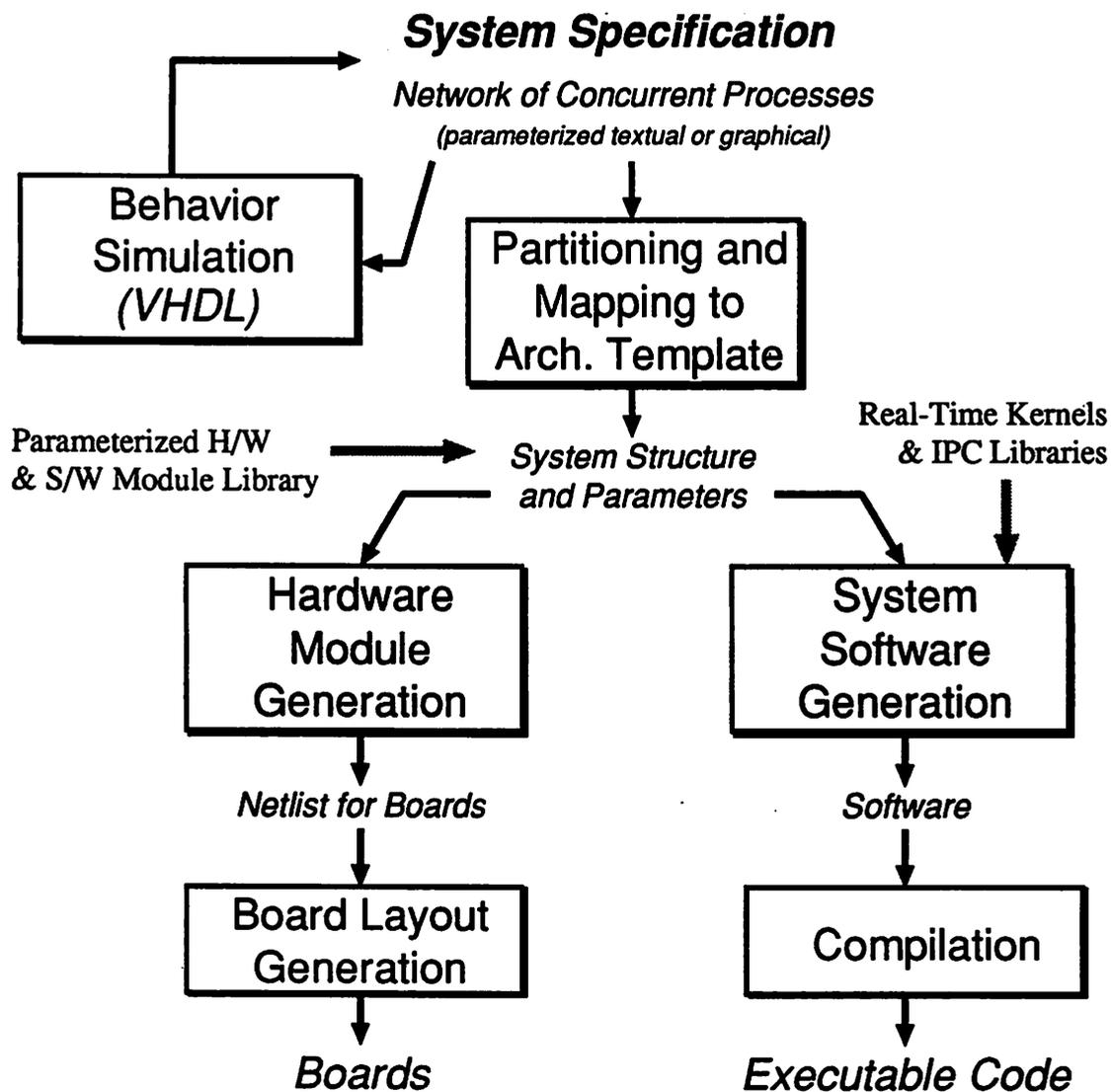


Figure 2-2 : System Design Using SIERA

Module Generation

Module Generation refers to the physical implementation of a system given its architecture as a composition of hardware and software modules. The sub-problems of *Hardware Module Generation* and *Software Module Generation* are addressed separately. In hardware module generation the focus here is on generation of multi-chip board level hardware modules - the corresponding problem at the level of a single chip is considered solved. As detailed in Chapter 3, a structure description language, database policies, and a design manager same as those in LAGER are used to provide a seamless integration of board and chip level hardware module generation. A variety of module generators are provided for automatically generating hardware modules from mixed structure/behavior descriptions, and for automatic placement according to several different strategies such as abutment and simulated annealing. A key component of the hardware module generation strategy is a library of reusable parameterized board-level sub-system modules. Such sub-system modules include complete software-programmable processor modules that can be embedded on a custom board, and modules for communication, signal processing, and data acquisition. These modules can be customized for a given application through parameters provided by the module designer. For example, processor modules typically allow the amounts of different types of memories to be set through parameters. The sub-system library together with the hardware module generators provides a environment that is much more sophisticated and versatile than commercially available tools for board design.

The output of the hardware module generation phase are netlists and placement information describing the custom boards in the system. A bit-level simulation of the hardware can be done from this netlist using the THOR simulator and the behavioral models for each chip on a board. Finally a commercial router is currently used to do the routing. The output is a set of files in the GERBER format that are used for the board fabrication.

Unlike hardware module generation, the problem of software module generation is not present at

the chip level. Initial effort was devoted towards defining an appropriate form of software modules. As discussed in Chapter 4, the *process* construct was chosen as the form of software modules that minimizes inter-module dependencies while exploiting the inherent concurrence provided by the ability of the hardware module generators to create boards with multiple heterogenous software-programmable processor modules embedded in them.

Instead of addressing the problem of automatically generating the code implementing a software process from a high level description of its functionality, effort was directed towards developing the underlying system software substrate that is needed to support processes. An approach where an autonomous real-time kernel is run on each of the software programmable processor module was chosen. One or more software modules - processes - are mapped to each of these processor modules, and are then managed by the corresponding kernel. Each kernel thus allows a single processor, capable of a single thread of execution, to be shared between multiple software modules, each with its own thread of execution. A set of software libraries has been developed to let the software modules communicate with each other, and with dedicated hardware modules attached to the software programmable processors.

In addition to the development of the system software itself, effort has also been directed towards providing various kinds of run-time services for the software modules. These include initialization and down-loading of the software modules, terminal and file I/O to a host workstation, and remote-procedure call based communication. Together these services provide a consistent and powerful software environment for custom boards developed using this design methodology.

Architecture Generation

Architecture Generation refers to the process of generating a suitable architecture for the system starting from a high level description. As mentioned earlier these problems are difficult to solve in the general case - experience with chip design suggests that a multiplicity of co-existing solutions

is desirable. In order to provide a vertically integrated design path for systems of most interest, a limited solution to the architecture level problems is currently provided in SIERA.

There are two sets of problems at the architecture level. The first problem is the representation used to describe the system functionality, and the simulation of this representation. As discussed in Chapter 5, the system is described here as a network of concurrent processes that communicate using single-reader and single-writer FIFO channels that connect input and output ports of processes. The ports are characterized by a protocol that specifies the behavior if the corresponding channel is not ready for communication, and the channels are characterized by an associated buffer depth. No unified language has been developed to describe the functionality of the processes themselves. Instead, a multiplicity of languages such as VHDL and C can be used for this purpose. This is done because of two reasons. First, a unified language, capable of capturing the diverse computation models found in typical systems, appears to not be the desirable solution from a designer's standpoint (though advantageous for the tool developer). Second, it was decided to concentrate primarily on the problem of communication and synchronization between the processes in a system, and not on the problem of synthesizing an architecture for implementing the processes.

A key feature of the process network model is that the notion of processes is equally applicable to hardware and software modules that constitute a system. In addition, the physical environment of a system can also be modelled by processes that run on their own dedicated hardware.

A set of VHDL packages is provided to allow the process network for a system to be simulated using standard VHDL simulators. In addition to emulating the channel based communication between processes, these packages also provide the capability to simulate continuous-time modules described by differential equations. This is useful for modelling the sensors, actuators and mechanical sub-systems found in the physical environment of most systems.

The second problem at the architecture level is that of generating a suitable architecture to

realize a system described using the process network model. An architecture template based approach is used for this where a process network description is partitioned onto an instance of the template. The architecture template defines a scalable and parameterized organization of hardware and software modules in a system. In accordance with the “application-driven” philosophy, the emphasis is placed on arriving at a suitable model for the architecture template. The problem of partitioning is not addressed - it is handled manually.

The architecture template specifies the organization of the dedicated hardware modules and software-programmable processor modules in a system in a multi-layered hierarchy of busses, as shown in Figure 2-3. The layering in the architecture template reflects the fact that systems have processes with different real-time requirements. Software processes that require user interaction and sophisticated a software environment are mapped to a workstation that constitutes the top layer. Other software processes with increasing real-time requirements are mapped to a single-board computer in layer 2, and to layer 3 processor modules on custom boards. All these software-

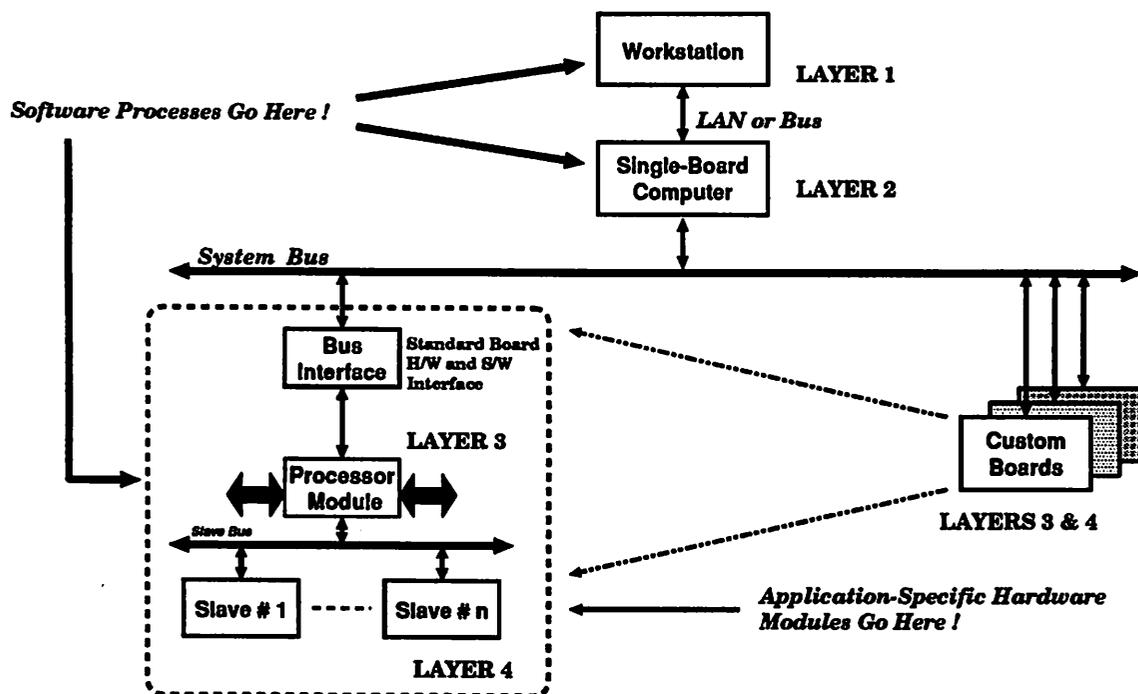


Figure 2-3 : Multi-Layered Architecture Template for System Architecture Generation

programmable processors run real-time kernels to allow easy mapping of multiple software modules (processes), as described earlier in this section. Dedicated hardware processes, on the other hand, are mapped to hardware modules in layer 4 that are memory-mapped slaves to layer 3 processor modules. At each level in the template the processor in the upper layer acts as a master to the processor in the layer below.

In addition to specifying the overall organization of the system architecture, the template also specifies the interfaces between the processors to implement communication and synchronization between processes mapped to different processes. This is accomplished by relying on two important concepts - *asynchrony* and *layering*. The processors run asynchronously, and are not required to have a global timing reference or clock. This simplifies the hardware implementation as problems such as clock skew are minimized. Any synchronization that is required between two processors is established, when needed, using the interface module linking them. *Layering* is used to implement the channel based inter-process communication in the process network description by using simpler communication and synchronization primitives to construct the channel objects. A suitable set of these simple communication and synchronization primitives have been defined for all possible cases of inter-processor communication in the architecture template. Chapter 6 investigates these issues in detail, and also describes reference implementations of interface modules for inter-processor communication.

The layered architecture template, while not universal, has proven applicable to a diverse set of applications. In general it is applicable where the coarse granularity flow of information in the system follows a hierarchical organization with the required bandwidth and real-time requirements decreasing and the functional/logical complexity increasing as one goes up in the hierarchy. Systems that interact with a user while controlling a device or processing a signal in real-time often reflect such an organization.

Thus a complete, though not fully automated, methodology for designing application-specific

systems is provided in SIERA. The work here focuses on board level hardware module generation, development of the system software substrate, high-level representation and simulation of the system behavior, development of a formal model for the system architecture, and techniques for inter-module communication and synchronization. The problems of automatically generating the code for software modules from a high-level description, and automatically partitioning the process network description onto an instance of the architecture template have been left for future research.

User's View of the System Design Process

Figure 2-4 shows the system design process in SIERA from the perspective of a user. The entry point is a description of the system as a network of processes. The system is viewed to be composed of concurrent processes that communicate using FIFO channels with appropriate protocols. Such a description can currently be written in VHDL, using the packages described in Chapter 5. At this stage all the processes are equal - no decision is made about whether the process will be implemented in hardware or software.

Next this process network description in VHDL is manually partitioned onto an instance of layered the architecture template. The output of this step is a file describing the system architecture in the SAIL (System Architecture Intermediate Language) format. The syntax of the SAIL file is described in detail in chapter 6. Its primary purpose is to store information about processors in the layered architecture model, their interconnection topology, the parameters of the inter-processor interconnect links, and a many-to-one mapping of processes in the process network description to processors.

In effect the SAIL file views the system to be made up of two types of entities: *processes* and *processors*. The *processes* are connected arbitrarily, although the interconnections follow the channel mechanism specified by the process network model. The *processors* are connected according the interconnection topology and interface modules allowed by the layered architecture

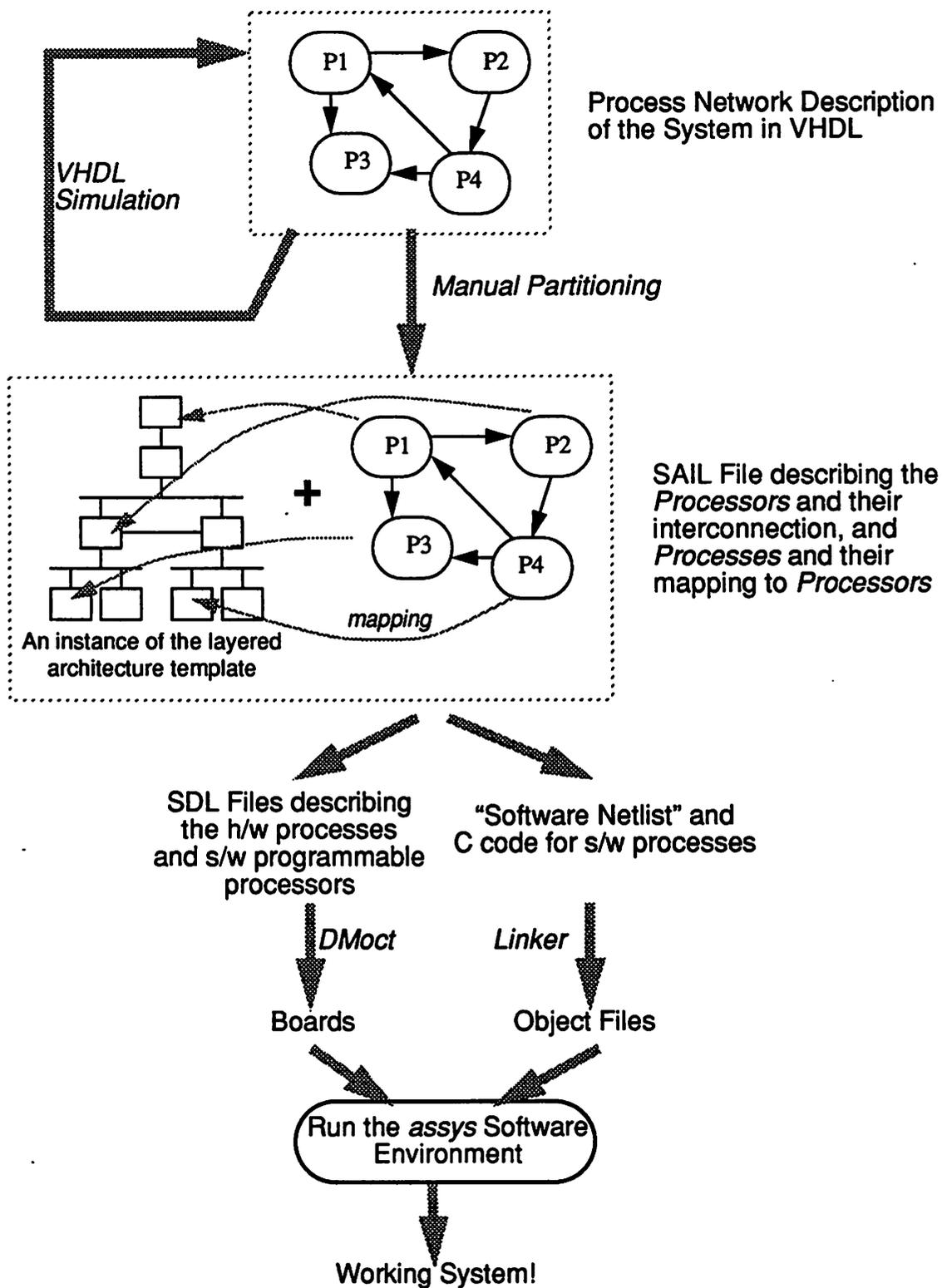


Figure 2-4 : User's View of System Design in SIERA

template. The *processors* can either be dedicated hardware modules, or software-programmable hardware modules. A dedicated hardware module can implement only one specific process. A software-programmable module, on the other hand, can simultaneously implement multiple processes. The *processes* are mapped to the *processors* taking this constraint into account.

Once the SAIL file is generated, the system design process splits into two parallel paths. The information about the *processors* in the system, and their interconnection, is extracted from the SAIL file. From this the SDL files describing the hardware organization of each of the custom board in the system are generated. Then various module generators and libraries are used to physically implement each custom board. Simultaneously, the mapping of *processes* to software-programmable *processors* in the SAIL file, together with information about the *processors* and their interconnection links, is used to configure the real-time operating-system kernels and software modules to be loaded into each of the software programmable processor. This process is essentially one of linking the code for implementing the software processes to that of the kernel for each processor. The output of this process is a set of *object files*, one for each software programmable processor, that can either reside in a PROM or be loaded at the run time.

After the boards have been fabricated, the run-time utilities provided by the software environment described in Chapter 4 are used to initialize the system, down-load object files into the processors, and interact with the system through terminal and file I/O from the workstation in layer 1 of the architecture template.

2.2 The Philosophy Behind SIERA

The previous section gave a global picture of the SIERA design environment and the following chapters will address some of its components in depth. However an understanding of the philosophy behind SIERA is essential in order to appreciate these details. The primary goal in SIERA to aid the designer in developing the board-level hardware and software in a very short

time. Together with the chip-level design tools and board and chip fabrication facilities with a short turn-around time this implies that a complete multi-board system can potentially be developed in a few months by a single designer. The strategy adopted to accomplish this goal is in de-emphasizing low-level design optimizations, such as board area or code size, in favor of optimizations at the architectural or the behavioral level. The various tools and libraries that constitute SIERA are targeted towards easy exploration of the design-space at such high levels.

Throughout SIERA the emphasis is on facilitating the management of the system design complexity by discouraging the designer from adopting *ad hoc low-level* approaches, and supporting the following principles [Strom91]:

- a. **Modularity**: viewing the system to be composed of smaller, interacting modules with well defined interfaces. The modules may be hardware modules or software modules.
- b. **Reusability**: the reuse of same modules in different systems.
- c. **Uniformity**: modules interact in a uniform way using a small number of well-defined mechanisms independent of whether the module is in hardware or software, local or remote. Besides the conceptual elegance, this uniformity also makes it practical to have CAD tools that automate the interfacing of modules.
- d. **Abstraction**: hiding the underlying implementation detail. *Performance transparency*, or the ability to estimate the cost of and to optimize the underlying implementation, is given up in favor of abstraction.

One of the key techniques adopted to enforce the above principles is the extensive use of libraries which serve as repositories of optimized and reusable modules. These include a library of board level modules that are composed of multiple chips and implement sub-system level functionality, and a library of software modules. The policies associated with each library enforce the uniformity of its member modules. In order to minimize the size of these libraries extensive use is made of parameterization. For example board-level hardware modules such as processor modules are designed so that the memory size, I/O interface and other attributes can be varied by passing appropriate parameters when the module is used.

Another idea that has proven useful in enforcing the above principles is that of asynchrony - the

modules execute asynchronously with no global timing reference (clock), and synchronize when needed using handshaking. This theme underlies the hardware and software architectures used in SIERA. Asynchrony naturally encourages modularity by making a module independent of the speed of computation in another module. In addition it simplifies hardware and software implementation by avoiding global timing references. This is particularly important because the target systems are physically distributed over multiple boards making global synchronization difficult.

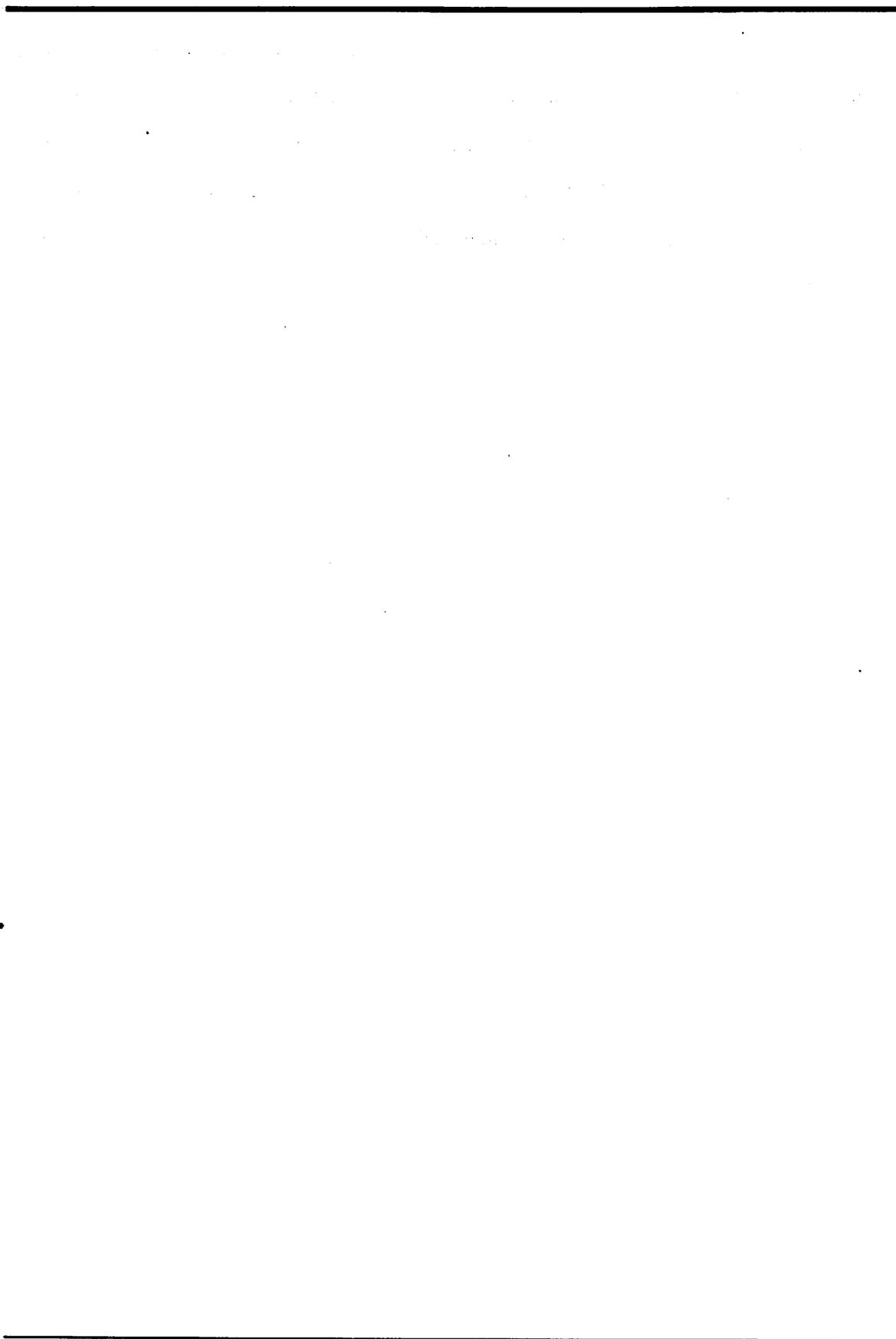
Since ASICs are an important part of most systems, seamless integration of the board level hardware design with that at the chip level is also considered important in SIERA. This is accomplished by using the same database mechanism and policies as in LAGER. As a result the structure description language SDL and the design management tool DMoct that were used in LAGER for ASIC design are also usable for structural design in SIERA, and an identical design environment is presented to a designer at the board and chip levels.

SIERA is an open system in that new modules can be added to libraries and new tools can be added to manipulate the design. This is intended to be done by expert users who are familiar with the low-level design issues in hardware or software design, the design constraints, and the underlying database policies for tool integration.

2.3 Summary

The main problems in the design of multi-board systems were identified and an overview of a system design environment called SIERA was presented which has its roots in the LAGER chip design environment. The design of a system in SIERA begins with a high-level description of the system as a network of asynchronous processes that communicate using FIFO channels. After the high-level functionality of the system has been verified using a process network simulator, the description is mapped to a parameterized layered architecture template that contains dedicated

hardware modules as well as software programmable processor modules connected using a hierarchy of busses. The result of this step is a SAIL file that describes the architecture of the system as an interconnection of processors and a mapping of processes to the processors. The physical implementation of the system hardware (boards) and software (object files) is then accomplished by parallel steps of hardware module generation and software module generation using information contained in the SAIL file together with hardware and software modules contained in various libraries.



CHAPTER 3

GENERATION OF HARDWARE MODULES

Module Generation refers to the task of converting the description of the system architecture in to a physical implementation. In the case of ASICs Module Generation is often called Silicon Assembly (note that we use the term 'Module Generation' in a much looser sense than is conventionally used at the chip level). At the system level, particularly the type of multi-board systems that are the focus of this work, there are two aspects to the module generation problem - hardware and software - which is due to the duality in the term system architecture. From a hardware perspective the term architecture refers to a set of hardware modules (processors) interconnected in a specific topology. From the software perspective the term architecture refers to a set of software modules (processes) that communicate in a specific topology, and are usually mapped many-to-one to the hardware modules. A complete physical implementation of a system means that both the hardware and software view of system architecture be implemented. This gives rise to the problems of hardware module generation and software module generation. These two problems cannot be completely isolated because the software implementation will at some level invariably depend on the hardware implementation, but for the sake of organization the

present chapter deals primarily with the hardware module generation problem, and the following chapter with the generation of software modules. The connection between these is covered in Chapter 6.

An important point that needs to be emphasized is the separation of the module generation phase from the architecture generation (i.e. the interconnection of the modules). The former problem is easier in many ways, and one expects the design aids and methodologies for it to be relatively general-purpose. This separation of the two phases allows the module generation aids to be used directly by the user, or as a back-end by other higher-level tools for architecture generation.

3.1 Overview of Hardware Module Generation Environment

A major goal in the development of the hardware module generation environment was to handle arbitrary hardware architectures implemented as custom boards using both off-shelf as well as custom chips, while providing an environment for quick generation of alternative implementations. Since boards can in turn be composed of chips, the hardware module generation problem spans these two levels of packaging - board, and chip. Many mature design environments are available for the generation of application-specific chips. Therefore, this work concentrates on the module generation issues that are present only at the board level. However, as discussed below, the structure description language, database and design management framework used by the LAGER Silicon Assembler are also being used for this board-level hardware module generation environment. As a result the chip level module generators in LAGER, as well as chip level behavioral tools that are built on top of LAGER are available to the board designer in a seamless fashion.

One question that arises is that since board design tools are already available commercially, is there really a need for the hardware module generation environment described here? Existing

board design tools typically provide the ability to place and route a netlist of chips. However hardware design at such a low level is not sufficient, as is indicated by the experience at the chip level. Describing a complex board as a hierarchical netlist of chips is extremely cumbersome - it is equivalent to designing a chip using a library of primitive gates such as NAND, OR etc. The answer at the chip level has been to raise the level of abstraction, and view a chip to be composed of high-level modules that either come from a library of parameterized reusable modules (adders, multipliers, RAMs etc.), or are synthesized from a behavioral description (FSM controllers, decoders etc.). This two-pronged strategy has proven very successful for the design of ASICs because it raises the design abstract to a sufficiently high level while providing the benefits of reusable and possibly hand-optimized library modules, as well as the use of synthesis where appropriate.

Board level design tools have lagged behind this use of higher levels of abstraction. The rest of this chapter is a description of how these lessons learned at the chip level have been applied to the board level hardware module generation problem. Of course, different layout and synthesis tools, and libraries are needed at the board level, but the underlying philosophy is the same as that adopted in LAGER for the chip level.

The key requirements for realizing the above design philosophy are an extensive module library, a variety of layout and synthesis tools, and a design management framework. Our solution to this consists of concentrating on the first two requirements (library and tools), and adapting the existing chip-level design management framework provided by LAGER.

3.2 Design Management Framework

The design management framework has three key components:

- a database with a procedural interface for information access by the tools
 - standard policies to represent various types of design information in the database
-

- a design manager to provide orderly access to the database by the user and the tools

In the following subsections these three components are described with emphasis on the board-level aspect, which is an extension of the chip design approach used in LAGER [Brodersen92].

3.2.1 OCT Database

OCT is an object-oriented database for electronic CAD applications which offers a simple mechanism for storing persistent as well as transient information about the various aspects of an evolving design. With appropriate policies it has proven capable of handling designs ranging from a single transistor to multi-board systems and of representing information ranging from low-level physical information to high-level structural and even behavioral information. It provides procedural interfaces in various programming languages, such as C and C++, using which a program can store and retrieve design information. The actual storage mechanism is hidden from the program.

OCT is based on object-oriented principles. The basic unit in a design is a *cell* which can, for example, represent a chip or a board. A cell can have multiple *views* which are used to represent different aspects of a cell. For example, one may have a physical view of a board which would specify the layers and the copper geometry on them. Similarly one may have a schematic view of the board which would contain information about its netlist structure. OCT does not specify what views a cell may have and what is contained in those views - these issues are decided by an appropriate design representation policy chosen by the tools. A view can have multiple *facets* which represent different abstractions of its contents. The contents of a view are defined by a required facet named '*contents*'. Finally, there can be multiple *versions* of a facet.

Cells in OCT may be hierarchical - they can contain instances of other cells. For example, the cell corresponding to a board may contain instances of cells corresponding to the chips on the board.

The basic unit in OCT that is edited by a program is a facet. It contains a collection of other objects

which can be classified into geometric objects (layer, point, box, circle, path, polygon, edge and label), interconnection objects (terminal and net), annotation objects (property and bag), hierarchy objects (instance) and change list objects (change list and change record). Geometric objects can be used to represent physical geometry. Nets and terminals can be used to represent interconnections - a very important aspect of electronic designs. Terminals provide a means to represent information flow between different levels of design hierarchy. A terminal is associated with either a view or an instance. In the former case they are called *formal terminals* and in the latter case they are called *actual terminals*. The actual terminals are automatically created along with the instance. The formal terminals are created in the contents facet of the view - the other facets only refer to them. A common policy for representing connectivity is by attaching the interconnected terminals to a net which can then be viewed as a node in an electrical design. The annotation objects bags and properties are used to specify new types of composite data objects that are not directly supported by OCT. This is particularly useful because the object structure of OCT is otherwise static. In fact the design representation policies make extensive use of properties and bags.

Relations between these objects can be defined by *attaching* one object to another. An OCT facet is essentially a directed graph of these objects. Through its procedural interface OCT provides mechanisms for storing and navigating such a directed graph, for creating, modifying, deleting, and accessing the component objects, and for creating or destroying attachments between objects. However, it does not say what such a directed graph *means* - this is defined by the policies.

3.2.2 Policies for Design Representation in OCT

An OCT policy specifies what directed graph structures of OCT objects are legal and more importantly what are the semantics of these legal structures. A view following a particular policy must be defined by a collection of OCT objects whose attachments are correct according to the policy. A standard set of policies is provided with OCT - schematic, symbolic and physical. The

schematic policy [Octtools91] represents the design in an abstract way showing the subcells of a cell and their connectivity. The symbolic policy [Octtools91] adds information about subcell size and shape, placement and the implementation of the interconnection. The physical policy [Octtools91] represents the design exclusively in terms of geometric shapes with no connectivity information. Every view has an associated policy and it is often the case that the view name is the same as the policy name. In addition, the policies are independent of the implementation technology. The technology details are encapsulated in a separate technology database which is actually just another OCT cell. Every view usually also has an associated technology.

These standard policies were developed for non-parameterized chip design and proved inadequate for parameterized chip design in LAGER. New policies that were extensions to the standard policies were developed in LAGER to represent the design of a parameterized chip at different stages of the design flow. However, even these policies required further extensions to handle system level issues as described in the following sub-sections.

The *structure_master* Policy

This policy is used to represent the design as a parameterized structural hierarchy. Parameterization is an efficient way for representing a class of modules in a well-defined manner and enhances the reusability of designs. For example, a memory module can have the word width and memory size as parameters. When a parameterized module is instantiated, each parameter is bound to a specific value.

A hierarchy of OCT views following the *structure_master* policy is the initial representation of the design in OCT. As a result this policy is closely related to the textual and graphical languages used by the designer to represent the design. In fact this policy almost has a one-to-one correspondence to the language *SDL* (Structure Description Language) which is a textual language for representing the structure of a parameterized design. The OCT view of a cell following the *structure_master* policy is usually named 'structure_master' and is created from a corresponding graphical

schematic or a SDL file. If the cell name is *mycell* then the SDL file is called *mycell.sdl*.

The purpose of the *structure_master* policy is to represent parameterized structure. A cell has terminals for communicating with the outside world - these terminals are called *formal terminals*. In addition a cell can contain subcells with the structure of the cell being defined by the interconnection of these subcells. The subcells are actually instances of other master cells. These instances automatically have terminals attached to them that correspond to the formal terminals defined in the corresponding master cell - these terminals are called *actual terminals*. A set of formal and actual terminals can be attached to a net to indicate that those terminals are connected.

The *structure_master* view of a cell can have parameters defined for it. These parameters are called *formal parameters* and are represented in the OCT view by defining a property object with the same name as the parameter inside a bag named FORMAL_PARAMETERS contained by the facet. The value of each property is an optional string containing a lisp expression. The value if present represents the default value of the parameter which can be overridden by binding a different value to the parameter when the cell is instantiated at a higher level in the design hierarchy. The order of definition of the parameters is important because the value of a parameter can be defined in terms of the values of parameters defined earlier. When a cell is instantiated as a subcell in another cell which is higher up in the design hierarchy, values can be bound to its parameters by attaching a string valued property with the same name as the parameter to a bag named ACTUAL_PARAMETERS which is in turn attached to the instance object. The string value of the property is a lisp expression defined in terms of the parameters of the container cell. There are several special parameter names that are used for uniform module interface and for passing information to tools operating on the design. For example, cells which have a physical package at the board level (such as a 74F00) also have a parameter called PACKAGENAME to specify which of the available packages should be used for a particular instance.

Attributes can be attached to the facet of a cell, to its formal parameters, to its formal terminals and

to nets and instances contained in the facet. These attributes can in general be tree structured with intermediate nodes being bag objects and the leaf nodes being either bag objects or property objects. The optional value of each of these property objects is a string representing a lisp expression in terms of the parameters of the cell. There are several reserved attribute names which have special meaning to various tools. Table 3-1 shows a list of attributes relevant for board level

	NAME	VALUE TYPE ^a	COMMENT
FACET	PACKAGECLASS	constant string	package hierarchy, <i>e.g.</i> PCB
	CELLCLASS	constant string	<i>e.g.</i> MODULE, LEAFCELL
INSTANCE	X	numeric	X translation
	Y	numeric	Y translation
	T	string	orientation, <i>e.g.</i> R90, MXR90 etc.
	POSITION	list of two numbers	(X translation, Y translation)
	ROTATION	numeric	multiple of 90
	OFFSETX	numeric	offset in X in absolute units
	OFFSETY	numeric	offset in Y in absolute units
TERMINAL	DIRECTION	string	<i>e.g.</i> INPUT, OUTPUT, BIPUT
	TERMTYPE	string	<i>e.g.</i> SIGNAL, SUPPLY etc.
	PINNUMBER	integer	pin number on the package

Table 3-1 : A Partial List of Reserved Attributes in *structure_master* Policy Relevant to Board Level Hardware Design

a. Unless indicated as a constant the value of an attribute is a string representing a lisp s-expression which can depend on the parameters of the cell and which must evaluate to the data type indicated.

module generation.

Besides the values of the attributes parameters of a cell can also be used to affect several other aspects of the design. For example, parameterized indexed sets of instances can be created allowing a single description to describe a class of circuits. One and two dimensional arrays are

trivial examples of such indexed sets. Similarly, parameterized indexed sets of nets and formal terminals can be defined with a bus whose width is parameterized being a trivial example. Generation of selected parts of the structure, including instances, nets and formal terminals, can be turned off conditionally. A description of how these are represented in OCT is too detailed to be presented here and is anyway peripheral to this thesis. Interested readers can refer to [Brodersen92] for a detailed description as well as the corresponding SDL syntax.

A key part of the *structure_master* policy is that the connectivity specified in the view can also be parameterized. The name of a net or a terminal really represents the root name of an indexed set instead of representing a singleton. The connectivity is represented by attaching a terminal to a net and defining a mapping between the indices of the associated index sets. The mapping is defined by four optional properties attached to a bag called MAP that is in turn attached to the net as well as the terminal. The WIDTH property is a string representing an integer-valued lisp expression that defines the number of net and terminal pairs that are connected. For each of these connections the indices of the terminal and the net being connected are calculated by evaluating the lisp expression contained in the string-valued NET_INDEX and TERM_INDEX properties after successively incrementing a lisp variable *_i* from 0 through WIDTH-1. For each of these connections a fourth optional property named CONDITIONAL is evaluated. If this boolean valued property evaluates to false the particular connection is not made. The terminal and net indices are in general lists of scalar elements such as integer, float, string etc. This mapping scheme allows arbitrarily complex mapping of terminal indices to net indices. The SDL language provides several syntactic mechanisms for expressing this mapping.

The heavy reliance on using lisp expressions as values is essential to representing parameterized structure using the *structure_master* policy. An approximate subset of Common Lisp called Light Lisp [Octtools91] is used for this purpose. The standard LightLisp environment has been extended by a set of functions useful for expressing parameterized hardware structure and this set of functions is designer extensible.

In addition to the structure of the cell, the *structure_master* policy also allows the view to be annotated with the names of tools that would generate an implementation of the cell. There are two types of tools: tools that modify or even create the structure of a cell and tools that generate the physical implementation. The former are specified by a string valued attribute named `STRUCTURE_PROCESSOR` and the latter by another string value attribute named `LAYOUT_GENERATOR`¹.

The *structure_instance* Policy

This policy is used to represent a non-parameterized design hierarchy. It is obtained from a parameterized design hierarchy following the *structure_master* policy by evaluating all the parameters and all attributes and also expanding all the terminal and net mappings. This is done by making use of a LightLisp evaluator. The resulting *structure_instance* hierarchy represents non-parameterized structure with all the bus nets fully expanded. This policy is similar to the standard OCT symbolic policy except for some minor extensions to reflect the fact that it is evaluated from a *structure_master* hierarchy. In particular, the evaluated parameters are copied into a bag called `FORMAL_TERMINALS` attached to the facet and the `STRUCTURE_PROCESSOR` and `LAYOUT_GENERATOR` attributes are also copied. The *structure_instance* hierarchy of a design contains all the information needed to generate its physical implementation.

The *physical* Policy

This is the simplest of the three policies - its purpose is to represent the physical implementation of the design as a geometrical structure. An OCT view following the physical policy primarily contains terminal, layer and instance objects. Geometric objects such as boxes, circles etc. are attached to the layers. The terminal objects define ports through which information is passed to the

1. The name `LAYOUT_GENERATOR` is an unfortunate legacy of the root of this policy in ASIC design.

next higher level in the design hierarchy. They are implemented by attaching geometric objects (which are also attached to layers) to them.

The layers provide a mechanism to group geometric objects in a technology specific way. The names of available layers and associated design rules are specified by the technology. The meaning attached to layers is technology dependent. For chip and board layouts the layers usually correspond to the masks required for fabrication and the geometries attached to the layers correspond to the two-dimensional shapes drawn on the masks.

A pure geometric representation is often not sufficient because the design may be only partially implemented even after a layout-generator has been run. For example, in a hierarchical design the placement might be done hierarchically but the routing may be done in one shot on the whole design after flattening the hierarchy. As a result retaining connectivity information for unimplemented nets is desirable. This is done in a manner similar to that in the *structure_instance* policy by attaching terminals to nets. This policy extension was made specifically for boards where this approach of doing hierarchical placement but one-shot routing gives good implementations.

3.2.3 Design Manager *DMoct*

The central design manager is *DMoct* which manages access to the design database by the user as well as the tools. It automates the generation of a hierarchical design which may require different tools to create different parts of the hierarchy. It calls the required tools in the correct order while making use of time-stamp checking and parameter comparison techniques to avoid regeneration. Most importantly, it presents the designer with a common frontend to all the tools.

Figure 3-1 shows the design flow as orchestrated by *DMoct*. It transforms the initial hierarchical textual representation of the parameterized design in SDL to a hierarchy of *structure_master* views in OCT. Next, with the aid of a built-in lisp interpreter and a set of tools called *structure-*

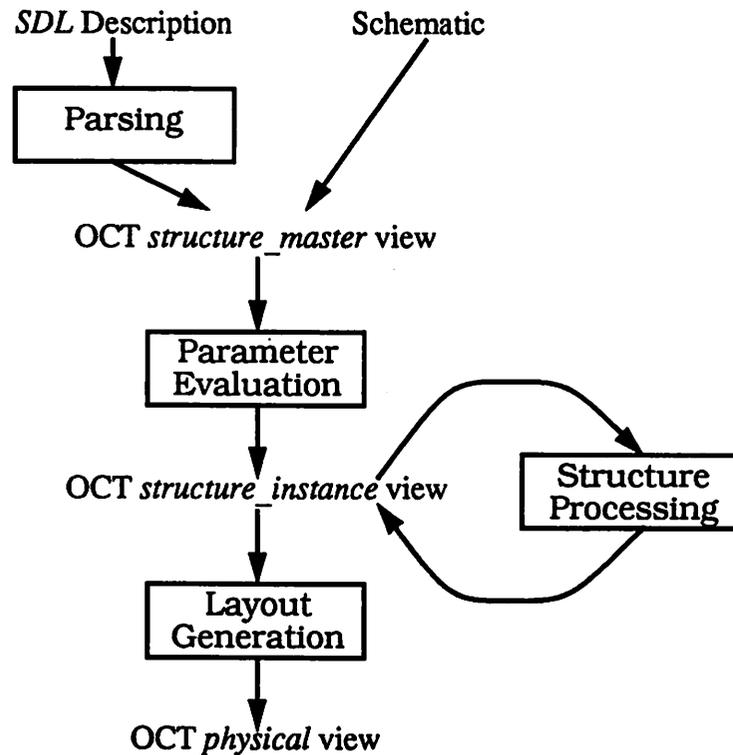


Figure 3-1 : Design Flow using *DMoct*

processors, it creates a hierarchy of `structure_instance` views. This is done by passing parameter values down the `structure_master` hierarchy and creating the `structure_instance` views in a bottom-up fashion by expanding subcell arrays and composite nets or busses. At each stage of the hierarchy after creating the `structure_instance` view *DMoct* calls special tools, referred to as structure-processors, to modify that view or the `structure_instance` sub-tree rooted at that view. Often these tools are synthesis tools that use a behavioral description to generate a net-list that implements the functionality. Section 3.4 discusses some such tools that are available for board-level designs. Optionally *DMoct* can also flatten the `structure_instance` hierarchy at any given point. Finally *DMoct* traverses the resulting `structure_instance` hierarchy in a bottom-up fashion and calls special tools, referred to as layout-generators, at each stage to generate a physical implementation. In case of boards these tools do the placement and routing to create the photoplotter files required for board fabrication. *DMoct* provides hooks to let the designer force the

generation of or ignore the generation of a selected part of the hierarchy, thus by-passing its built-in time-stamp checking and parameter comparison mechanism. A sophisticated search mechanism is used by DMoct and other tools to access files from libraries. The search space is partitioned according to keys that indicate the type of file and a library search path is associated with each key.

Handling of Packaging Hierarchy by *DMoct*

DMoct also handles some special design management issues that arise when designing systems as opposed to single chips. A key distinction between systems and chips is the existence of multiple types of packaging in systems. A chip package, for example a PGA (pin-grid array) package, is usually the lowest level of packaging - it contains a silicon die and uses bonding wires and pins to connect pads on the die to copper traces on a board. For improved electrical performance, spatial efficiency and economic reasons complex systems make use of a variety of packaging levels organized in a packaging hierarchy. According to the industry standard jargon, the chip die is the level 0 packaging. The level 1 packaging is formed by the packages for single chips and discrete parts. MCMs composed of multiple silicon dies that are mechanically and electrically bonded on a substrate using a variety of emerging technologies form what is informally referred to as the level 1.5 of the system packaging hierarchy. Level 2 of the packaging hierarchy is formed by printed circuit boards (or other substrates) which use copper traces in an epoxy laminate to electrically connect lower level packages that are soldered to the boards. Level 3 packaging, usually the top level of the system packaging hierarchy, is the card-cage with a backplane bus or cables to interconnect multiple boards.

Different tools treat a design with a multi-level packaging hierarchy differently. A board placement tool treats MCMs and chips as primitives. This requires that the design hierarchy below these packaging levels (levels 1 and 2) be truncated. Similarly a chip router tool connects modules inside a chip package - this requires that the design hierarchy above the chip level packages (level 1) be ignored. A simulator on the other hand is not concerned with the packaging hierarchy at all -

it is concerned only with the functional hierarchy. This suggests that in addition to super-imposing a packaging hierarchy on top of a design hierarchy, one also needs the ability to extract parts of a design hierarchy that are below, or above, or between, certain packaging levels.

DMoct provides support for handling of packaging hierarchies by following a similar standard for expressing levels of packaging as described above. Design cells that correspond to mechanically packaged entities are required to have an attribute called `PACKAGECLASS` and an optional parameter called `PACKAGENAME`. The value of the `PACKAGECLASS` attribute indicates the level of packaging hierarchy whereas the value of the parameter `PACKAGENAME` is used to select a specific package. For example, the `PACKAGECLASS` may be `PCB` to indicate that the package is meant to be contained by a board and the `PACKAGENAME` may be `DIP20` to indicate a particular package. The existence of a `PACKAGECLASS` attribute indicates that the cell is a mechanical primitive at the level above and that the contents of the cell are encapsulated in that mechanical primitive. Using the `PACKAGECLASS` attribute *DMoct* can be forced to ignore the design hierarchy below a certain packaging level. This is done by ignoring the substructures inside cells that have `PACKAGECLASS` attributes belonging to a specified set of values. Similarly, selected portions of the design hierarchy can be flattened down to a certain packaging level. For example, a multi-board system can be flattened to the level of gates. This is the opposite of partitioning - synthesis and partitioning tools create these packaging hierarchies.

3.3 Board Layout Generation Tools

The term layout generation tool at the board level is used to refer to tools that given a net list of chips or macro modules, do the placement and/or routing. The open architecture of the design management framework provided by *DMoct* allows different layout-generators supporting different layout styles to be easily integrated. Just as at the chip level, different placement and routing algorithms are appropriate for different types of modules. For example, a memory module is very regular, and a tiling based placement based on abutment of subcells might be appropriate

for it. On the other hand, at higher levels of the hierarchy one typically has a collection of polygonal macro blocks, for which a more general-purpose placement strategy is needed.

The set of layout generation tools currently available in this system is small but nevertheless provides capabilities superior to most existing tools. The current set of tools supports a hierarchical placement of the design together with one-shot routing of the flattened and placed design hierarchy. This is accomplished using two tools for placement and a foreign (commercial) router.

3.3.1 psg: Package Symbol Generator

Psg is the layout generator used for board-level modules that have their own package. Examples of such modules are individual chips, discrete parts, MCMs, SIMMs etc. The functionality of *psg* is very simple. It first looks for a package name specification in the module by searching a sequence of attributes. Then it uses the package name to search for physical information about that package in a package library. As described later, for each package this library has the physical geometry information, such as the copper shapes on the various board layers, pin locations, package bounding box and package height. This package information is copied into the `OCT structure_instance` view of the module. In addition the geometry for each pin is also attached to the formal terminal of the module that corresponds to that pin.

The rationale behind *psg* is to encapsulate the physical information about packages in a separate library. Many different chips may use the same package so that a separate package library makes management of the package information easier.

3.3.2 pfp: PCB Floor-Planner

The board floor-planner *pfp* is the most important of the available layout-generators. It is meant for use with modules that are composed of sub-cells. In cooperation with the design manager, *pfp*

allows the sub-modules within a module to be placed according to a variety of methods. The placement can be done in one of several modes:

a. User Specified Absolute Placement

In this mode the position of each sub-cell is specified by the user in absolute coordinates (physical units). This is done by attaching attributes called X and Y to the sub-cells in the SDL file for the module. In addition an attribute named T is used specify the orientation. The attributes POSITION and ROTATION provide an alternative interface for specifying the absolute placement.

b. User Specified Relative Placement

This is similar to the previous mode except that values of X and Y are now treated as relative coordinates. The subcells are then tiled in a row-major or a column-major fashion in as compact a fashion as allowed by their bounding boxes.

c. Automatic Placement

In this mode the placement is done automatically by calling a simulated annealing based placement program called Puppy [Octtools91]. Due to the nature of the algorithm the resulting placement may have overlaps and the parts may not be on routing grid. Consequently this mode is meant to get a good initial placement which is then modified interactively.

d. Interactive Placement

In this mode VEM which is the graphical editor for the OCT database is used to let the user interactively do the placement. A starting placement can be obtained by using one of the other non-interactive modes.

e. Placement Using Previously Saved Floorplan

In this mode a previously saved floorplan file is used to do the placement. This is useful to save and reuse floorplans that are created or modified interactively.

The real power of *pfp* comes from the fact that the attributes X, Y and T for specifying the subcell placement can be arbitrary lisp expressions that can use the parameters of the module. These lisp expressions are evaluated by the design manager *DMoct* using a lisp interpreter. Using this mechanism sophisticated parameterized absolute as well as tiling based placement can be achieved. In effect this allows custom placement algorithms for a module to be embedded in its SDL file.

In addition *pfp* provides options to 'flip' the resulting floorplan horizontally or vertically. 'Flipping' is different from a simple 'mirroring' transformation in that it ensures that the subcells are not mirrored because that is not meaningful on a board. *pfp* has some other options that are concerned with the router constraints, such as ensuring that the pins of the parts lie on the routing

grid. Following is the functionality of *pfp* in pseudo-code:

```

/*
 * X, Y, and T refer to the horizontal translation, vertical
 * translation, and rotation/mirroring transformation respectively
 */

pfp(cell) {
  sourceFacet = cell:structure_instance;
  outputFacet = cell:physical;

  copy sourceFacet to outputFacet;

  foreach s = subcell of outputFacet {
    change master of instance s from a structure_instance
    view to a physical view;
  }

  if (placement_mode == use_floorplan_file) {
    foreach s = subcell of outputFacet {
      get X, Y, and T of s from floorplan_file;
    }
  } else if (placement_mode == do_tiling) {
    L = list of all subcells of cell;
    if (tiling_mode == row_major) {
      sort L according to Y attribute of subcells in L
      breaking ties based on X attribute;
    } else {
      sort L according to X attribute of subcells in L
      breaking ties based on Y attribute;
    }
  }
  foreach s = subcell from sorted list L {
    if (s == first element of list L) {
      cursor.x = 0;
      cursor.y = 0;
      next_cursor.x = 0;
      next_cursor.y = 0;
      col = attribute X of s;
      row = attribute Y of s;
    }
    if (tiling_mode == row_major && row != attribute Y of s) {
      cursor = next_cursor;
      row = attribute Y of s;
    }
    if (tiling_mode == column_major && col != attribute X of s) {
      cursor = next_cursor;
      column = attribute X of s;
    }
  }

  /* translate s so that its lower-left corner is at the cursor */
  let bbox = bounding box of s;
  translate s in x-direction by (cursor.x - bbox.lowerleft.x);
  translate s in y-direction by (cursor.y - bbox.lowerleft.y);

  /* apply offsets */
  translate s in x-direction by OFFSETX attribute of s;

```

```
translate s in y-direction by OFFSETY attribute of s;

/* update the cursors */
if (tiling_mode == row_major) {
  cursor.x += attribute OFFSETX of s + width of s;
  if (next_cursor.y < cursor.y + attribute
      OFFSETY of s + height of s) {
    next_cursor.y = cursor.y + attribute OFFSETY
      of s + height of s;
  }
} else {
  cursor.y += attribute OFFSETY of s + height of s;
  if (next_cursor.x < cursor.x + attribute
      OFFSETX of s + width of s) {
    next_cursor.x = cursor.x + attribute OFFSETX
      of s + width of s;
  }
}
}
} else if (placement_mode == automatic_placement) {
  prepare input for PUPPY simulated annealing based placement tool;
  run PUPPY;
} else {
  foreach s = subcell of outputFacet {
    get POSITION and ROTATION attributes of s if they exist;
    default values are (0,0) and 0 respectively;
    convert values into OCT units, and then translate and rotate s;
  }
}

if (horizontal_flip_flag == 1) {
  /*
   * flip cells horizontally about the vertical axis
   * without mirroring the cells
   */
  let v = x-ccordinate of the vertical axis of cell;
  foreach s = subcell of outputFacet {
    let llx = lower left x-coordinate of s;
    let width = width of s;
    translate s in x-direction by -2*(llx - v)-width;
  }
}

if (vertical_flip_flag == 1) {
  /*
   * flip cells vertically about the horizontal axis
   * without mirroring the cells
   */
  let h = y-ccordinate of the horizontal axis of cell;
  foreach s = subcell of outputFacet {
    let lly = lower left y-coordinate of s;
    let height = width of s;
    translate s in y-direction by -2*(lly - h)-height;
  }
}
```

```

}
close outputFacet;

if (interactive_mode_flag == 1) {
    run VEM to let the user do interactive placement;
    foreach s = subcell of outputFacet {
        save (s, X, Y, T) in floorplan_file;
    }
}

if (cell == design_root) {
    dump placed netlist for foreign router;
}
}

```

3.3.3 oct2rinf: Interface to Foreign Router

At present there is no built-in board router available in the framework. Instead commercial routers are interfaced for routing. At this time only the router from Racal-Redac has been integrated through an interface provided by *oct2rinf* which can be used as a layout generator on its own, or can be accessed via a special flag to *pfp*. Since the commercial router cannot do hierarchical routing², *oct2rinf* first flattens the design hierarchy and then converts the resulting flattened OCT netlist into an ascii file format called *rinf* that is used as input by the commercial router. Besides this format translation, *oct2rinf* also produces part list information and detects some types of errors in the netlist.

A one-way interface to commercial routers is by no means sufficient - one also needs the ability to obtain physical information generated by the router and put in the standard OCT database so that tools such as extractors can access it in a uniform fashion. This task is made difficult because there is no standard interchange format or database that is used by the commercial routers. There are four approaches to solve this problem:

- a. *Routers that use the same database as the other tools (OCT in our case)*

2. This seems to be typical of all available PCB routers.

This approach is idealistic because it would be difficult to make the vendors to adopt a common database. The alternative would be to write our own board router, which is not trivial due to the complex nature of the design rules for board fabrication technologies.

b. *Routers that can read and write a common interchange format for board physical information*

This too is idealistic but probably more palatable to the commercial vendors - after all common interchange formats are already used elsewhere, for example CIF for chip physical information and EDIF for netlist information.

c. *Extract physical information from GERBER files*

Most commercial routers can generate a set of *GERBER* format files which contain the drawing commands for the plotter that generates the photo masks for the various layers of the board. Although this appears analogous to the CIF files used to describe the layer geometry in case of chips, there is a key difference. Chips do not have a packaging hierarchy and therefore the information contained in a CIF file is sufficient to completely create a chip. Boards, on the other hand, have a packaging hierarchy. The GERBER files only contain information required to create the copper interconnect patterns on the various layers of the board, and no information on the packages or parts being used and their electrical connectivity with the copper patterns on the various layers. In theory the information contained by the GERBER files together with the net-list describing the board, the part placement information, and the physical description of the packages, is sufficient for a complete physical characterization of the board. In practice this would require a complicated matching of the pin locations on each package with the copper pattern specified in the GERBER files.

d. *Dedicated tools for every router to convert the physical output to OCT*

This appears to be the most pragmatic approach, although it too is not easy. For every commercial router that is to be integrated into SIERA, a special tool needs to be written to convert its output into an OCT physical view. The task is complicated by the fact that the database policies and file formats at the physical level are poorly documented, and often not disclosed publicly. Therefore, substantial "reverse engineering" may be required to write such a translation tool. This is the approach used to convert the output of the router from Racal-Redac into OCT.

3.4 Module Generation from Behavioral Specifications

Even though the layout-generator tools described in the previous section together with a suitable sub-system module library provide an environment which is superior to those provided by most board design tools, it is possible to raise the level of abstraction higher. Many modules in the top-level architecture are more conveniently described behaviorally and it is often possible to synthesize efficient hardware for them. This is particularly true for modules such as random logic,

memory address decodes, bus-interface logic etc. for which special-purpose synthesis tools can be used to transform the behavioral representation into a netlist of chips at the board level. Several existing as well as new tools are available as module generators in the framework. These tools are typically used as structure_processors which synthesize the structure of a design entity.

The system synthesis process can be viewed as a top-down recursive process. The behavioral specification of the system is transformed during synthesis to a network of entities that are either available as primitives or are specified behaviorally using a different (and usually simpler) abstract model of computation. The synthesis process is then repeated on each of these behaviorally specified entities. This process is recursively repeated until the entire system is specified structurally in terms of known primitive components. Therefore there is a hierarchy of behavioral synthesis tools that are used at different levels of the structural hierarchy of the system. According to this view the tools described in this section are lower-level synthesis tools in the sense that they are used at lower levels of the structural hierarchy of the system. While they cannot synthesize the system hardware as a whole, they can certainly synthesize parts of the system from a suitable behavioral description. In fact the output of some of the tools described below are input for the other tools.

3.4.1 Mapping Random Logic to PLDs and FPGAs

Many modules at the board level are often best implemented on field programmable devices such as PALs, PLDs and FPGAs (for example the ones from Actel and Xilinx). Modules such as address decoders, bus control logic, glue logic, interface logic etc. fall in this category. Implementing such modules using SSI parts is too inefficient in board area while implementing them as ASICs is usually not cost effective. Field programmable devices provide a nice compromise and the ability to map a behavioral level description to a net-list of these devices is very desirable.

A set of tools called *PLDS* [Yu91] is integrated in this module generation framework and provides

the ability to map a block described in a mixed behavioral and high-level structural fashion to a net-list of homogeneous field-programmable devices together with suitable information for programming each of those devices. The mixed behavioral and high-level structural description is basically a hierarchical parameterized net-list (in SDL) where the leaf nodes come either from a macro library or are specified using the language BDS for describing combinational logic. The macro library contains a variety of abstract and often parameterized macro blocks for which the structural decomposition into the primitive building blocks is known. This description can be mapped to a variety of programmable devices which roughly fall into two classes. The first class consists of devices like PALs (e.g. PAL16L8) and PLDs (e.g. Altera EP610) which are small devices with a few large basic blocks. The second class of devices are FPGA devices (Field Programmable Gate Arrays) like Xilinx and Actel FPGAs which consist of a large number of simple basic blocks. This distinction is made because different strategies for synthesis and partitioning are needed for these two class of devices. However a single set of tools accessed as structure-processors and layout-generators provide a common interface to these variety of devices.

Mapping to FPGA Devices [Yu91]

Currently two types of FPGA devices are supported - Xilinx and Actel. As mentioned earlier these FPGAs are characterized by a large number of simple homogeneous basic blocks. Therefore logic synthesis techniques based on technology mapping to a library as well as newly proposed special-purpose synthesis techniques can be used to map the combinational logic described behaviorally in the BDS language to a net-list of these basic blocks. This is done using the built-in utilities in the misII logic synthesis system. This is done for all the modules in the initial description that were specified behaviorally. The synthesized net-lists of basic blocks are then merged with the structurally specified part of the initial description. The resulting hierarchical net-list is then flattened and partitioned to a network of identical FPGA devices. The output is a board-level net-list of the FPGA devices together with specifications for each of the individual FPGAs in an appropriate format (for example, Xilinx's XNF format and Actel's ADF format) for use with the

low-level tools provided by the FPGA manufacturer. Figure 3-2 shows this process as a block diagram.

Some of the FPGA devices also provide a small number of complex or special purpose basic blocks - such as the decoder blocks on Xilinx's XC4000 series FPGAs. The mixed behavioral and structural description allows these special blocks to be used structurally through the macro block library.

Mapping to PALs and PLDs [Yu91][Stone91]

These devices are characterized by a small number of large basic blocks. In FPGAs the basic blocks typically have 4-6 inputs and they are able to implement a large fraction of functions possible from these inputs. In contrast the basic blocks in PALs and PLDs are like PLAs with a large number of inputs (20-30) and a relatively small number of minterms (<10). As a result library based technology mapping techniques available in logic-synthesis systems such as misII do not suffice for implementing the logic as a multi-level network of such large basic blocks. The reason for this is that the large number of inputs results in an exponential explosion in the size of the library. Such devices are therefore handled by a two step partitioning process. In the first step the boolean description is partitioned into a network of these large basic blocks. This is done via special procedures added to misII. This can be viewed as a special-purpose technology mapper. Next this network of large basic blocks is partitioned into a board-level network of chips, just like in the case of FPGAs. Figure 3-3 depicts this mapping process. Together with a suitable macro library for using special resources such as registers that are provided in these PALs and PLDs, this approach works extremely well for the most common uses of these devices where the depth of logic is not very high.

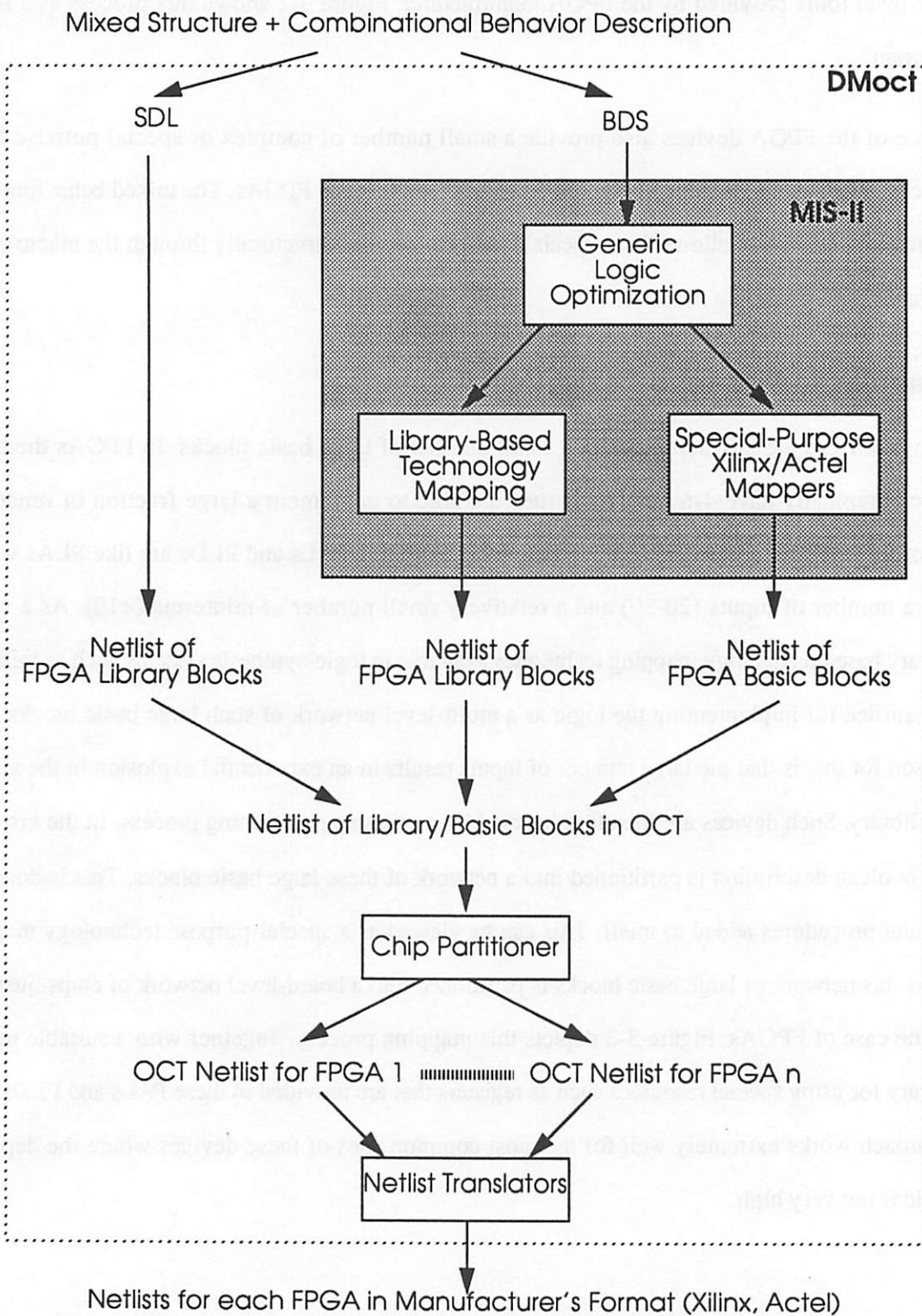


Figure 3-2 : Board-Level Module Generation Using FPGAs

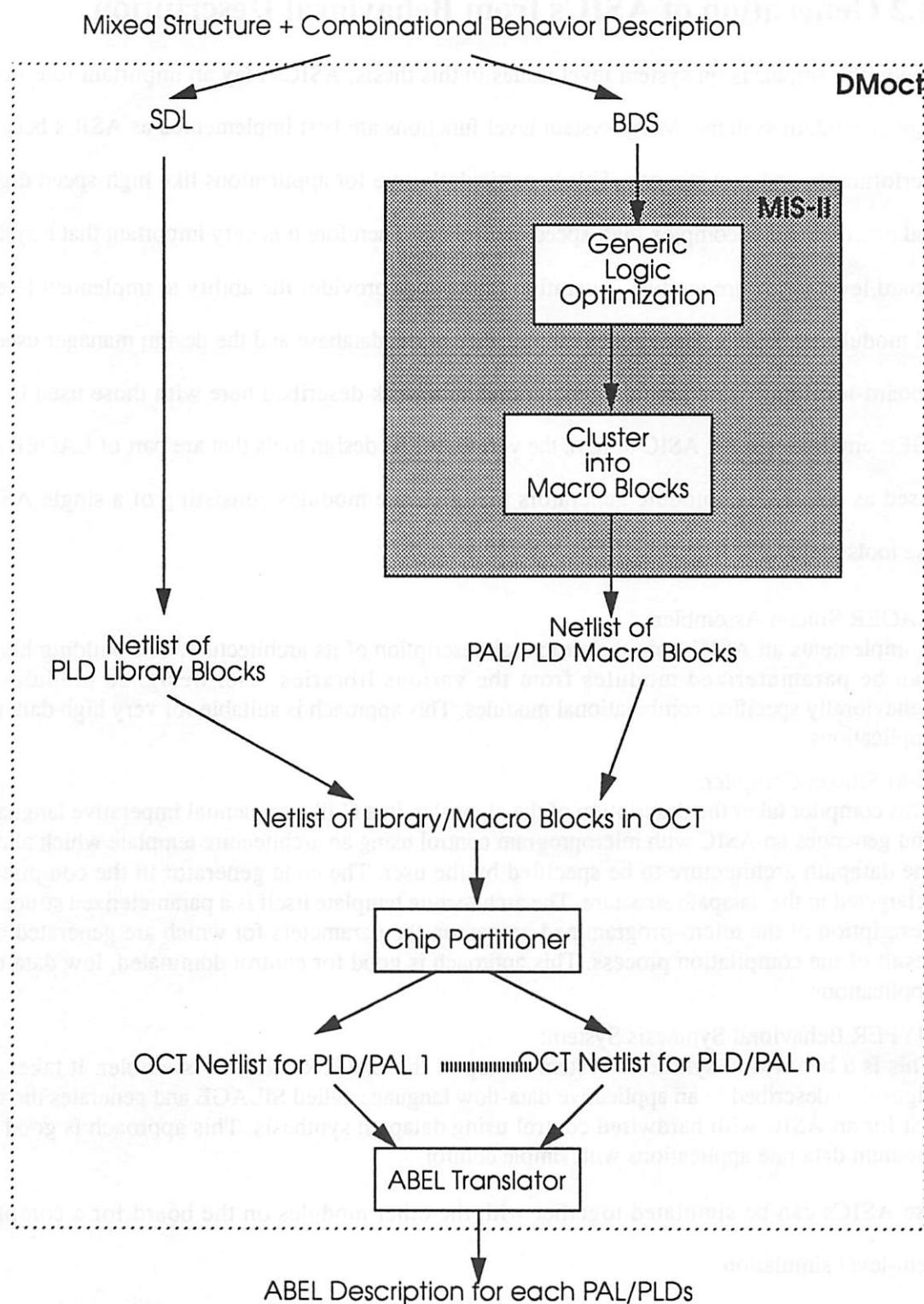


Figure 3-3 : Board-Level Module Generation Using PALs and PLDs

3.4.2 Generation of ASICs from Behavioral Description

Despite the emphasis on system level issues in this thesis, ASICs play an important role in the design of modern systems. Many system level functions are best implemented as ASICs because of performance and cost reasons. This is particularly true for applications like high-speed digital signal processing and complex high-speed controllers. Therefore it is very important that a system or board level hardware module generation framework provides the ability to implement board-level modules as ASICs. Due to the commonality of the database and the design manager used in the board-level hardware module generation framework described here with those used in the LAGER environment for ASIC design, the various ASIC design tools that are part of LAGER can be used as board-level module generators that generate modules consisting of a single ASIC. These tools include:

a. LAGER Silicon Assembler:

It implements an ASIC from the structural description of its architecture. The building blocks can be parameterized modules from the various libraries, user designed modules or behaviorally specified combinational modules. This approach is suitable for very high data rate applications.

b. C-to-Silicon Compiler:

This compiler takes the description of the algorithm in a C-like sequential imperative language, and generates an ASIC with microprogram control using an architecture template which allows the datapath architecture to be specified by the user. The code generator in the compiler is retargeted to the datapath structure. The architecture template itself is a parameterized structural description of the micro-programmed processor, the parameters for which are generated as a result of the compilation process. This approach is good for control dominated, low data rate applications.

c. HYPER Behavioral Synthesis System:

This is a behavioral synthesis system on top of the LAGER Silicon Assembler. It takes the algorithm described in an applicative data-flow language called SILAGE and generates the netlist for an ASIC with hardwired control using datapath synthesis. This approach is good for medium data rate applications with simple control.

These ASICs can be simulated together with the other modules on the board for a complete system-level simulation.

3.4.3 Synthesis of Interface Logic

A very important class of board-level hardware module is that of *Interconnect Modules* as distinguished from the *Data Processing Modules* on which most of this chapter is focussed. These interconnect modules are the glue logic that physically link the data processing modules while meeting I/O protocol and timing constraints. Due to the wide variety of I/O protocols encountered at the system level, a library of interconnect modules is of limited scope. The ability to synthesize these interconnect modules from a high-level description is the key to having a library of reusable sub-system level data processing modules. It eliminates the need to design multiple versions of these sub-systems that have the same computational ability but satisfy different I/O protocol constraints. For example, instead of having several versions of a processor module - say one with a VME interface, another with a Multibus interface, and yet another with a SBUS interface - one can have only one processor module in the library and synthesize the appropriate bus-interface when needed. This ability to synthesize interconnect modules is particularly desirable at the system level because off-shelf components play a very important role and unfortunately there is little control over their I/O interface protocols.

The module generation environment described here uses the ALOHA interconnect module synthesis system that is being developed by Jane Sun at Berkeley. [Sun92a] gives a detailed description of the ALOHA synthesis system, a pictorial overview of which is presented in Figure 3-4. The interconnect module is specified at a high-level in a language called *IDL* (Interface Description Language) in which only the data flow in the interconnect module is expressed together with the appropriate protocol names. The IDL description essentially specifies the *temporal* and *spatial* mapping of the source data streams to the destination data streams in a protocol and technology independent manner. The details about the I/O protocols of the modules being interconnected are described separately and stored in the module library. An example of this is given in section 3.5.3. The event graphs (also called signal transition graphs or STGs) corresponding to the I/O protocols for the various modules are merged according to the data-flow

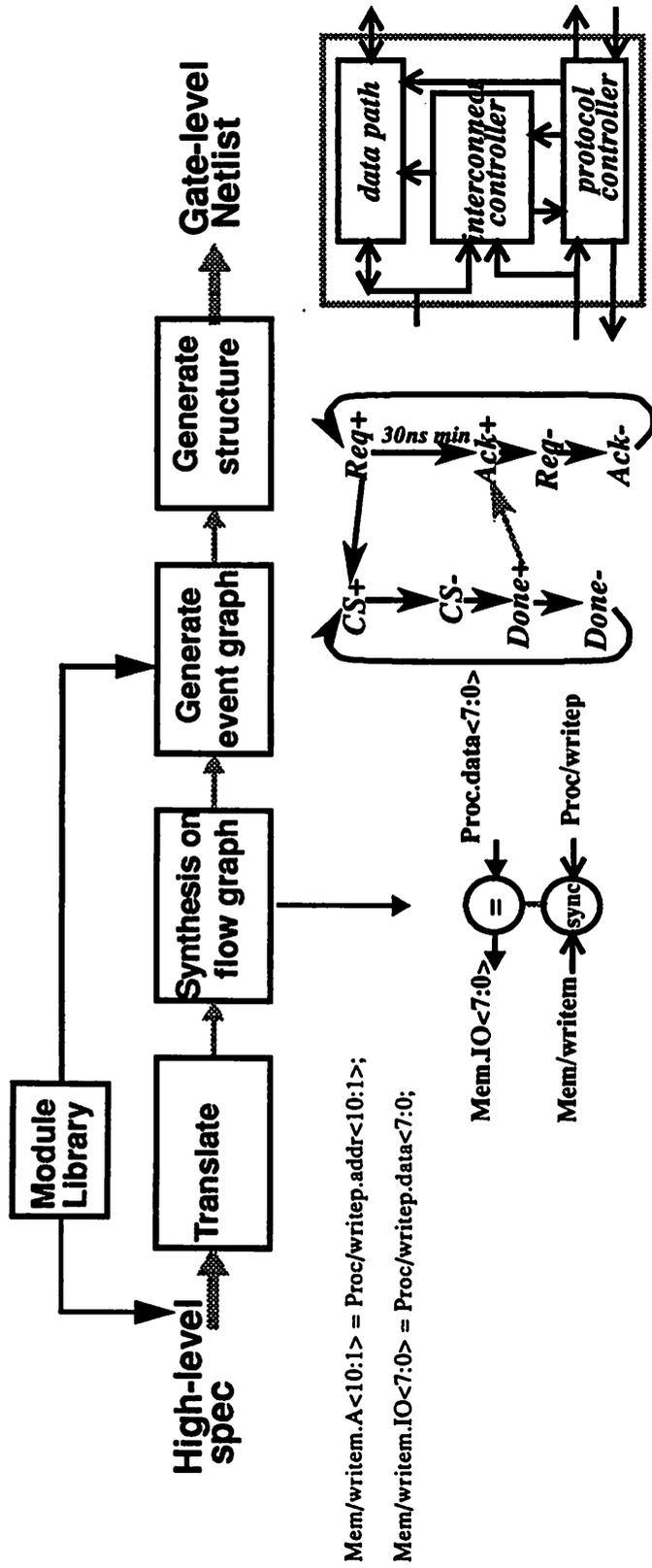


Figure 3-4 : Interconnect Module Generation Using ALOHA [Sun92a]

described in the IDL description to produce a STG for the entire interconnect module. Then lower-level foreign tools, such as *Async* [Jones91] for synthesizing asynchronous finite-state machines (FSMs) from the STG description, are used by ALOHA to produce a gate level net-list together with timing constraints. The gate level net-list can then be implemented as an ASIC or as FPGAs or PLDs using the tools described earlier.

3.5 Libraries

A key aspect of the hardware module generation strategy is the library of reusable components. The purpose is to encourage reuse of design effort, and modularity. The library members can use the same layout generators and synthesis tools as described in the previous subsection. However, in order to make them reusable, the modules are usually parameterized and can be customized with suitable values of the parameters.

There are three distinct types of libraries that are needed corresponding to three different types of reusable elements. The following subsections describe the three libraries.

3.5.1 Package Library

The package library contains the physical information about the various board level packages. There is no analogue to the package library at the chip level. The reason is that at the chip level every primitive component, for example a leafcell such as a NAND gate, has a distinct physical layout. On the other hand, at the board level many different primitive components may be housed in packages with the same physical characteristics. From a data management perspective it is better to keep such shared information centralized in a single copy.

The library includes a variety of packages including DIP and SIP packages, PGA packages, packages for discrete parts, surface-mount packages such as SOIC, PLCC etc. Each package is specified by a physical view in the OCT database. The physical view contains all the geometry

corresponding to the package in the various layers of the board. In addition, a special placement layer is used to specify a placement boundary. Some of the physical geometry is attached to a formal terminal that indicate the pins, using which the package connects to the signal traces on the board.

The library also has information about translating the package name and pin number information to that of commercial package databases - this is used by tools like *oct2rinf* for generating net-lists for foreign routers.

The policy for this library is still in its infancy and a more elaborate policy is planned to incorporate additional physical information such as 3D information about the package height, capacitances and inductances on the pin leads, etc.

3.5.2 Primitive Component Library

The primitive library is akin to a leafcell library at the chip level. It contains information about board level primitive components - chips, connectors, discrete components etc. The distinguishing characteristic of these primitive components is that a package name must be specified for each of them in order for the tools to access the physical information about the component.

Table 3-2 shows a partial listing of contents of the primitive component library. Each primitive component in the library requires a SDL file written according to a specific policy, and additional optional files for simulation model, I/O protocol and documentation.

Figure 3-5 shows the SDL file for a TTL xx00 part. The file gives a black-box description of a generic 7400 series part. Since the part is available as a primitive, there are no subcells instantiated in the SDL file. The file lists all the formal terminals of the device. The attributes **TERMTYPE** and **DIRECTION** are defined for all formal terminals to facilitate checking the final design for electrical problems such as open inputs and shorted outputs. The attribute **PACKAGECLASS** with

```

(parent-cell xx00 (PACKAGECLASS PCB) (CELLCLASS LEAF))

; sel_pkg is a lisp function that returns its argument
; which is in the same position as the position of the
; PACKAGETYPE in the list of packages PACKAGELIST.
; This is used to select a value appropriate for the
; package being used.

(parameters
  ; pre-defined local variable - list of allowed packages
  (PACKAGELIST `("DIP" "SOIC" "PLCC") (local))

  ; user parameter - the package type
  (PACKAGETYPE "SOIC"
    (assert (memql PACKAGETYPE PACKAGELIST)))

  ; other parameters - usually not altered by the user
  (PARTNAME "xx00")
  (PARTTYPE "DIGITAL")
  (PACKAGENAME (sel_pkg "DIP14" "SOIC14" "PLCC20")))
)

(layout-generator psg)

; PINNUMBER, TERMTYPE, and DIRECTION are attributes
; required to be attached to each formal terminal

(term A1 (PINNUMBER (sel_pkg 1 1 2))
  (TERMTYPE SIGNAL) (DIRECTION INPUT))
(term A2 (PINNUMBER (sel_pkg 4 4 6))
  (TERMTYPE SIGNAL) (DIRECTION INPUT))
(term A3 (PINNUMBER (sel_pkg 9 9 13))
  (TERMTYPE SIGNAL) (DIRECTION INPUT))
(term A4 (PINNUMBER (sel_pkg 12 12 18))
  (TERMTYPE SIGNAL) (DIRECTION INPUT))
(term B1 (PINNUMBER (sel_pkg 2 2 3))
  (TERMTYPE SIGNAL) (DIRECTION INPUT))
(term B2 (PINNUMBER (sel_pkg 5 5 8))
  (TERMTYPE SIGNAL) (DIRECTION INPUT))
(term B3 (PINNUMBER (sel_pkg 10 10 14))
  (TERMTYPE SIGNAL) (DIRECTION INPUT))
(term B4 (PINNUMBER (sel_pkg 13 13 19))
  (TERMTYPE SIGNAL) (DIRECTION INPUT))
(term Y1 (PINNUMBER (sel_pkg 3 3 4))
  (TERMTYPE SIGNAL) (DIRECTION OUTPUT))
(term Y2 (PINNUMBER (sel_pkg 6 6 9))
  (TERMTYPE SIGNAL) (DIRECTION OUTPUT))
(term Y3 (PINNUMBER (sel_pkg 8 8 12))
  (TERMTYPE SIGNAL) (DIRECTION OUTPUT))
(term Y4 (PINNUMBER (sel_pkg 11 11 16))
  (TERMTYPE SIGNAL) (DIRECTION OUTPUT))
(term VCC (PINNUMBER (sel_pkg 14 14 20))
  (TERMTYPE SUPPLY))
(term GND (PINNUMBER (sel_pkg 7 7 10))
  (TERMTYPE GROUND))

(end-sdl)

```

Figure 3-5 : Sample SDL File for a xx00 (7400) Chip

PART CATEGORY	PART NAMES
SSI Logic	TTL parts such as xx00, xx04, xx240, xx241, xx646 etc.
Memory	CYM1621, CYM1641, CYM1831, CYM1841, IDT7025 etc.
Processors	DSP32C, TMS320C30, MC96002
Programmable Logic	PALs (16L8, 20L8 etc.), PLDs (610, 910, 1810 etc.)
FPGA	Actel (ACT1010, ACT1020), Xilinx (XC3090, XC4005)
I/O Devices	VME interface (VME2000, VME3000, VME1220A, VME1220B), UARTS (SCN2681, Am7968, Am7969)
Miscellaneous	TTL Oscillator, ECL-TTL converter MC10H350, clock divider MC74HC4040, opto-isolator HCPL2631, RS232 driver (MAX233) etc.
Analog	Discrete parts (capacitors, resistors, inductors, pots, leds), Opamps (TL082, TL084, AD711), A/D (AD7870), D/A (AD558, DAC811), Optical driver and receiver (Am79h1000T, Am79h1000R), Transistor arrays (MC1411, MC1413), Delay Lines, LM555, voltage regulators (LM79xx, LM78xx) etc.
Packaging	Variety of headers, connectors, switches, terminal blocks etc.

Table 3-2 : Partial Listing of the Board Level Primitive Component Library

value PCB for xx00 is used to indicate that it is packaged so as to be contained by a printed-circuit board. The cell is parameterized so as to let the user specify the package name. The parameter `PACKAGENAME` is used by tools as a key to access the package database. The parameters `PARTNAME` and `PARTTYPE` are used for annotation and for generating a part list. The other two parameters `PACKAGELIST` and `PACKAGETYPE` together with the lisp function `sel_pkg` facilitate handling of multiple packages. The function `sel_pkg` selects its *i*-th argument if `PACKAGETYPE` is the *i*-th member of `PACKAGELIST`. This file also shows some other features of SDL. Parameters such as `PACKAGELIST` with the attribute *local* are not stored in the database after evaluation and are hence not visible to other tools - they are like temporary or local variables. Also the parameter `PACKAGETYPE` has an attribute named *assert* whose value is a string representing a lisp s-expression which on evaluation must return a non-nil value. This provides a

mechanism for ensuring that the parameter values is an acceptable one. In the example this mechanism is used to ensure that the value of `PACKAGETYPE` is one of the packages listed in `PACKAGELIST` - the lisp function `memql` returns true only if `PACKAGETYPE` is contained in the list `PACKAGELIST`. Finally the attribute `CELLCLASS` for the cell `x00` is used to indicate the type of cell. The value `LEAF` indicates that it is a primitive cell with no substructure. It further indicates that the black-box behavior of the cell is independent of the parameters. This information is used to optimize the database storage by sharing the same *structure_instance* and *physical* view between different instances of `xx00`.

3.5.3 Subsystem Module Library

This library is analogous to a macrocell library at the chip level. At the chip level such a library may contain adders, multipliers, RAMs, FSMs etc. At the board level an adder or a multiplier is functionally at too low a level - in fact a single chip is usually far more complex. Complete sub-systems, such as embedded programmable computers, I/O interfaces, signal processing sub-systems etc., are more appropriate modules for representing hardware architecture of boards.

An extensive library of such reusable sub-system level modules has been created using the tools described earlier in this chapter. Table 3-3 is a partial list of the sub-system modules available in the library. Many modules in the library are parameterized so that they really represent a class of sub-systems from which a particular module that is tailored to the needs of the application can be instantiated. As shown in the table, the library has a variety of memory sub-systems, complete embedded computer modules based around different processor chips, bus-interface modules, data acquisition modules etc. The library has been developed over the course of several board designs and continues to grow as more systems in diverse application areas are being designed. For example, some of the modules contributed by other on-going design projects include SBUS interface module, video frame-buffer module, JTAG test controller module etc. This library of sub-systems is a major contributor to the goal of reusability.

	MODULE NAME	DESCRIPTION
MEMORY	sram64Kx32	64Kx32 static ram with multiple area/speed/cost/loading choices
	sram256Kx32	256Kx32 static ram with multiple area/speed/cost/loading choices
	dpram8Kx32	8Kx32 true dual-ported static ram with h/w semaphores and mail-box interrupts
COMPLETE PROCESSOR MODULES	procC30	an extensively parameterized TMS320C30 based processor module with multiple types of SRAM local memory banks, optional host interface based on multiple 32 Kbyte dual-port ram banks, and I/O interface for attaching a parameterized number of memory mapped I/O slave devices. The memory and I/O address map and the memory sizes are also configurable.
	procsimple32C	a simple DSP32C based processor module with one bank of 256 Kbyte 0-wait state local memory, optional 32 Kbyte dual-port RAM based host interface, bus-control/glue logic and I/O interface for attaching a parameterized number of memory mapped I/O slave devices
	procsimple96K	a simple MC96002 based processor module with one bank of 1 Mbyte 0-wait state local memory, host interface and bus-control/glue logic
I/O MODULES	outport8	digital output port <= 8 bits with load, and, or and xor bit-mask operations
	a2d	analog to digital converter for <= 12 bits, <= 100 KHz
	d2a	digital to analog converter for <= 12 bits, <= 100 KHz
	rs232dual	1/2 channel RS232/432 serial I/O module
	opticalT/opticalR	Transmitter and receiver modules for 8/9/10 bit parallel I/O over a serial fiber-optic link up to 125 Mbits/sec (10 Mword/sec) with serial-parallel conversion and error-detection.
MISC.	vmeslave	module to interface an arbitrary number of memory-mapped slave and interrupter devices to VME bus
	brdreset	generation of multiple power-up/switch/host controlled resets

Table 3-3 : Partial Listing of the Board Level Sub-system Module Library

These sub-system modules are usually composed of other modules and/or primitive components. As such a sub-system module is usually specified by a hierarchy of SDL files describing the structural interface between its building blocks which may be available as an off-shelf primitive component or may be behaviorally specified using one of the tools described in section 3.4. The SDL files also contain floorplanning information using one of the strategies listed in section 3.3.2. In addition optional files for the simulation model, I/O protocol and documentation are also present in the library for each sub-system module. The I/O protocol information is particularly important for many sub-systems. It describes the timing diagrams relating the events on the various formal terminals of the sub-system when it interacts with other sub-systems. Several different protocols corresponding to different types of transactions may be defined for a given sub-system. The timing diagram is represented by an *event graph* that describes the signalling protocol in terms of events or transitions on the I/O signals. The causality relationships and timing constraints between these events are represented by directed edges in the event graph with the events themselves being represented by the nodes. The event graph is represented in the library by a text file in *afl* format as described in [Sun92b].

Example 1: A TMS320C30 Based Processor Module

As an example the TMS320C30 based uniprocessor module *procC30* provides a complete microcomputer based around a powerful 32-bit digital signal processor. It can be used wherever a powerful, software programmable, embedded computation core is required on the board. The configuration of the module in term of its memory organization, I/O devices etc. is specified through parameters. All the blocks for making a self-sufficient computer, such as the memory, address decoder, clock generator, wait-state generator, host interface, I/O device controller etc., are included. Some of these building blocks make use of field-programmable devices such as PALs and PLDs. The implementation information for these devices is also generated as part of the module generation process. The address and data buffers are also tailored according to the amount

of the memory and I/O devices. The module is also placed giving a fairly good first-cut placement. Figure 3-6 shows an example instance of this processor module.

Figure 3-7 shows the basic architecture of the processor module. It has the following major components:

- a. TMS320C30 processor chip
- b. clock generator (TTL oscillator)
- c. parameterized wait-state generator
- d. parameterized address decoder
- e. data and address bus buffers
- f. OEN generator
- g. dual-port memory bank for host communication
- h. three different kinds of SRAM banks with different speed/area/cost trade-offs

The TMS320C30 processor has two busses - a main memory bus, and an extension bus. In the design the extension bus is left untouched, and brought out for use outside the module. All the memory bank inside the module reside on the main memory bus. In addition, the main memory bus is also brought out of the module so that other devices can be attached to it. To simplify this task, the processor module can also generate interface signals needed for attaching external slave devices to the main memory bus. This is done using the parameterized wait-state generator and parameterized address decoder modules that are already being used for the internal memory banks.

The design is optimized for speed and ease of software. For example, the wait-state generation is handled completely in hardware and can support devices ranging from 0 wait-states to 2 wait-states. This makes software manipulation of wait-states unnecessary thus enormously simplifying the code.

Following are the parameters that can be specified for the module:

- a. NDPRAM8C = number of 8Kx32 dual-port banks in the host interface.
It should be ≥ 0 (default=2). If NDPRAM8C == 0 then no host interface is present.
-

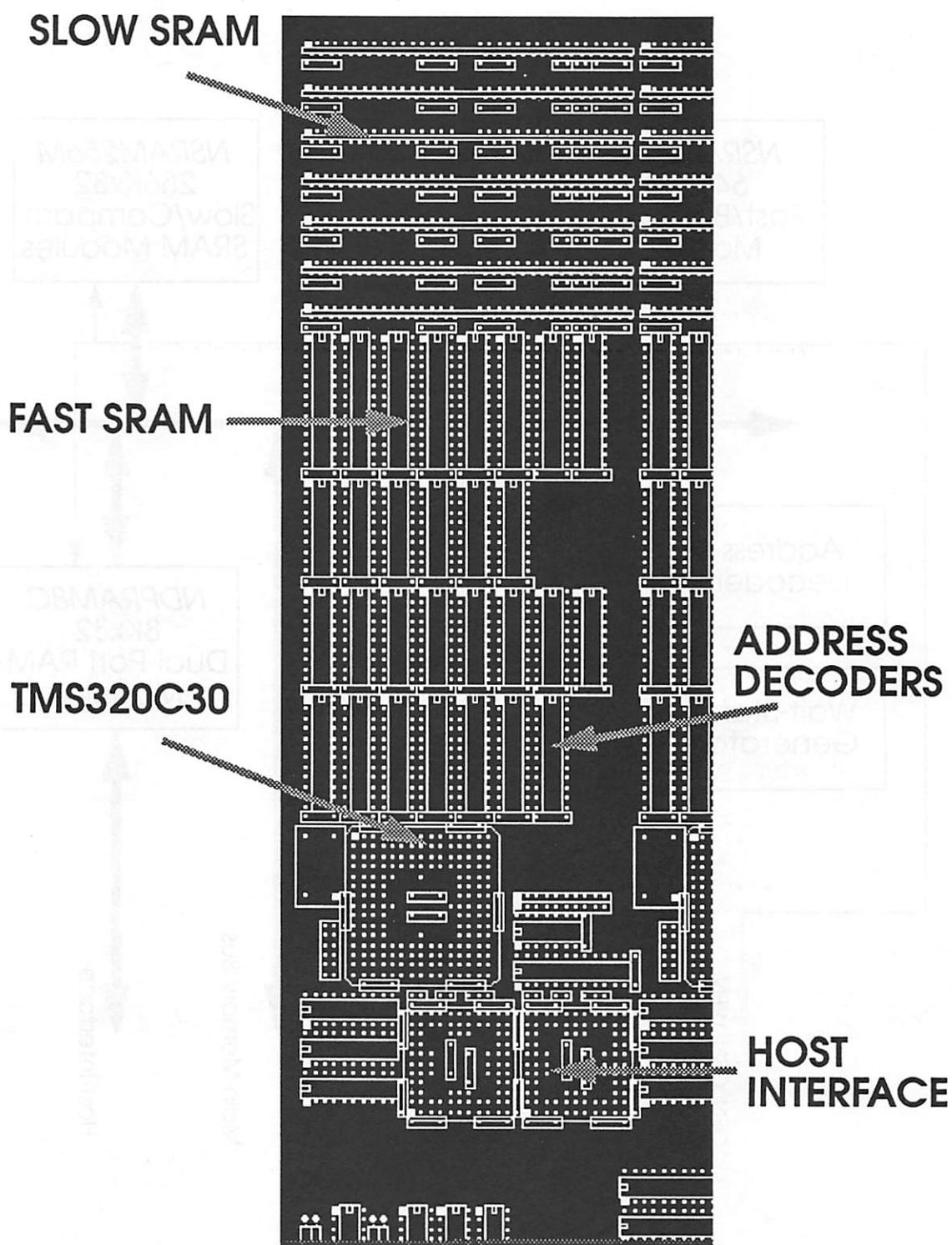


Figure 3-6 : Layout of an Instance of *procC30*, a TMS320C30 based processor module

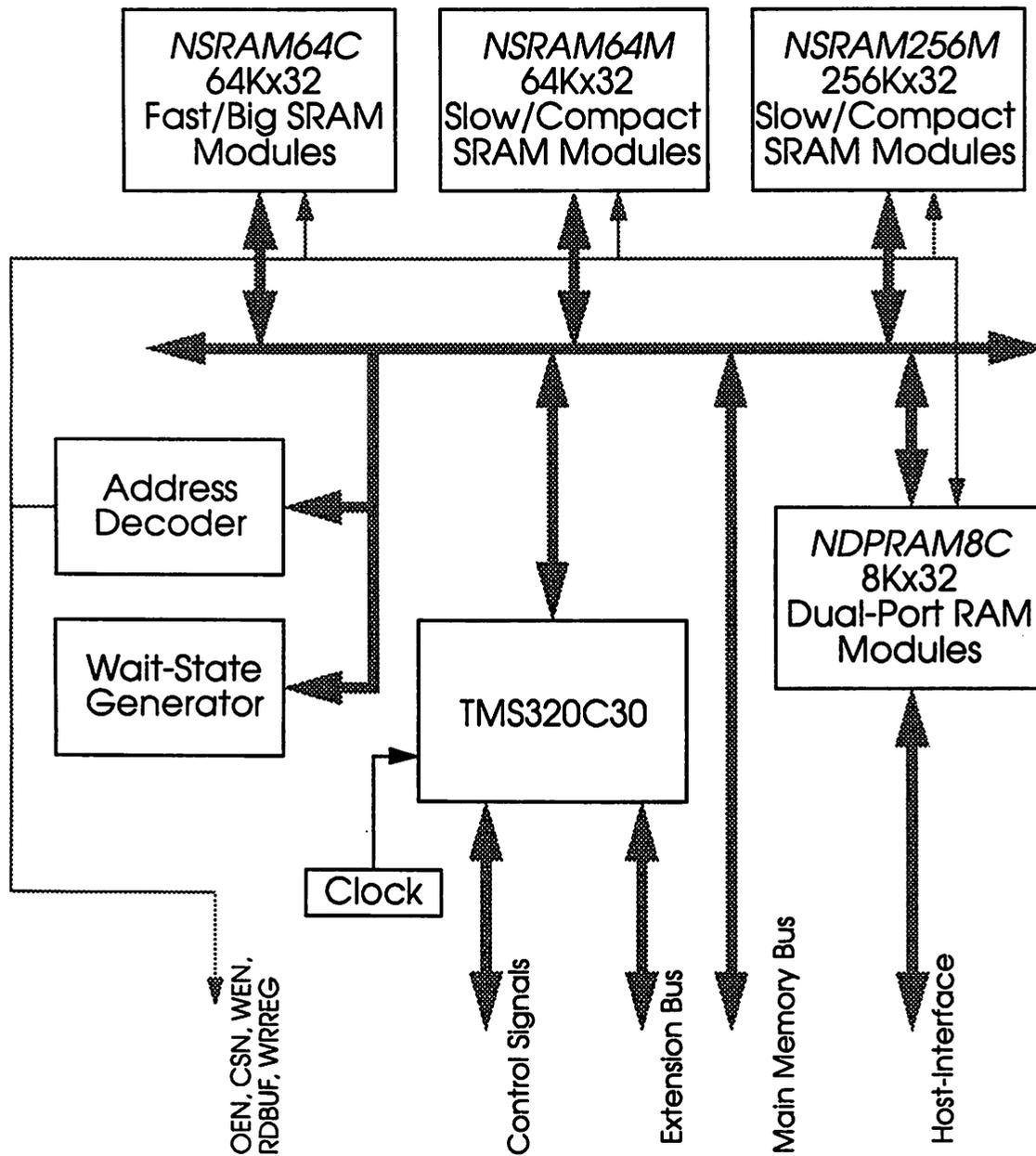


Figure 3-7 : Architecture of the TMS320C30 based *procC30* Processor Module

-
- b. NSRAM64C = number of fast but big 64Kx32 SRAM modules composed of 64Kx4 chips. It should be ≥ 0 (default=1).
 - c. NSRAM64M = number of slow but compact 64Kx32 SRAM modules composed of 64Kx32 MCMs. It should be ≥ 0 (default=0).
 - d. NSRAM256M = number of slow but compact 256Kx32 SRAM modules composed of 256Kx32 MCMs. It should be ≥ 0 (default=3).
 - e. SRAM_BASE_ADDR = base of address of the internal SRAM modules. It should be a multiple of 0x040000 (default=0x040000). The modules are mapped in the following order, starting from this address: NSRAM256M 256Kx32 SRAM modules, NSRAM64M 64Kx32 modules, and NSRAM64C 64Kx32 modules.
 - f. NEEDEMU $\in \{0, 1\}$. If NEEDEMU==1 then a special interface to TMS320C30 in-circuit emulator is also included. Its default value is 1.
 - g. PROCNAME is a string giving the name of the processor. Its default value is "TMS".
 - h. FLAGTHIN $\in \{t, nil\}$. If FLAGTHIN==t then the aspect ration of the placed module is made tall and thin. Its default value is t.
 - i. NEXTRDBUF is the number of external RDBUF signals needed for attaching slave devices to the main memory bus. A RDBUF signal is useful for attaching tri-state buffer chips as input ports. It should be ≥ 0 and has a default value of 0.
 - j. NEXTWRREG is the number of external WRREG signals needed for attaching slave devices to the main memory bus. A WRREG signal is useful for attaching positive edge-triggered flip-flops or negative level-sensitive transparent latches as output ports. It should be ≥ 0 and has a default value of 0.
 - k. NEXTOEN is the number of external OEN signals needed for attaching external slave devices to the main memory bus. An OEN signal is useful for active low output enables of SRAM-like devices. It should be ≥ 0 and has a default value of 0.
 - l. NEXTWEN is the number of external WEN signals needed for attaching external slave devices to the main memory bus. A WEN signal is useful for active low write enables of SRAM-like devices. It should be ≥ 0 and has a default value of 0.
 - m. NEXTCSN is the number of external CSN signals needed for attaching external slave devices to the main memory bus. A CSN signal is useful for active low chip-selects. It should be ≥ 0 and has a default value of 0.
 - n. SEL_EXT_OWS, SEL_EXT_1WS, SEL_EXT_2WS are strings indicating the wait-state map for memory locations on the main bus that are used by the external slave devices. These strings are in the form of boolean (0,1) valued functions of the variables *Address*, and *PAGE4KW*. These variables are identical and correspond to the bits A12..A23 of the memory address. The boolean functions are in ABEL syntax. Every external device on the main memory bus is required to either generate a handshake signal which results in the TMS_RDY_L input to be pulled low, or should result in one, and only one, of the above three functions evaluating to 1 when the device is being addressed.
-

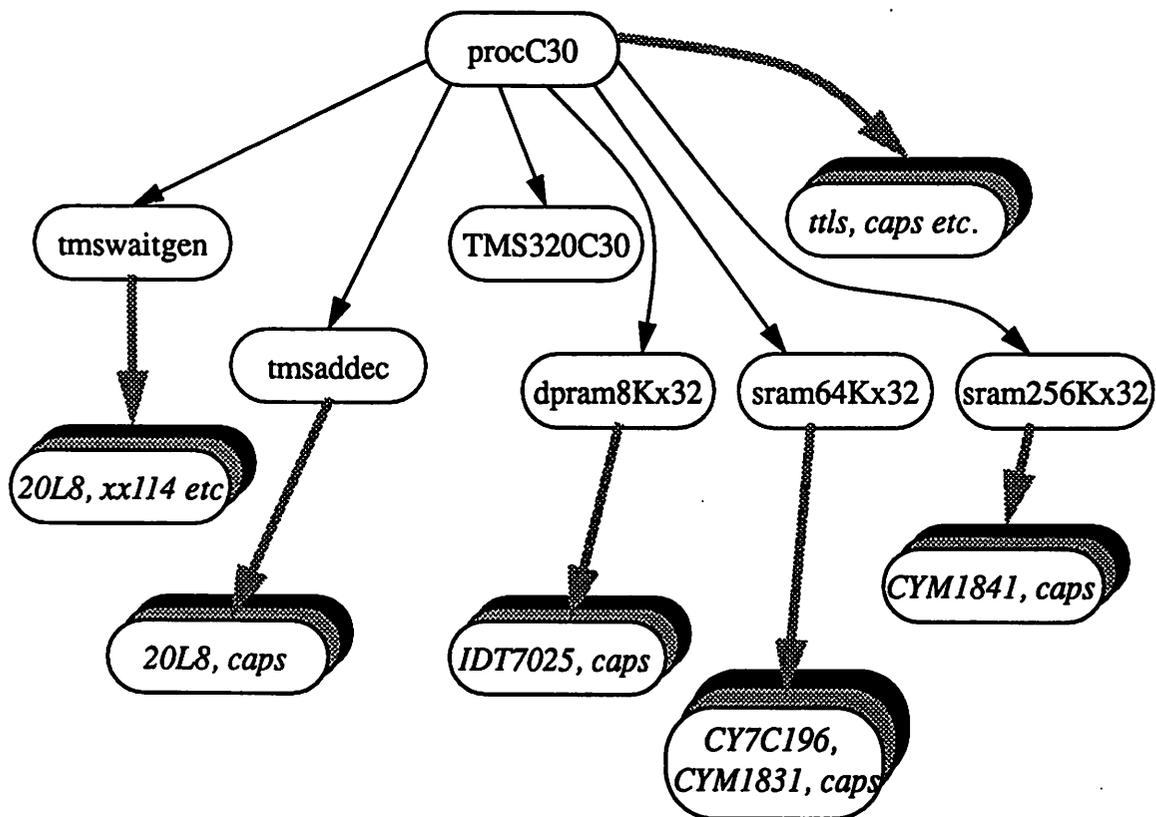


Figure 3-8 : Hierarchy of SDL files for the *procC30* Module

- o. EXTCSN_COND is a list of NEXTCSN elements where each element is a string corresponding to a boolean valued function of *Address* and *PAGE4KW* that evaluate to 1 when the corresponding CSN signal needs to be asserted.
- p. EXTRDBUF_COND and EXTWRREG_COND are similar to EXTCSN_COND except that they indicate the address selection conditions associated with the RDBUF and WRREG signals.

The SDL file for the module is far too complex (> 900 lines of SDL code) to present here.

Therefore only the hierarchy of SDL files used in designing the module is shown in Figure 3-8.

Appendix C contains information on accessing the on-line copies of the SDL files for this module as well as for the other modules in the library. Several different instances of this module have been fabricated and tested. The module operates flawlessly at the maximum speed of 33MHz.

Example 2: An Asynchronous Bit-Parallel Fiber Optic Receiver Module

This example is a complete bit-parallel optical receiver section. Together with its dual *opticalT* it provides the ability to send a multi-bit word at one end of an optical communication link and receive it at the other end. Data rates up to 125 Mbits/sec can be achieved together with a bit-error detection capability. The module is based around commercial parts. The two main parts are *Am79h1000R* and *Am7969*. The first part is a optical data link receiver which converts the optical signal received through an optical fiber with a ST connector to an electrical signal. The optical signal is assumed to be 0-1 intensity modulated and the output is a bit-stream on a pseudo-ECL differential signal pair. The second part, popularly known as the TAXI receiver, is a high-speed serial-to-parallel receiver. It asynchronously decodes the serial bit-stream that has been encoded by its sister chip *Am7968* TAXI transmitter that is used in the *opticalT* module. The remaining parts in this module are there for noise considerations. Since the optical data link receiver deals with very small currents (nano amperes) to distinguish between 0 and 1 bits, noise is a major consideration. The module uses ferrite bead filters at strategic locations and in addition decouples the optical data link receiver from the rest of the system by optionally using a. c. coupling which is selected through a parameter. Encapsulating these details about design for noise into a robust module makes a high-speed point-to-point optical communication link sub-system easily accessible to a system designer. The robot peripheral board example described later in this chapter uses such a communication link to communicate with a remote robot controller using a custom packet protocol.

Figure 3-9 shows the black-box picture of this module while the SDL file describing the module in a purely structural fashion is shown following this paragraph. All the subcells in the SDL file are primitive components except for the *linecoupler* subcell which is a separate module that provides a.c. or d.c. coupling.

```
; File: opticalR.sdl  
;  
; A complete bit-parallel optical receiver section using Am79h1000R
```

```

; optical data link receiver and Am7969 TAXIchip receiver

(parent-cell opticalR)
(parameters
; number of data bits - should be 8 or 9 or 10
; sets TAXI in 8 or 9 or 10 bit mode
(DBITS 10 (assert (memql DBITS `(8 9 10))))
; 1 ==> use ferrite beads
(use_ferrite 1 (assert (memql use_ferrite `(0 1))))
; 0 ==> dc coupling, 1 ==> ac coupling + separate ODL and TAXI planes
(ac_coupling 1 (assert (memql ac_coupling `(0 1))))

;local variables
(CBITS (- 12 DBITS) (local))
(dc_coupling (= ac_coupling 0) (local))
(no_ferrite (= use_ferrite 0) (local))
)

; use pfp for placement
(layout-generator "pfp -B -a")

; declare the subcells and their placement attributes
(subcells
; ODL receiver chip
(Am79h1000R ((inst odlR (ROTATION `270.0) (POSITION `(0.8 2.6)))))
; taxi receiver chip

```

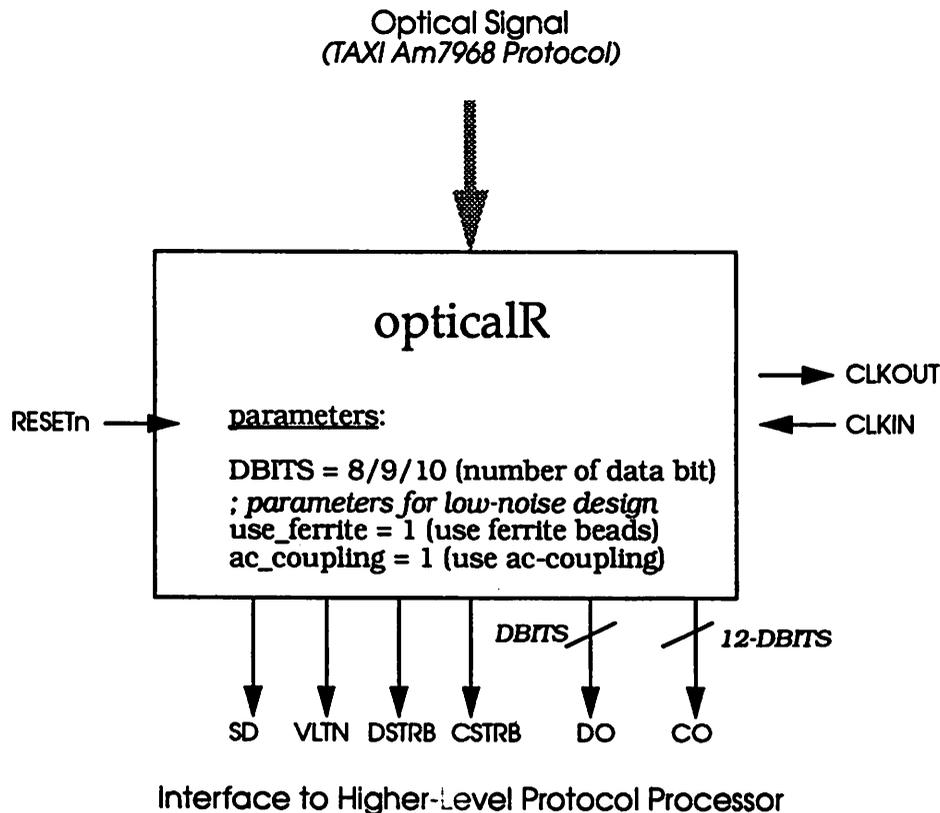


Figure 3-9 : Black-Box Picture of the *opticalR* Module

```
(Am7969 ((inst taxiR (ROTATION '90.0) (POSITION '(1.1 0.0))))
((DBITS DBITS))
; PECL to TTL converter to generate SD (Signal Detect)
(MC10H350 ((inst pecl2ttl (ROTATION '270.0) (POSITION '(0.0 0.8)))))

; 2.7 uH inductors to connect TAXI and ODL planes
; note the use of CONDITIONAL so that the inductors are
; included only if ac_coupling is specified to be 1
(inductor (
(inst I1 (ROTATION '0.0) (POSITION '(0.6 1.4)))
(inst I2 (ROTATION '0.0) (POSITION '(0.5 1.4)))
) ((CONDITIONAL ac_coupling)))

... declaration of other components ...

)

; declare the connectivity in a pin-list fashion

; these are the terminals coming out of this module
(inst parent (
(DSTRB DSTRB)
(CSTRB CSTRB)
(VLTN VLTN)
(DO DO (width DBITS)) ; bus with parameterized width ...
(CO CO (width CBITS))
(CLKIN CLKIN)
(CLKOUT CLKOUT)
(SD SD) ; signal detect
(RESETn RESETn)
(ODLGND ODLGND)
(TAXIGND TAXIGND)
(ODLVCC ODLVCC)
(TAXIVCC TAXIVCC)
))

; these are the subcells
(inst odlR (
(VEE1 ODLGND)
(VEE2 ODLGND)
(VEE3 ODLGND)
(SD+ SDpos)
(SD- SDneg)
(VEE4 ODLGND)
(VEE5 ODLGND)
(VEE6 ODLGND)
(Vcc1 ODLVCC)
(Vcc2 ODLVCC)
(Rx- Rxpos)
(Rx+ Rxneg)
))

(inst taxiR (
(DO DO (width DBITS))
(DSTRB DSTRB)
(CO CO (width CBITS))
(CSTRB CSTRB)
(VLTN VLTN)
(SERIN+ cRxpos)
(SERIN- cRxneg)
(CNB CLKOUT) ; TAXI in local mode
```

```

(IGM IGM) ; it really is unused
(DMS TAXIGND (CONDITIONAL (= DBITS 8))) ; conditional connection
(DMS TAXIVCC (CONDITIONAL (= DBITS 9)))
(DMS DMS_FLOAT (CONDITIONAL (= DBITS 10)))
(CLK CLKOUT)
(X1 CLKIN)
(X2 TAXIGND)
(RESET L RESETn)
(VCC1_TTL TAXIVCC) ; TTL power plane
(VCC2_CML TAXIVCC (CONDITIONAL no_ferrite)) ; logic and analog
(VCC2_CML fTAXIVCC (CONDITIONAL use_ferrite)) ; logic and analog
(GND1_TTL TAXIGND) ; TTL
(GND2_CML TAXIGND) ; logic and analog
))

```

... connections to other subcells ...

(end-sdl)

This module interfaces electrically with the rest of the system through a set of signals over which two types of transactions are defined. The two types of transactions correspond to the arrival of a data word and the arrival of a control word - the communication link implemented by the module has the provision for a 8/9/10-bit data channel and a 4/3/2-bit control sub-channel. The signalling protocols for the two types of transactions are pictorially represented in Figure 3-10 from which it is clear that the transactions are distinguished by the strobe that is asserted by the module - DSTRB or CSTRB. These event graphs are stored in the library in the *afl* text format mentioned earlier. Figure 3-11 shows the *afl* description for the event graph corresponding to the arrival of a data word.

3.6 Simulation and Netlist Checking

Although the process of board hardware generation is quite automated, the integrity of the resulting implementation is never quite guaranteed because of the inevitable presence of “bugs” in the module generators and libraries. Therefore it is desirable that one needs to be able to verify that the resulting implementation has the required functionality, and that it is free of electrical problems.

Two tools are provided in SIERA for this purpose. The first is *SIVcheck* which does a static

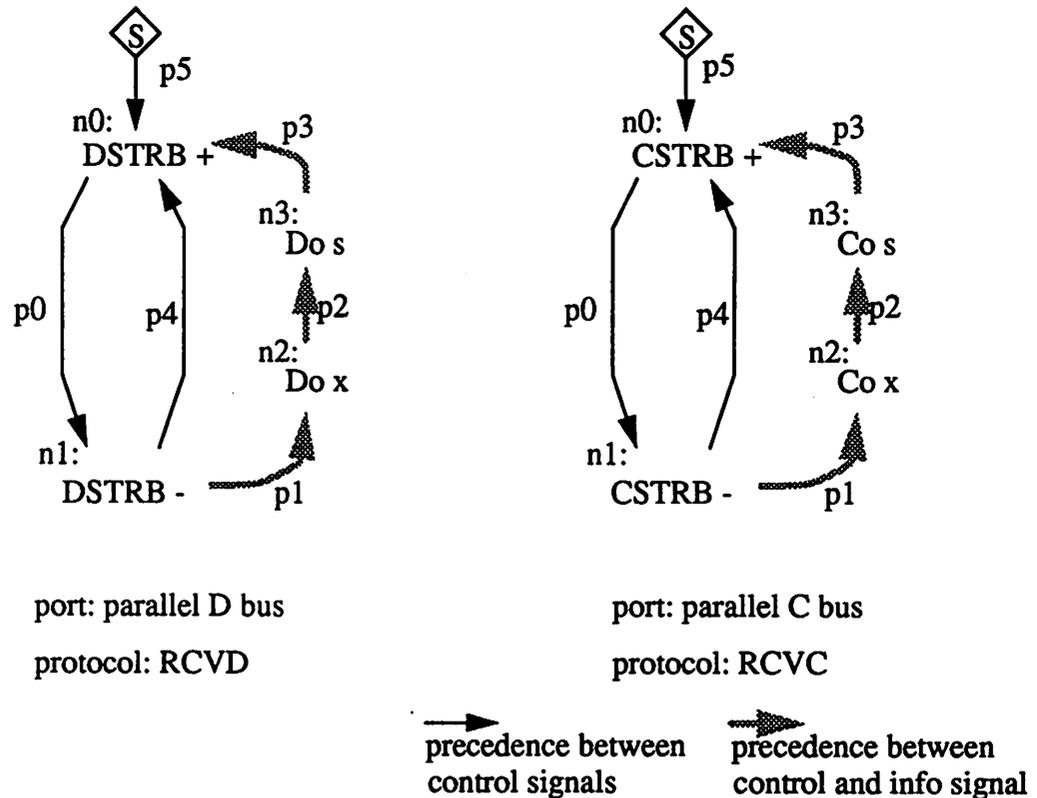


Figure 3-10 : Event Graphs for *opticalR* Sub-system Module

analysis of a flattened *structure_instance* view of the board, and tries to check for electrical correctness. It discovers problems such as multiple outputs driving the same net, or a floating input. It accomplishes this by using the *DIRECTION* and *TERMTYPE* attributes attached to the formal terminals of the individual chips. Experience has shown that such a static analysis of the final netlist catches a vast majority of design problems - and should be done before costlier dynamic techniques such as simulation are attempted.

Static analysis tools such as *SIVcheck* are usually not powerful enough to detect logical and timing problems in the design. In the absence of formal verification techniques, such problems are best detected by doing a bit-level simulation of the entire board. Support for such simulation is provided in *SIERA* using the *THOR* simulator from Stanford [THOR]. *THOR* is an event-driven simulator for modelling digital system at a functional level using bit and bit-vector data types.

```

/* Event graph for
* module : opticalR
* port : parallel D bus
* protocol: receive cycle
*/
(GRAPH
  (NAME rcvD)
  (CLASS MODULE)
  (MODEL ( (model_name masterp) ))
  (ARGUMENTS ( (port taxircvr_dbus) (timeunit ns)
  (parameter CBITS) (parameter DBITS) ))
  (NODELIST
    (NODE
      (NAME n0) (CLASS event) (MASTER event)
      (ARGUMENTS ( (signal DSTRB) (value r) (direction out)
      (valid DO) (phase set) ))
      (IN_CONTROL (p3 p4) ) (OUT_CONTROL (p0) )
    )
    (NODE
      (NAME n1) (CLASS event) (MASTER event)
      (ARGUMENTS ( (signal DSTRB) (value f) (direction out)
      (invalid DO) (phase reset) ))
      (IN_CONTROL (p0) ) (OUT_CONTROL (p1 p4) )
    )
    (NODE
      (NAME n2) (CLASS event) (MASTER event)
      (ARGUMENTS ( (signal DO) (bitvectwidth DBITS)
      (bitvectbase 0) (value x) (direction out) ))
      (IN_CONTROL (p1) ) (OUT_CONTROL (p2) )
    )
    (NODE
      (NAME n3) (CLASS event) (MASTER event)
      (ARGUMENTS ( (signal DO) (bitvectwidth DBITS)
      (bitvectbase 0) (value s) (direction out) ))
      (IN_CONTROL (p2) ) (OUT_CONTROL (p3) )
    )
  )
)

(CONTROLLIST
  (EDGE
    (NAME p0) (CLASS control)
    (ARGUMENTS ( (min "3*period/(CBITS+2)") ))
    (IN_NODES (n0) ) (OUT_NODES (n1) )
  )
  (EDGE
    (NAME p1) (CLASS control)
    (IN_NODES (n1) ) (OUT_NODES (n2) )
  )
  ...
)

```

Figure 3-11 : Event Graph for *opticalR* Module in *afi* Text Format

THOR views a system to be hierarchically composed of modules that are modelled using the C language with some extensions.

The tool *MakeThorSim* from LAGER is used to convert the *structure_instance* view of the board into the netlist format accepted by THOR. The details of this process are described in [Brodersen92]. However, briefly, *MakeThorSim* traverses the *structure_instance* view hierarchy of the board in a depth-first fashion until it hits a sub-cell containing the attribute THOR_MODEL. All the modules in the board level primitive component library have associated THOR models in a .THOR file. These models are stored in the THOR_MODEL attribute in the corresponding *structure_master* view during the process of library installation, and later on the THOR_MODEL attribute is copied into the corresponding *structure_instance* view by *DMoct*. This process guarantees that *MakeThorSim* will always find the THOR_MODEL attribute in the leaf nodes of the design hierarchy. The output of *MakeThorSim* is a netlist in THOR format, together with the C models for each of the sub-modules. These are then used by THOR to simulate the entire board at the bit level.

Although THOR is the simulator that is currently used, the library organization and tool policies are modular enough that other simulators, such as VHDL, can be used in a similar fashion. Adding a new simulator requires a tool similar to *MakeThorSim* for generating input for the simulator, and simulation models for the various modules in the primitive component library.

3.7 Board Example

So far only the hardware module generation part of the SIERA environment has been discussed. Just these hardware module generators can often be used fruitfully in isolation from the software and architecture generation utilities available in SIERA that are described in the following chapters.

A good example of such a board whose design used only the hardware module generators is the

robot peripheral board that is part of the robot control system presented in Chapter 8. Here we present another board that made such stand-alone use of the hardware module-generators.

3.7.1 DSP Multiprocessor Board

This example is an interesting one because it was done in less than three months by a graduate student researcher from the DSP group at Berkeley who had no previous board or system design background. The three month time period included the learning curve associated with getting familiar with the various tools. The board is part of a multi-processor system with a special shared memory architecture called *Ordered Memory Access* described in [Lee90]. The system being developed by the DSP groups is made up of one or more identical interconnected boards. The key feature of the architecture is that the access to the shared memory is granted to the processors by a central controller (called MOMA) according to a static schedule. This is in contrast to conventional shared memory architectures where the processors request access to the shared memory. Once access to the shared memory is granted to a processor, it is not released until the processor has completed a shared memory transaction. The order of accesses to the shared memory is determined by a fully static scheduler and loaded into the central controller as a list of memory transactions. An advantage of such fully static scheduling is that no hardware or software semaphores based synchronization is required. Such fully static scheduling is possible for a subclass of dataflow graphs called *Synchronous Dataflow Graphs* that lack data dependency. Many important algorithms, particularly in digital signal processing, belong to this category. The ordered memory access architecture offers an efficient and low-cost architecture for such applications.

Figure 3-12 shows a block diagram of the board and Figure 3-13 shows the corresponding hierarchy of SDL files. It consists of four processor modules using the MC96002 32-bit digital signal processor as the CPU. Each processor module contains 1 Mbyte of fast SRAM and some glue logic for initializing the CPU mode. This particular CPU has two identical memory ports, one of which is used for the local SRAM mentioned above and the other is brought out of the processor

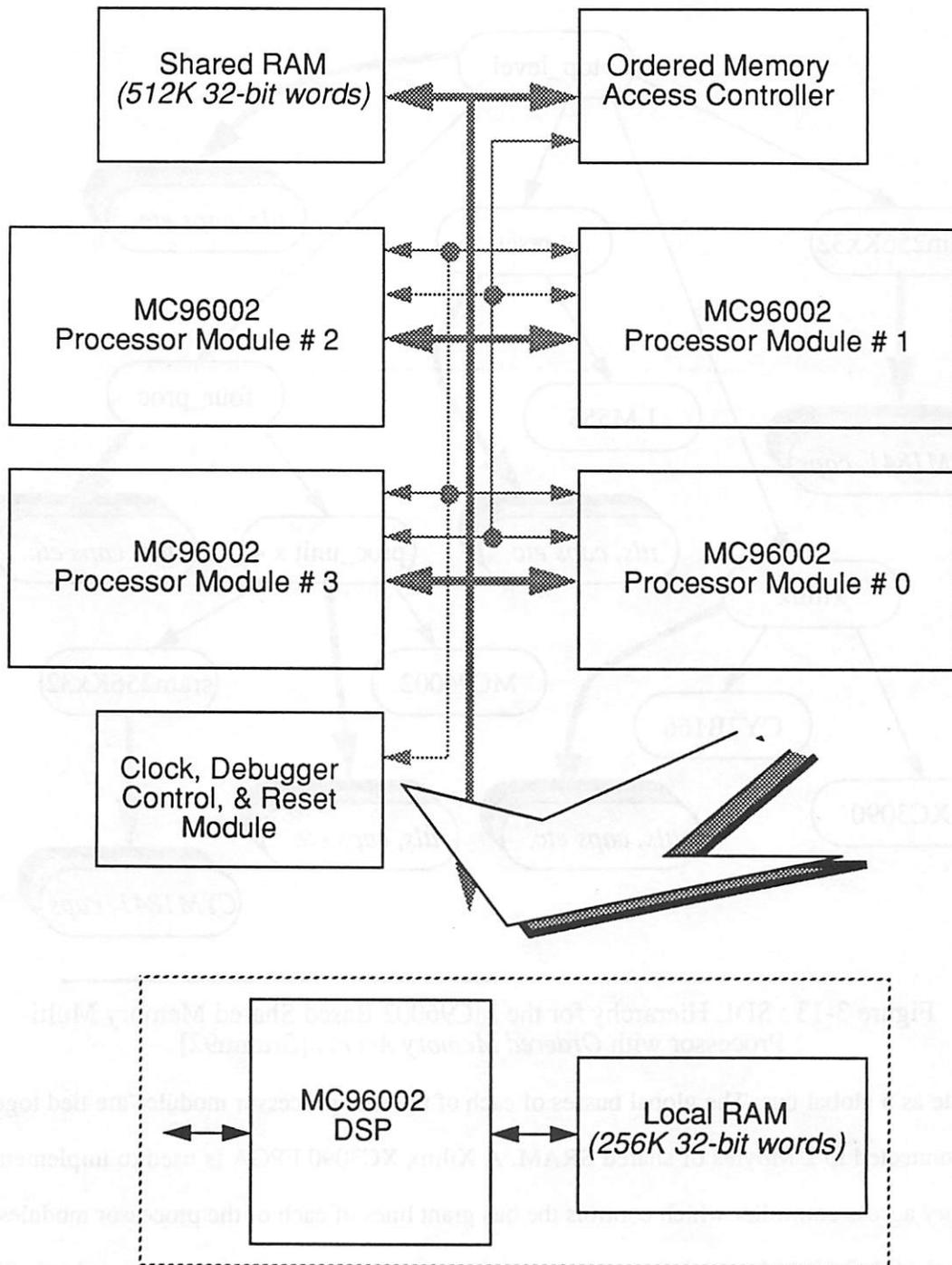


Figure 3-12 : Board Architecture of the MC96002 Based Shared Memory Multi-Processor with *Ordered Memory Access* [Sriram92]

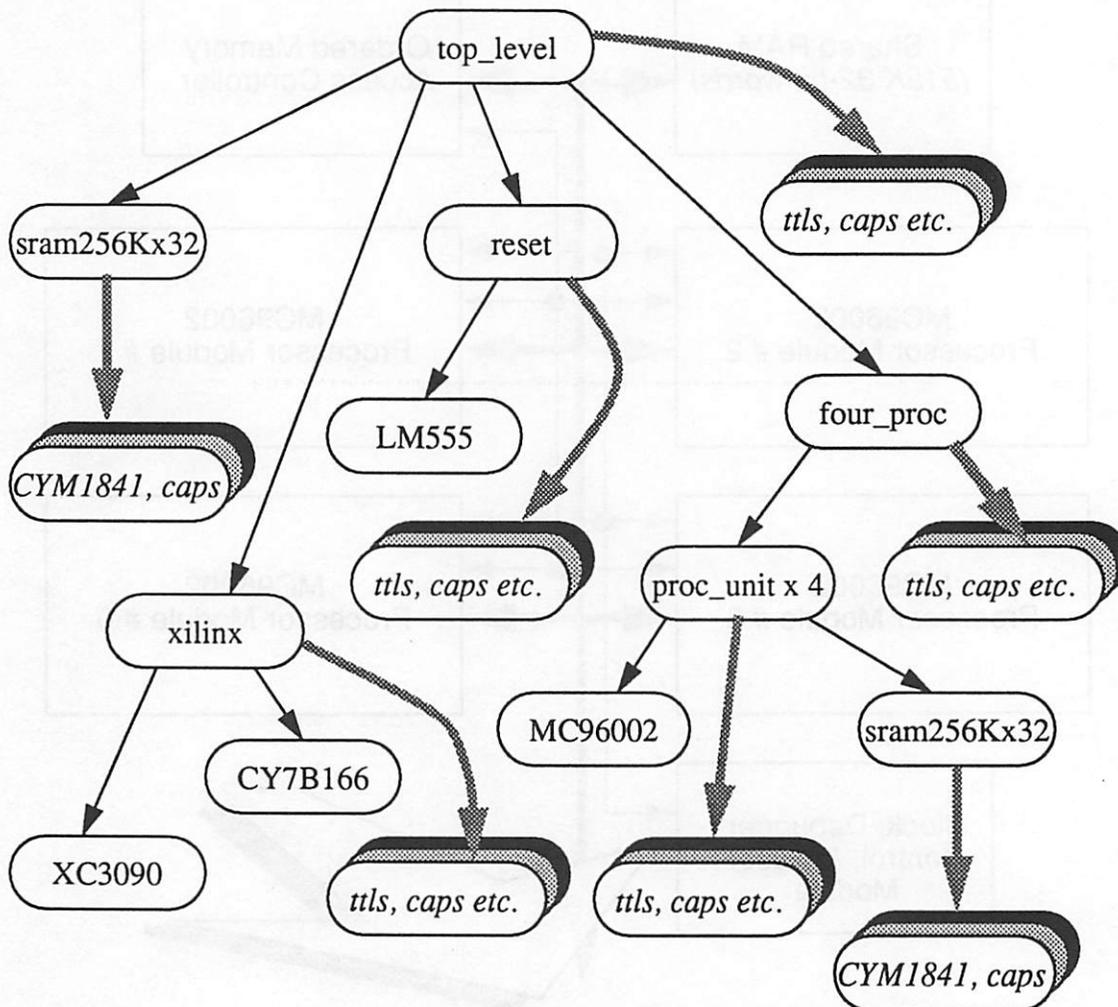


Figure 3-13 : SDL Hierarchy for the MC96002 Based Shared Memory Multi-Processor with *Ordered Memory Access* [Sriram92]

module as a global bus. The global busses of each of the four processor modules are tied together and connected to 2 Mbytes of shared SRAM. A Xilinx XC3090 FPGA is used to implement the memory access controller which controls the bus grant lines of each of the processor modules. An external 16Kx8 SRAM module is used to store the memory access schedule which is loaded from a workstation host. The Xilinx FPGA is also connected to the global bus for loading the program code into the processor modules on initialization. The host interface is also mapped to the Xilinx FPGA. Finally, multiple boards can be connected in a chain by modifying the logic in the Xilinx

FPGAs so that they act as gateways.

This design project made use of some of the existing memory modules and in turn contributed the MC96002 based processor module which is easily reusable in other designs. More importantly, variations of the board with different numbers of processors can be generated in a very short time. The physical layout of the board is shown in Figure 3-14. As is evident from the figure an extensive use of tiling-based placement was made in the design of this board. Further, the four processor modules have horizontally and/or vertically flipped versions of the same floorplan, resulting in a very compact and efficient layout. Table 3-4 summarizes the characteristics of the board.

Dimensions	13.6" x 7.7"
Number of Layers	10
Number of Components	230 parts + 170 bypass capacitors
Design Time	3 man-months, including time for learning the tools
Amount of SDL Code	2800 lines

Table 3-4 : Main Features of the MC96002 Multi-Processor Board

3.8 Summary

The board level hardware module generation framework presented in this chapter is now quite mature at the structural level, and in doing placement and routing. However, a key component that is missing is the extraction of physical information from the final placed and routed board for doing accurate performance analysis. Due to increasing clock frequencies, transmission line effects have become important so that the ability to extract capacitance, inductance and resistance to do a transmission line analysis is very important.

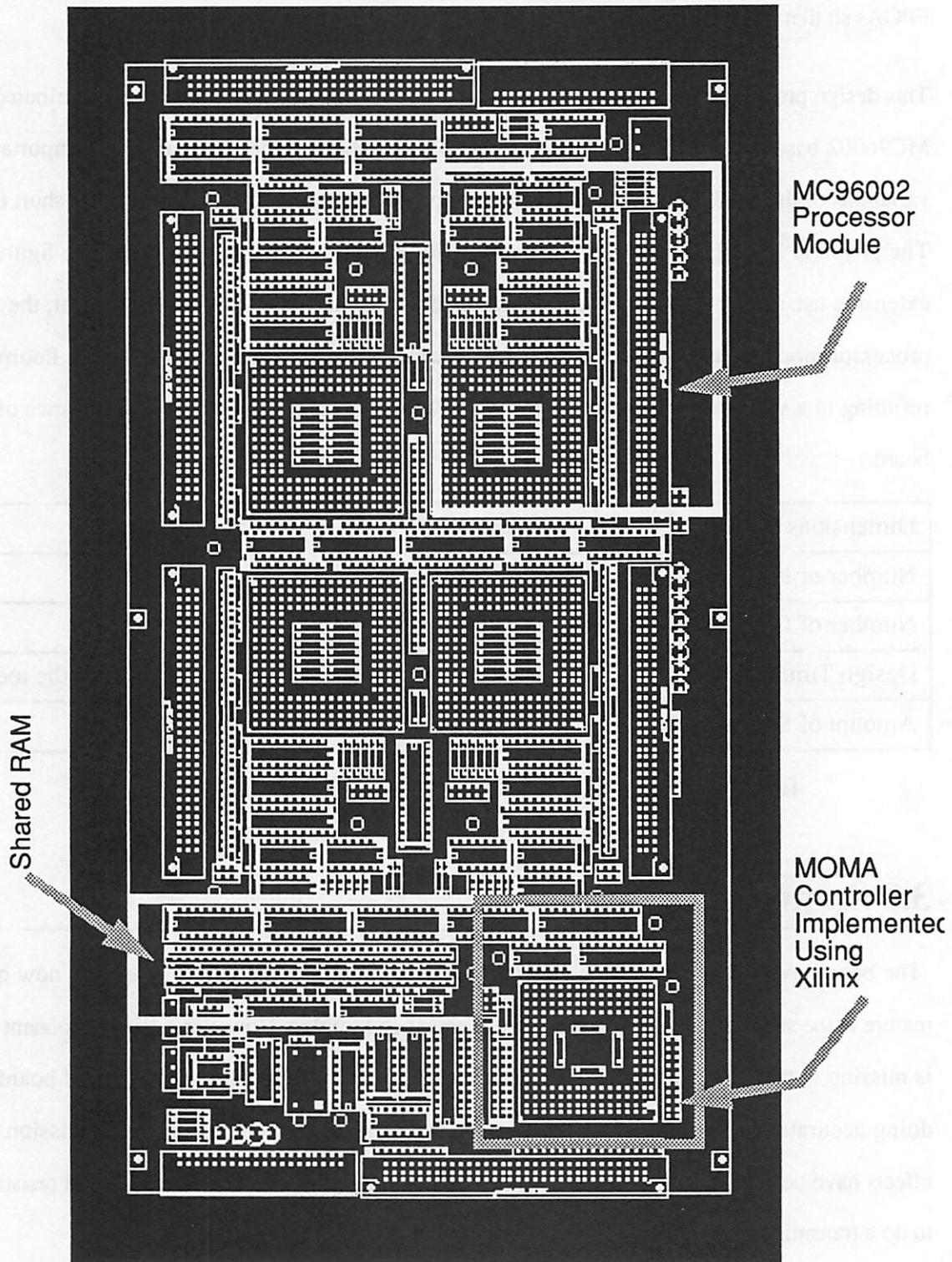


Figure 3-14 : Physical Layout of the MC96002 Based *Ordered Memory Access* Shared Memory Multi-Processor Board

CHAPTER 4

SOFTWARE MODULES

The use of parameterized hardware modules has played a major role in enabling automatic generation of chips as well as boards. Modularity brings with it the advantages of reusability and standardized interfacing to simplify the task of composing complex modules from simpler ones. It is similarly important in software as demonstrated by the efforts towards modular programming styles, object oriented languages, and reusable software libraries. However, achieving modularity has arguably been less successful in the software domain - reusable software libraries as well as automatic code generation (software module generation) are for most purposes still not available. This is a result of two factors. First, software is an inherently more flexible medium than hardware, and hence more prone to ad hoc approaches. Second, software systems usually have a much higher logical complexity.

In light of the above, the goal of the work described in this chapter is not to solve the problems of general purpose software module generation and reusable libraries. Instead, the focus is on the software modules and techniques needed to design systems using the hardware modules described in the previous section. This includes the system and application software that actually runs on the

software programmable processor modules constituting the system, as well as the support and development software needed for developing the system.

4.1 Software Issues in Application-Specific Systems

In order to study what are the software issues relevant in the context of application specific systems being investigated in this thesis, Figure 4-1 shows the hardware organization of a typical system that can be built using the hardware module generators and libraries described in the previous chapter, together with some off-shelf hardware.

The example system is characterized by being composed of a number of heterogenous programmable processors that are spread over a workstation, an off-shelf board, and two custom boards. The processors are connected using some suitable interconnect hardware. The custom boards also have ASICs and other dedicated hardware coupled to the programmable processor

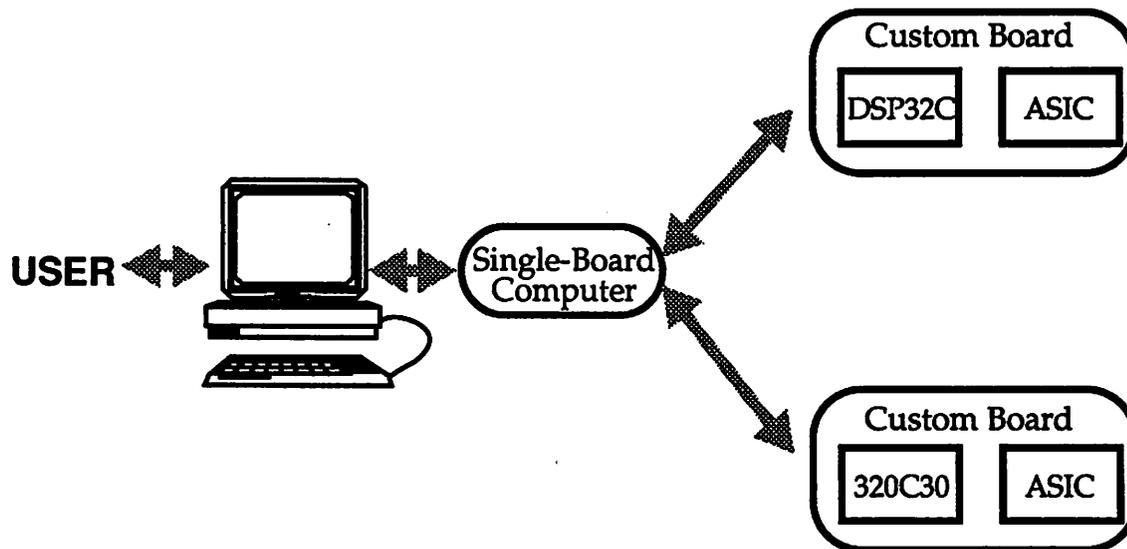


Figure 4-1 : Organization of a Simple System Using Off-Shelf Hardware and Module Generators and Libraries from Chapter 3

modules on those boards.

The logical functionality of the system is implemented by the dedicated hardware on the custom boards together with software executing on each of the processor modules. It is the problems relating to the development and modular organization of this software that are being addressed here. Following are the important problems that were identified:

- a. *Software Modules Organization*
 - what is an appropriate form of software module?
 - what is a suitable model of software organization?
 - how can multiple software modules be mapped to a single processor?
 - how are the software modules distributed across multiple processors?
- b. *Communication*
 - how do the software modules communicate with each other?
 - how do the software modules communicate with the dedicated hardware modules?
- c. *Run-Time and Development Environment*
 - how does the user interface with the software modules?
 - how are the software modules developed, down-loaded, monitored, and debugged?
- d. *Generation of Software Modules from High-Level Description*
 - is it feasible to have some form of automated software module generators?

It is highly desirable that the solutions to the above problems be applicable to parameterized processor modules, and encourage reusability. In particular, one would like to identify software modules and other support software that, with appropriate configuration, is reusable across designs and then encapsulate them in libraries just like hardware module libraries. The solutions to the above problems are discussed in detail in the following sections.

4.2 Software Module Organization

Software, unlike hardware, is a flexible and abstract medium, as a result of which many different ways of organizing software systems exist. It is therefore not obvious as to what is an appropriate form of software module. This problem is not present in the case of hardware where modules are naturally defined to be concurrently operating parts of a hardware system that interact using electrical wires. In the case of software multiple modules may be mapped to a single physical

processor which is an inherently serial device. Therefore concurrency alone is not a sufficient criterion for defining a software module. The nature and organization of software modules depends on the control mechanism used to orchestrate them. This section explores the alternate control mechanisms, and tries to determine a suitable form of software module.

4.2.1 Existing Control Mechanisms for Software Modules

There are three fundamental module control mechanisms that are commonly used in software and relevant to this work - *subroutines*, *coroutines*, and *processes*. As shown in Figure 4-2, these mechanisms differ in how the control is transferred between the modules.

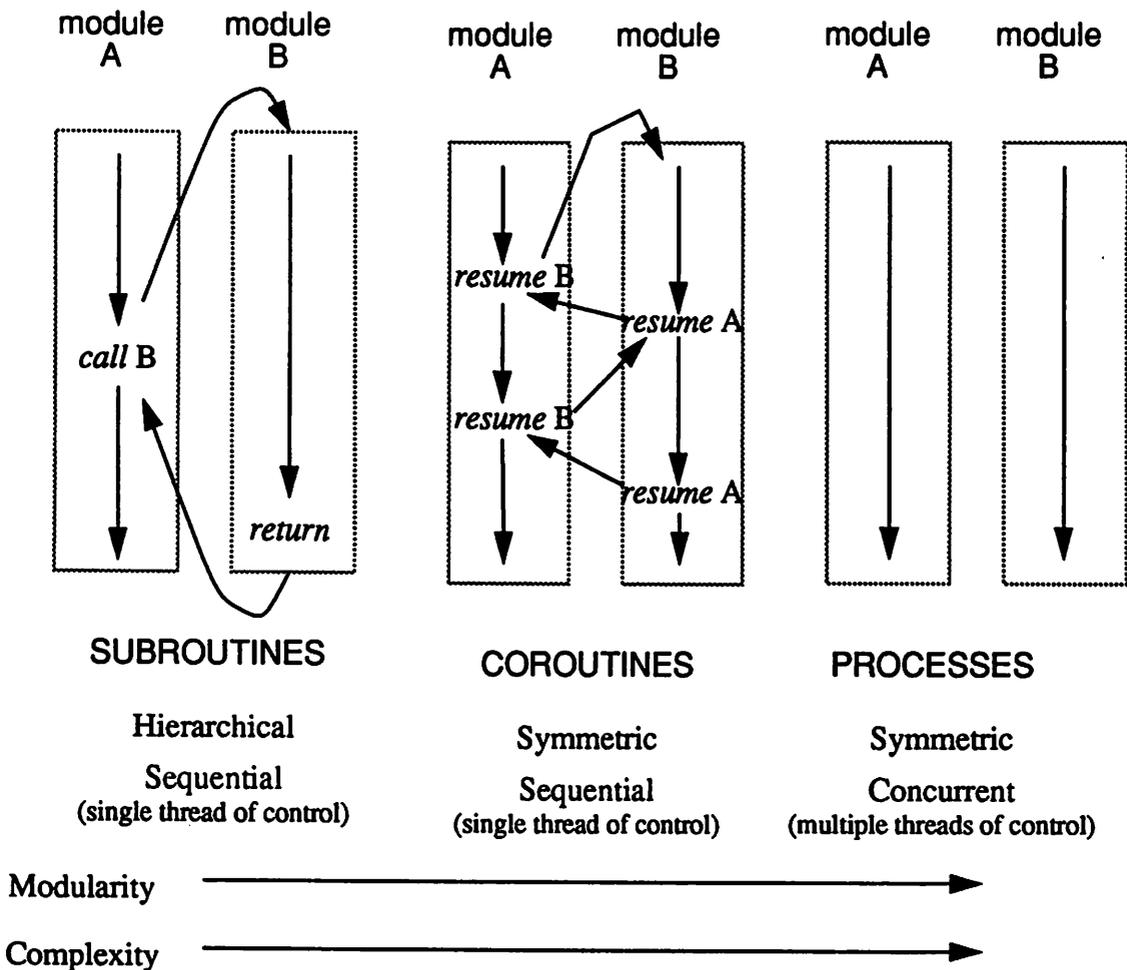


Figure 4-2 : Control Mechanisms for Software Modules

Subroutines

In subroutines the modules have a hierarchical master-slave relationship: the master module invokes the slave module using a *call* statement, and the control is always returned to the master by the slave using a *return* statement after the slave has done its job. This mechanism leads to efficiencies by allowing several masters to share a single slave, and is considered so useful that it is built into the instruction sets of most processors. Subroutines encourage hierarchical decomposition of the system leading to many dependencies among the modules. Further, subroutines are an inherently serial control mechanism - a master module is idle while the slave module is doing its task. This is true even if the two modules are mapped to different processors, such as in the case of remote procedure calls (RPC). Both these attributes - hierarchical decomposition and inherently serial control - not only detract from modularity but also render subroutines unsuitable in the case of multiple processors where true concurrency is available.

Nevertheless, it certainly is possible to fake concurrency and avoid hierarchical decomposition using subroutines when multiple modules share a processor - a supervisor module is designated as the master to all the application modules which are then repeatedly invoked as subroutines by the supervisor module. This is an acceptable way of organizing concurrent modules if the order in which they need to be invoked can be statically determined, and is cyclic. For example, this is the case with systems composed of *Synchronous Data Flow* modules [Lee87]. This is also extensible to the case of multiple processors. However if a good static schedule of invocation cannot be statically determined then this scheme can lead to poor utilization of the processors. This is usually the case in the presence of event-driven reactive modules which exhibit asynchrony and non-determinacy, such as modules interacting with external I/O devices.

Coroutines

Coroutines [Andrews83] are a generalization of subroutines where the control is transferred between modules in a symmetric rather than a strictly hierarchical fashion. Control is transferred

between modules by the *resume* statement, the execution of which transfers control to the named module. When control returns to a module, it starts executing at the instruction following the last *resume* executed by it. If multiple modules share a processor then enough state information about a module executing *resume* needs to be stored so as to enable control to be returned to the instruction following the *resume*. This makes coroutines more expensive than subroutines.

Although a fundamental concept, coroutines are a relatively non-standard control mechanism and most computer programming languages do not support them directly. They form a conceptual bridge between subroutines and processes. Coroutines are particularly useful to handle properly modules that retain state information between successive invocations. It is worth noting that subroutines can easily be implemented using coroutines although it may not be efficient to do this, but the reverse is difficult and inelegant to implement. Also, conceptually there is nothing that prohibits calling a subroutine type module from within a coroutine module.

The symmetric control transfer of coroutines make their organization more egalitarian than the strict hierarchical relationship imposed between modules by a subroutine structure. As a result coroutines can be used easily and efficiently to organize multiple concurrent modules that share a single processor. This, for example, is the case with the architecture model used by the Vulcan-II hardware-software co-design system (see Chapter 1 and [Gupta92a][Gupta92b]) where only one software programmable processor is allowed.

However, coroutines are not a suitable control structure in the presence of multiple processors. Their semantics allow for only one thread of control. In other words, only one module has the control and is executing at any given instant. The single thread of control that is inherent to the semantics of coroutines make them unsuitable as a solution to software organization for systems designed using the hardware module generation techniques presented in the previous chapter. Using those techniques it is extremely easy to build systems that have multiple programmable processor modules. These systems therefore are inherently capable of supporting multiple threads

of control which a coroutine based software organization will not be able to exploit.

Processes

Processes are a generalization of coroutines, and are the most powerful of the three control mechanisms. A process module has its own thread of control and is scheduled independently and separately from other process modules. This makes them extremely attractive conceptually when multiple processors are present to provide true concurrency. The concept of processes is useful even when multiple process modules are mapped to a single processor as their independent threads of control lead to more modularity than provided by coroutines and subroutines. Although in the single processor case only one process module can have the control at any given instant, because of the inherently serial processor, it is certainly possible to simulate concurrency by using a *kernel* to multiplex the modules onto the single processor. As a result the multiple process modules sharing a processor *appear* to run concurrently. The kernel implements the transfer of control between the process modules, and also uses some policy to schedule these transfers, called *context switches*. In other words the kernel helps in interleaving the execution of multiple process modules on a single processor to give the appearance of concurrency. Information about the state of the process module needs to be saved, just like in the case of coroutines, in order to resume execution when the control is transferred back to the process module.

Process is a very fundamental and widely used concept. Multitasking operating systems, for example UNIX, support this concept as do languages like ADA and OCCAM. Further, some processors support the notion of processes, providing support for process scheduling and context switching. Examples of such processors are INMOS Transputers and Intel i960 series. It is easy to implement coroutines using processes. In fact coroutines can be viewed as concurrent processes mapped to a uniprocessor where the context switching is completely specified rather than being done by the kernel. Also, conceptually nothing prohibits the use of multiple coroutines and subroutines inside a process.

While processes are the most versatile, elegant, and modular of the three control mechanisms discussed, they are also the most expensive to implement. Their independent and concurrent threads of control require that either every process module be mapped to its own dedicated processor, or a kernel be used to do scheduling and context-switching of multiple process modules sharing a processor.

Finally, while processes have independent concurrent threads of control, it is not very useful if the processes are isolated. Most meaningful systems will require that the processes co-operate so that their composite behavior implements the desired system functionality. Co-operation may require that the processes communicate, share data, and synchronize.

4.2.2 Using Processes as Software Modules

From the discussion in the previous sub-section it should be clear that when multiple processors are available, thus making true concurrency available, processes provide the only control mechanism that makes sense. This makes a strong case for using processes as the software modules. However, one can argue against this on grounds of efficiency. It is possible to use coroutines or subroutines as modules within each individual processor, and then viewing all the coroutines or subroutines on a processor together to constitute a single process. This gives a process view at the top level while retaining coroutines or subroutines underneath.

Despite these valid efficiency concerns the model used for organization of software modules in this work is that of processes at both levels - within a processor and across multiple processors.

There are several reasons for this, as listed below:

- a. It is conceptually elegant as processes provide the maximum modularity and concurrency. It forces much less inter-module interaction than subroutines or coroutines because the control structures of the modules are completely separated. This often results in less stringent timing constraints.
 - b. It is easier and more efficient to migrate a process across processors than is it to migrate subroutines or coroutines.
-

-
- c. The notion of processes with their independent threads of control provide a unifying theme between hardware modules and software modules. The hardware modules are inherently concurrent being a separate physical entity co-existing with other hardware modules. In fact one can view a dedicated hardware module to be a process running on its own dedicated processor. In fact hardware modelling languages, such as VHDL, Verilog, and Hardware-C, have adopted this process view of hardware. This point is important in this work because both software programmable processors and dedicated hardware modules form an integral part of the systems that are of interest to us.
 - d. Dedicated hardware modules that interface to a software programmable processor often require interrupt handlers. These interrupt handlers can also be viewed as processes except that they are scheduled outside the control of the scheduler in the kernel managing the process modules. These interrupt handlers appear to execute concurrently with the process modules, and share similar problems and attributes.
 - e. Implementation of kernels for many processors are available, thus simplifying the task of demonstrating these ideas.
 - f. The notion of processes is also very useful in describing the behavior of systems. Processes very nicely capture the coarse granularity concurrency present in the real-time reactive systems that are the focus of this work, as described in Chapter 1. For example, the robot control system described in Chapter 1 can naturally be structured as a number of processes which all apparently run in parallel.

The formalism chosen for specifying the high level behavior of systems in this work is that of a network of processes. This formalism is described in detail in Chapter 5. An important point to be noted is that the processes used to specify a system need not have a one-to-one relationship with the process modules, hardware or software, in the implementation. Still, having the notion of processes for specification as well as implementation certainly simplifies the implementation.

Of course nothing comes for free - a drawback of the process approach when compared to subroutines and coroutines is that it is much more difficult to estimate and evaluate its performance. This is due to the loose coupling between the multiple threads of control which makes it extremely hard to predict the actual order of various events during execution of the system.

4.2.3 Implementation of Process Modules

The arguments in the previous section should convince one that processes are indeed a desirable form for structuring software modules. The implementation of this approach is however not

straightforward. Unlike subroutines, very few processors support the notion of processes directly in hardware. Therefore supporting software is needed to implement the processes. Usually in real-time systems this is often done in an ad hoc fashion with little sharing of efforts across different projects. To avoid this repeated re-invention of the wheel, and to also make the implementation of processes easier, it is desirable if an operating system with a multi-tasking kernel is used.

There are two possible approaches to using multi-tasking operating system kernels in a system composed of multiple processor modules generated using techniques of Chapter 3. The first approach would be to use a *distributed* kernel that can simultaneously handle processes mapped to all software programmable processor modules in the system. Unfortunately the technology of distributed kernels is not very mature, and the design of a distributed kernel is further complicated in our case by the fact that the processor modules can be heterogenous, and be connected in arbitrary topologies.

The second approach, which is the one used in this work, is that of using multiple autonomous kernels with one kernel being associated with each processor module. The kernel on each processor module manages only the processes that are mapped to it. To accomplish this the kernels need to be able to create separate processes, schedule their execution, handle communication between processes, and synchronize processes with each other and with external events.

Figure 4-3 illustrates the organization of software. The disadvantages of this approach over the distributed kernel approach include under utilization of processors, and a lack of handling of inter-process communication when two processes are on different processors.

A configurable real-time multi-tasking operating system kernel has been associated with each processor module in the hardware module library described in Chapter 3. Ideally one would like to have identical kernels on each processor module - this would have the advantage of identical semantics. However currently a mix of existing commercial or home brewed kernels is being used in order to demonstrate the concept. Table 4-1 lists the various processor modules and kernels

PROCESSOR	KERNEL	COMMENTS
TMS320C30	SPOX	Priority Driven Preemptive Multi-Tasking Kernel
DSP32C	VDI	Locally Developed Foreground-Background Kernel
MC96002	(SPOX)	(not available yet)
MC68020/30	VxWorks	Priority Driven Preemptive Multi-Tasking Kernel
SPARC	LWP	Priority Driven Preemptive Multi-Tasking Kernel

Table 4-1 : Currently Supported Processor Modules and Associated Kernels associated with them.

These kernels provide the ability to map multiple process modules to a single processor. Most of these kernels - SPOX, VxWorks and LWP - are true multitasking kernels which can manage an

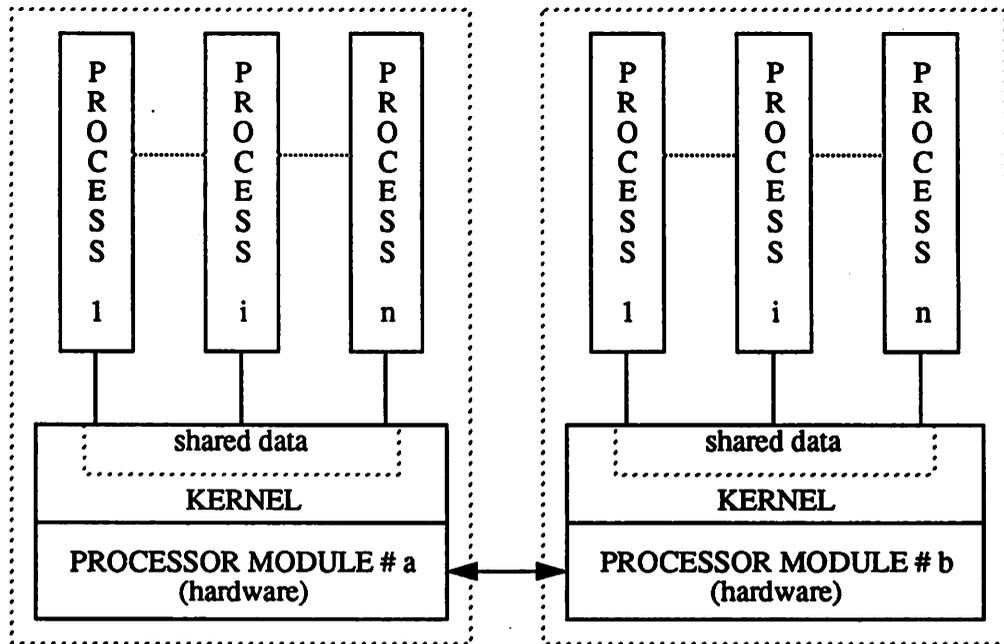


Figure 4-3 : Implementation of Process Modules by Multiple Autonomous Kernels

arbitrary number of process modules. All these kernels provide support for real-time behavior by supporting pre-emptive, priority driven scheduling of the process modules. Basically, a priority is associated with each process module. At any given instant the highest priority process module that is ready to execute is given control of the processor by pre-empting the currently executing process module. The thread of control associated with a process module is blocked if it has to wait for a synchronizing event from another process within the processor, or from a source external to the processor.

In some instances, like VDI for DSP32C, the kernels are foreground-background kernels. These are essentially multi-tasking kernels with only two priority levels. Multiple process modules are allowed at the higher priority level while a single low priority process module executes in the background. The higher priority processes are implemented as interrupt handlers.

These kernels typically provide a subroutine interface to let the processes access the various services provided by the kernel. Because of our decision to use a mix of different kernels there are different subroutine interfaces associated with each kernel. Further there are subtle as well as not so subtle differences in the semantics across the kernels. To mitigate the situation somewhat a common extra layer of subroutines is used on top of each kernel at the cost of performance penalty. This is however not the best solution - the real solution to these problems is to develop a common and simple kernel that can be ported across the different processor modules.

In the previous sub-section it was mentioned that the notion of processes is equally applicable to hardware as well as software, thus providing a common thread between the two for purposes of representation and simulation. This uniformity can be taken one step further. The hardware modules usually plug in a bus structure which has specific electrical, functional, and synchronization requirements. Similarly the software modules (processes) plug into a *software bus* that is defined by the interface to the kernel being used to manage the processes. The kernel is like the bus controller and imposes data, functional, and synchronization requirements on the software

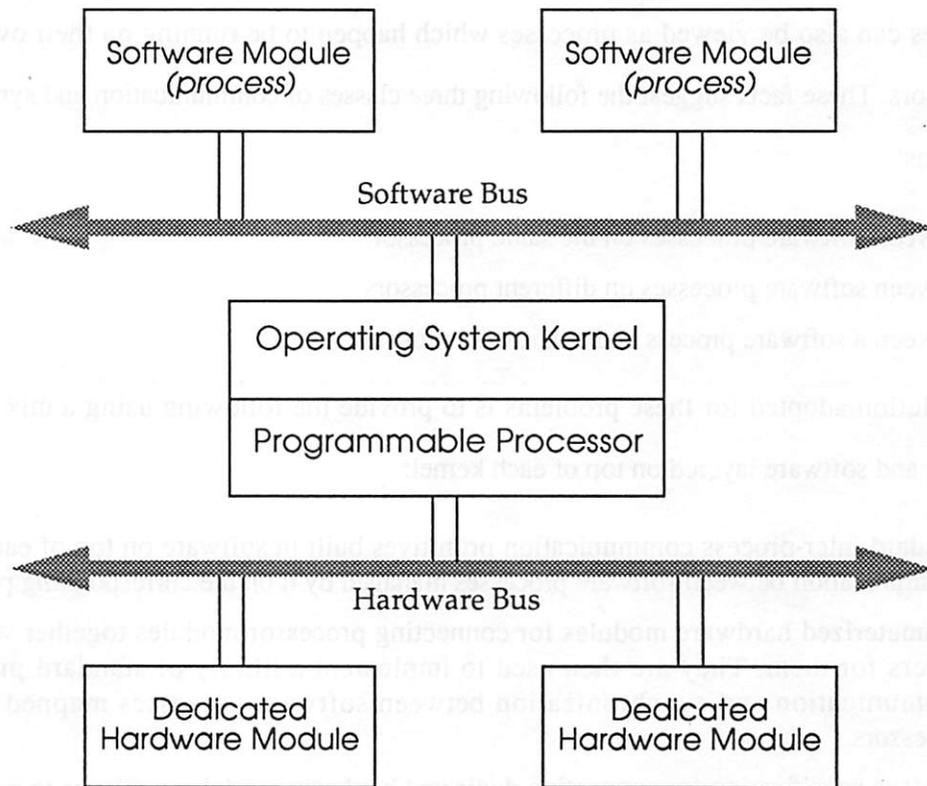


Figure 4-4 : Similarity between Organization of Hardware and Software Modules in a System

bus defined by it. Figure 4-4 illustrates this interesting similarity between hardware and software. This analogy provides much of the motivation behind a layered template based approach to generation of system architecture that will be presented in Chapter 6.

4.3 Communication and Synchronization of Software Modules

Earlier in this chapter it was suggested that processes executing in isolation are not useful for most systems of interest to us. This cooperation takes the form of communication and synchronization between processes that are in general mapped in a many-to-one fashion onto one

or more software programmable processors. Further, as already mentioned, dedicated hardware modules can also be viewed as processes which happen to be running on their own dedicated processors. These facts suggest the following three classes of communication and synchronization problems:

- a. Between software processes on the same processor
- b. Between software processes on different processors
- c. Between a software process and a hardware process

The solution adopted for these problems is to provide the following using a mix of hardware support and software layered on top of each kernel:

- a. Standard inter-process communication primitives built in software on top of each kernel for communication between software processes managed by it on the corresponding processor.
- b. Parameterized hardware modules for connecting processor modules together with software drivers for them. They are then used to implement a library of standard primitives for communication and synchronization between software processes mapped to different processors.
- c. Interface specification for connecting dedicated hardware modules as slaves to programmable processor modules, and software drivers for them. Communication and synchronization primitives are then built on top of this to let a software process interact with the dedicated hardware module as if the hardware module was another software process.

The basic approach in implementing the above three is to first provide a lowest layer of synchronization and communication primitives using a mix of hardware and software best suited for the processor and the kernel. The desired standard primitives are then implemented on top of this low-level layer, somewhat akin to the conventional layered approach to computer communication and networking.

The choice of the standard communication and synchronization primitives depends on the architecture model being used for the system. It is not meaningful to talk about the implementation without describing the system architecture. Therefore the discussion on the hardware and software implementation of the communication and synchronization primitives is being deferred to Chapter 6 after the model of system architecture used here has been presented.

4.4 Run-Time and Development Environment

One feature that distinguishes software modules from hardware modules is the importance of user interface modules, and the need for a run-time environment for developing, down-loading, and monitoring the software modules. Interaction with the user as well as with a host computer are much more important in software. On the positive side the implementation of this supporting software is considerably eased in our process based organization of software. Special software processes are run on the various programmable processors under the same kernel as the software process modules implementing the system. These special processes provide a consistent run-time environment for user interface, file input and output, program loading etc. This is thus a trivial by-product of our use of multi-tasking kernels, and results in a considerable saving in time often wasted in ad hoc implementations for custom hardware systems.

Following subsections describe the implementation of the three main classes of services that are currently provided as part of the run-time environment. A full appreciation of these services may require reading about the system architecture model in Chapter 6.

4.4.1 Software Module Loaders and Initialization Utilities

A standard set of utilities have been implemented to initialize and bootstrap the software programmable processors on the custom board, and to download the software modules and kernels. One of the processors in the entire system is required to be the workstation where the software modules are developed and the kernels configured using appropriate cross-development tools. This processor is treated as the root, and then all the software programmable processors are initialized by a depth-first traversal of the system interconnect graph.

There are two distinct parts to these utilities. First is a *loader* which understands the object-file format for all the supported processor and then uses a lower level bootstrap mechanism to download the memory image to each processor.

The second is the utility to bootstrap each processor. The current implementation requires that every processor either has a self-contained bootstrap ROM using which it goes to a ready state on reset, or that it has a shared memory area with the processor that is its master in the tree formed by the depth first traversal such that the processor goes to a location in the shared memory on reset. The bootstrapping is done in two phases using the shared memory. A tiny bootstrap program is downloaded into the shared memory and the processor is reset. This bootstrap program then communicates with the downloader process on the master processor (which has already been initialized) using message queues implemented on the shared memory. A simple and portable message queue package, called SQ, has been implemented for this. The queues implemented by SQ are created in the shared memory with an arbitrary size buffer area, and a header. The access routines to the queue allow arbitrary sized messages to be sent back and forth. If the message size is larger than the buffer area, then it is automatically broken up into smaller packets, and then re-assembled on the other end. This allows complete contiguous sections of the object file of the boot image to be downloaded with one message passing call. The functions to send and receive messages allow the process making the call to either block (using busy-waiting) or return immediately in case the queue is not ready. Although meant for the bootstrap process, the SQ package is useful for other purposes too. Figures 4-5 and 4-6 respectively show the boot-strap process, and the functions available in the SQ package.

- Since the processor modules are parameterized, the amount of resources (for example, memory) available to them can vary from instance to instance. To avoid making these parameters hard-coded into the software, a set of parameters describing a processor module are also downloaded into an area in its memory during the boot-strapping process. These parameters include the sizes and addresses of the various types of memories, information about processors that are adjacent to this processor, and specification about the inter-processor hardware such as the address of the shared memory and hardware semaphores if any. These parameters are then accessible at run-time for the software to be configured appropriately.
-

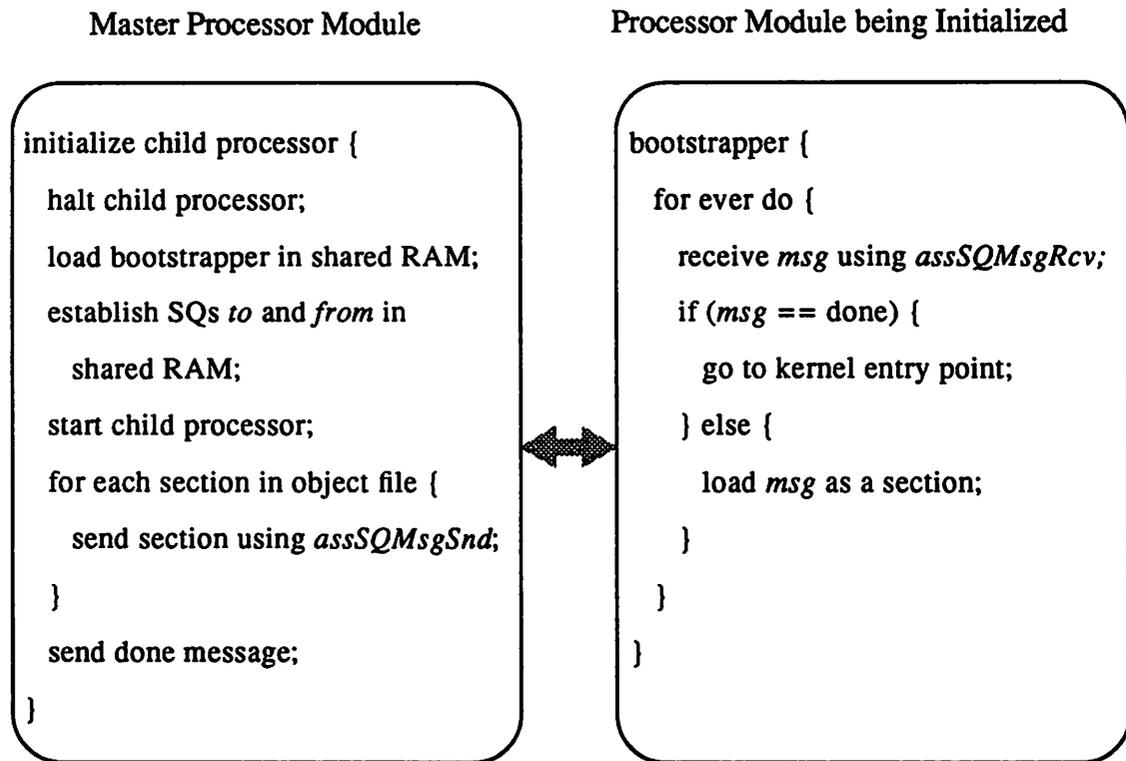


Figure 4-5 : The Pseudo-Code for Boot-Strapping a Processor Module

4.4.2 Client-Server Model Using RPC

Low level communication primitives such as message passing, as mentioned in the previous section and described in detail in Chapters 5 and 6, are useful as a general-purpose mechanism. However, at a higher level it is often useful to abstract them even further by imposing a structure on the way the processes communicate using message passing. Remote Procedure Call is one such mechanism that is widely used in general-purpose networked computing. In the case of systems using custom hardware it is often very natural to structure the user-interface as if the system was a *server* providing certain services. For example, in the case of a robot controller the high-level user-interface and planner process runs on a workstation and sends motion commands to the robot controller hardware. One way of looking at it is as if the robot control system is a *motion server* that receives requests from the planner and carries them out. Structuring such user-interface

```
/* functions to access a SQ queue */
/*
 * send a packet containing data from d with tag tag to queue sq and a timeout
 * timeout. The packet size n must be less than the buffer size of sq.
 */
extern assStatus assSQPckSend(assSQ sq, uint32 tag, int n, void *d, int timeout);
/*
 * receive a packet from queue sq storing the data, tag, and size of the received
 * packet in d, *ptag, and *pn respectively. The timeout specifies the behavior
 * when the queue does not have a packet ready. room is the room available in
 * the data buffer pointed to by d. A packet larger than size room is truncated.
 */
extern assStatus assSQPckRecv(assSQ sq, uint32 *ptag, int *pn, void *d, int room, int
timeout);
/*
 * send an arbitrary sized message with data d, tag tag, and size n. The message
 * is broken up into smaller packets if needed.
 */
extern assStatus assSQMsgSend(assSQ sq, uint32 tag, int n, void *d);
/*
 * receive an arbitrary sized packet into a buffer of size room pointed to by d.
 * The tag and actual message size are returned in ptag and pn. The message
 * is assembled from smaller packets if needed.
 */
extern assStatus assSQMsgRecv(assSQ sq, uint32 *ptag, int *pn, void *d, int room);
```

Figure 4-6 : The SQ Package for Simple Low-Level Message Passing

software in a *Client-Server* fashion is therefore very natural. The Remote Procedure Call model provides such an abstraction.

In order to allow rapid development of interactive user-interfaces for systems generated using the hardware and software module generation techniques of this chapter, a framework based on SUN's RPC has been developed. The user-interface process runs on a SUN workstation and is based on a commercial interactive C interpreter. This allows the user to access the system functionality by calling C functions and using the control structures of C - thus giving an extremely powerful programmatic interface. This user-interface process acts as an RPC client that generates requests for the dedicated system. A base protocol for controlling the system, such as initialization and bootstrapping, has been defined and the corresponding library is linked into the C interpreter. The protocol is extended for a specific system together with wrapper functions for executing application-specific tasks. For example, in the case of a robot system the protocol is extended with requests for moving the robot, initializing the robot, controlling its gripper etc. An RPC server for carrying out the tasks specified in these requests is then executed on one of the processor nodes in the system, which in turn passes the requests to appropriate processors. Currently implementation of such an RPC server exists only for the MC68020 module running VxWorks. The server uses the communication mechanisms discussed later in Chapter 6 to send messages to processors in the system that actually carry out the specified task.

Figure 4-7 shows an abstract view of an user-interface organized according to this client-server model, and a good example of an user interface based on this is given later in this thesis as part of the robot control system example.

4.4.3 File and Terminal I/O, and Other System Services

Real-time software modules executing on a processor module embedded on a custom board typically do not need to do any file and terminal I/O because such I/O is inherently non-real-time. However the capability to do such I/O even from these embedded processor modules is desirable

for two reasons. First, even for software modules with real-time constraints, such I/O is extremely useful during the debugging phase. A simple sprinkling of *printf()* in a C program is often enough to debug or monitor what is going on. Second, having such I/O allows simple user interface processes to be run on the embedded processors that exchange data with the user or a file. Given

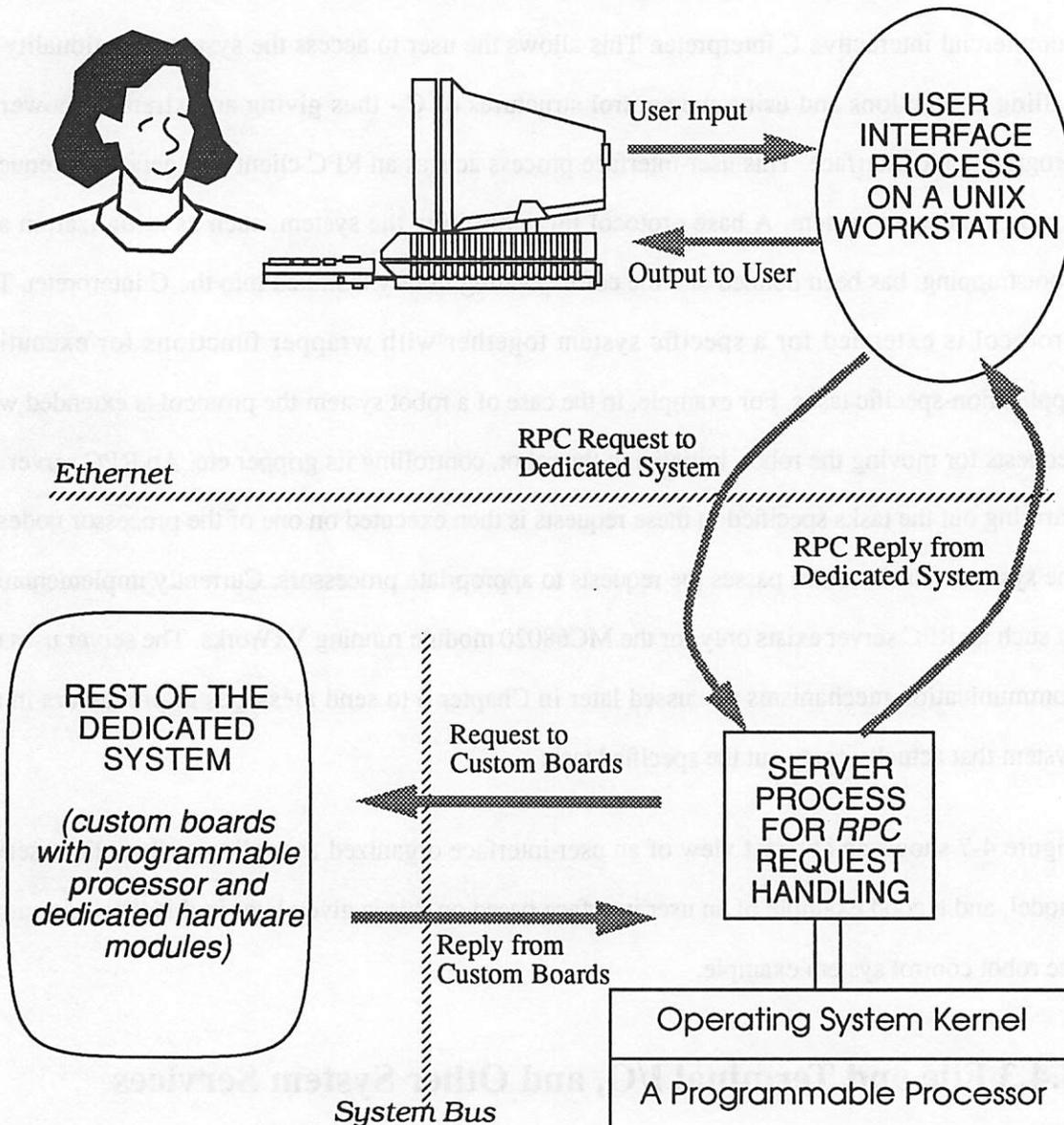


Figure 4-7 : Client-Server Model for User-Interface to a Dedicated System

our ability to do multi-tasking on every processor, such user-interface processes can easily be run in the background at a low priority.

This service is implemented using the RPC (remote procedure call) model except that unlike the previous subsection the embedded processor modules on the custom boards are the clients and not the servers. Figure 4-8 shows the basic organization of the implementation. There are three key components in the implementation.

First, one or more *server* processes are run on a workstation connected to the system, or any other processor that is part of the system and whose kernel has the ability to do file I/O. Each of these server processes waits for a request to arrive on a port, services the request, and then sends an answer back. At present such servers have been implemented for the UNIX workstation, and for the VxWorks kernel running on the MC68020 processor module.

The second part of the implementation is the protocol that is understood by the servers providing the file and terminal I/O services. The protocol defines a set of request and reply packets, and the contents of their fields. The protocol is modelled after the low-level UNIX file I/O calls, and has provisions for opening and closing files, file descriptors, reading and writing from file descriptors etc. The request packets correspond to the different subroutines and their arguments. The reply packets correspond to the return value or status. The protocol is synchronous in the sense that a reply packet is generated for every request packet - just like in the pure RPC model. How the packets are actually transmitted is immaterial, the only thing that is required is that there be an underlying communication library that allows a packet based communication in a reliable fashion (i.e., no re-transmission or packet re-ordering is needed). The protocol itself is easily extensible - in fact, the current implementation also uses the same protocol to provide services like getting the current time from the clock residing on a UNIX workstation, and the ability to execute arbitrary commands on a UNIX workstation by making a *system()* call. This ability of remote execution of UNIX processes is a powerful utility similar to the *rsh* command in UNIX.

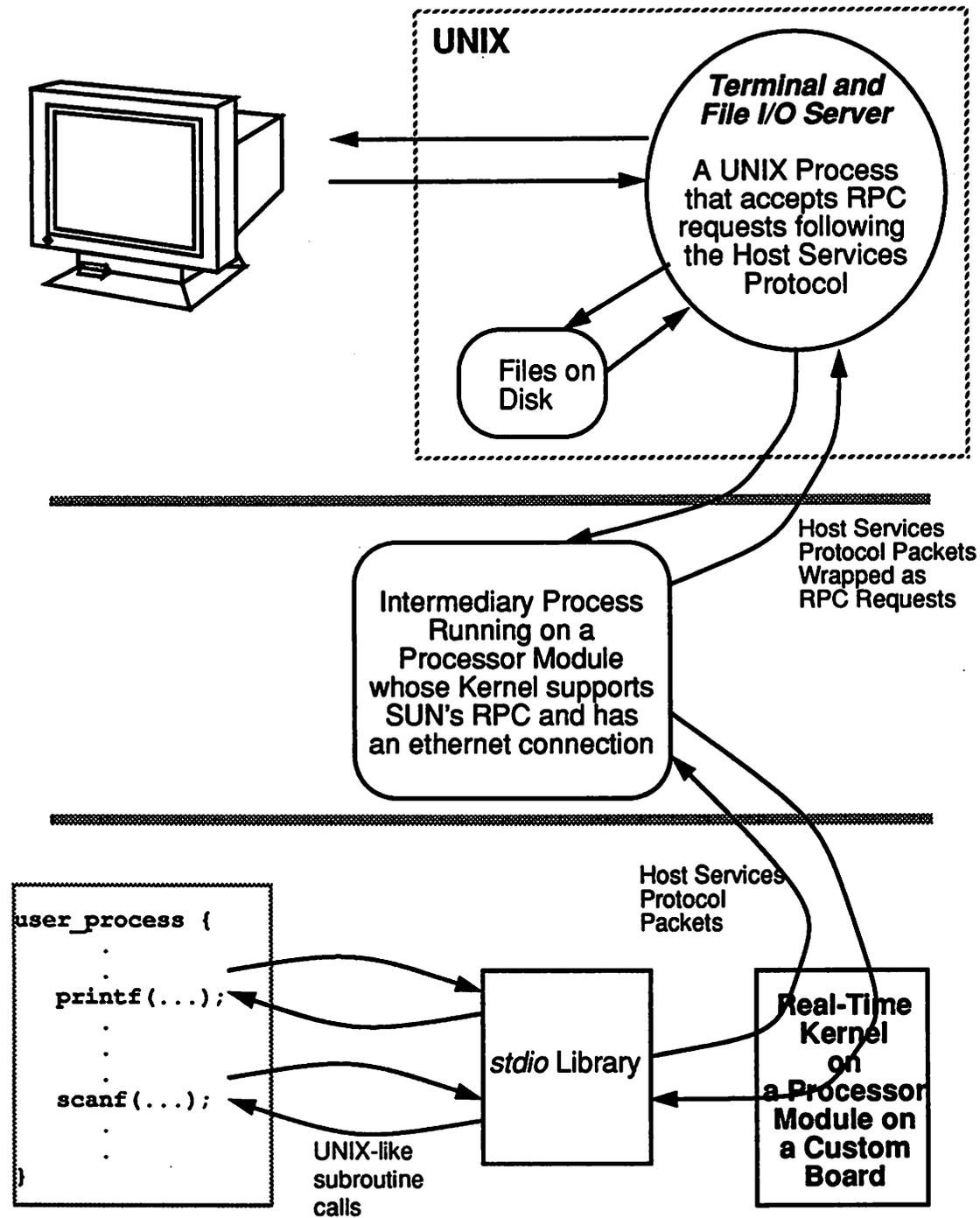


Figure 4-8 : Implementation of UNIX-like File and Terminal I/O for Programmable Processor Modules on Custom Boards

The third part of the implementation is a UNIX-like standard I/O library that is layered on top of each kernel. This subroutines in the library then accomplish the task by sending one or more requests to an I/O server running on a workstation or some other processing node.

In order to incorporate a new kernel and processor two things need to be done: provide a communication link to an existing processor module, and layer the UNIX style standard I/O library on top of the new kernel.

Lastly, the conceptual similarities between this implementation and that of the X window system is worth noting. The UNIX-like standard I/O library layered on top of each kernel is like the Xlib in the case of X. It serves to hide the underlying protocol from the application code. The protocol used for I/O has its analogue to the X protocol. A key difference is that the X protocol is inherently asynchronous - a request may not generate any reply at all, or may generate it in the future. The protocol we use for file I/O is however fully synchronous. The reason for this is that the UNIX standard I/O library itself has fully synchronous semantics. In addition, it also simplifies the implementation. Finally, the I/O server is similar to the C server, except that instead of managing the screen and the mouse it manages I/O to files and terminal emulator windows that provide a terminal console to every embedded processor module.

4.5 Generation of Software Modules

At present there is no automated method to generate the code for the software modules from a high level description. The modules are manually implemented in C, or even assembly, while using the routines provided by the kernel for inter-process communication and synchronization.

This approach has the disadvantage that the implementation needs to be tweaked individually for each processor module. Further, the C implementation is not useful for simulation as it is too low-level.

A more desirable approach would be one where the C or assembly code is generated from a common high level specification language just like the description languages used for hardware module generation. This high-level specification can provide a common entry point for module generation as well as for modelling and simulation. This approach has not been investigated at all in this work. However it is worth mentioning that there exist tools that provide this capability in certain domains. For example, there exists a tool called S2C that provides the ability to generate C code from a description of a DSP algorithm in the SILAGE language, which was also mentioned in the context of hardware module generation in Chapter 3. Thus it is possible to have a library of modules described in SILAGE that can be mapped to software or hardware. The combination of Vulcan-II and Olympus synthesis systems provide a similar ability for control oriented modules described in Hardware-C.

4.6 Summary

This chapter presented processes as the preferred form of modules in software. The notion of processes provides a common thread between software and hardware parts of a system because a dedicated hardware module can also be viewed as a software process running on a processor dedicated to it. This helps in unifying the description and simulation of mixed hardware and software systems. The next two chapters will further explore this theme in the context of representation and simulation, and architecture generation.

The ideas described in this chapter have been implemented as a set of software libraries and programs that are collectively called *assys*. Using the facilities provided by *assys*, a user configures, programs, and interacts with a system designed in SIERA. Appendix D describes the organization and the use of *assys*.

CHAPTER 5

SYSTEM REPRESENTATION AND SIMULATION

The problems of what a system does (algorithm or behavior), how to describe it (representation), and how to simulate the representation are closely related. The formalism used to represent a system intrinsically restricts the behavior that can be exhibited by it. It is also very important to be able to simulate the representation in order to get an unambiguous description of the intended behavior under certain stimuli. In Chapter 1 we already know the type and complexity of systems that are of interest was defined - namely real-time reactive systems that are implemented as multiple custom boards. This chapter investigates the related problems of representation and simulation, i.e. the problem of formally specifying the functionality of these systems at a high level, and the problem of simulating the description.

Different approaches to these problems are needed for system design than used for a single ASIC design. The techniques used at the chip level lead to an explosion of details when used to represent and simulate complex systems, and the computation models used by these chip level formalisms do not reflect the heterogenous and distributed nature of the architecture at the system level.

5.1 High-Level System Modelling

A model of a system is used to specify and express its behavior by abstracting its salient properties. Models can be used for design, simulation, analysis, verification etc. Even within the domain of systems doing digital computation there is a plethora of diverse models that have been used by researchers in fields such as CAD of ICs, CASE, DSP etc. The diversity of these models stems primarily from one or more of the following factors:

- a. *Application Domain*
e.g., general-purpose software systems, ASICs, DSP systems etc.
- b. *Goal of the Model*
e.g., rapid-prototyping, verification and validation, analysis, simulation etc.
- c. *Underlying Model of Computation*
e.g., sequential imperative, functional multiprogramming, communicating processes formalisms etc.

Associated with a model is also some form of syntactic representation, usually a textual or graphical language, using which systems are modelled.

There is no one single model that is suited for all applications. For example, digital signal processing applications are usually best modelled as signal-flow graphs, whereas control dominated applications, such as communication protocol processors, are usually best described as communicating finite state machines. Consequently, the choice of a particular system model, inevitably restricts the type of systems that can be expressed in it in a natural fashion.

Most systems belonging to the class of systems that is of interest to us, namely, dedicated real-time systems that continually interact with the environment, are naturally expressed at the top level as a set of processes that operate concurrently and communicate with each other and the environment. For example, the architecture of many major robot systems is one of communicating sequential processes. This was also the case with the first-generation implementation of our driver robot controller example where the system software turned out to be a specialized, distributed, real-time, message-passing operating system [Arya89]. Further, the underlying hardware in such systems is

inherently concurrent. It has multiple general-purpose software-programmable processors together with dedicated hardware devices that are used as computation accelerators or as I/O peripherals. These hardware devices operate in parallel with the software-programmable processors, and communicate and synchronize with them. The representation of these hardware devices at the software level is also facilitated by treating them as separate processes running on their own dedicated processors.

The utility of describing a system as a set of co-operating processes has been realized by many researchers over time, and has resulted in the proposal of many models for structured specification and implementation of such systems. These models can all be loosely classified as viewing a system like a graph, where nodes represent concurrent computing agents or *processes*, and edges represent paths along which the processes communicate. These models mostly differ in the inter-process communication (IPC) mechanism used, the allowable behavior of the individual processes, and whether the set of processes is static or dynamic.

The underlying model used by VHDL also belongs to the above category. In VHDL a system is viewed as a static set of sequential processes that communicate and synchronize via *signals* that have very well defined semantics. Unfortunately, as pointed out by others previously [Hubbard90], the VHDL signal is not a suitable IPC primitive for high-level specification and modelling of systems. Message-passing channels, events, and shared-data structures are the more commonly used IPC and synchronization primitives. The reason the VHDL signal is not suitable for high-level modelling is two-fold. First, VHDL allows only static resolution in case of multiple drivers whereas a last-driven resolution is very useful for high level modelling. Second, modelling of asynchronous communication using a VHDL signal is difficult. For example, although VHDL signals can have memory (by using guarded signal of the kind *register*), they do not easily provide queue buffer type memory that is needed for high-level modelling of asynchronous communication. The complexity of doing high-level modelling in VHDL is also indicated by the techniques presented in other recent publications [Hubbard90][Aylor91][Narayan91]. For

example, Aylor *et. al.* [Aylor91] describe an elaborate VHDL technique for high-level system modelling based on extended Petri nets. The token passing mechanism is implemented by a handshake protocol built on top of VHDL signals.

5.1.1 Alternative Models of Inter-Process Communication

As mentioned earlier, many models for system specification have been proposed over the years that view the system as a set of concurrent processes that communicate with each other. A classification of the various such models can be done along the following dimensions:

- a. Process Set:
 - is it *static* or *dynamic*?
 - is it *bounded* or *unbounded*?
- b. Behavior of Processes:
 - do the processes have *state*?
 - can the processes exhibit *indeterminate* behavior? (*e.g.*, can it wait for 1 of 2 inputs, whichever arrives first?)
- c. IPC mechanism:
 - is it *shared memory* or *message passing* based?
 - is the message passing *synchronous* (unbuffered) or *asynchronous* (buffered)?
 - is the buffer depth *bounded* or *unbounded*?
 - is the information flow *unidirectional* or *bidirectional*?
 - if *bidirectional*, is it *simultaneous* or *delayed*?
 - is the communication control *symmetric* or *asymmetric*?
 - is the message channel *FIFO* ordered?
 - can the channel have multiple *readers* or *writers*?

•Using the above dimensions as a guideline, some of the popular system specification models are described below. While some of the following are languages with independently defined semantics, the rest are defined indirectly by an associated simulator or synthesis tool.

- **CSP** [Hoare85]

It models the system as a static set of possibly non-deterministic sequential processes which communicate with each other via unidirectional unbuffered (synchronous) communication channels with single reader and single writer.

- **Functional Multiprogramming** [Kahn74]

It models the system as a static network of deterministic sequential processes which communicate with each other via unidirectional FIFOs with infinite buffering and single writer and multiple readers. The network can have unbounded recursive parallelism. The processes cannot wait for input on more than one channel simultaneously. Mathematically, such a sequential process is equivalent to a continuous function from the sequences on input channels to sequences on its output channels. This model is a progenitor of the various dataflow models. It can model only *determinate* computation where the result does not depend on the order of computation.

- *Static Dataflow* [Dennis74]]

This is a subset of Functional Programming where the set of processes is bounded, the FIFO channels are restricted to single readers, and are non-buffered. Asynchronous hardware techniques, such as the one used in [Jacobs91], also follow the static dataflow model.

- *Dynamic Dataflow*

This too is a subset of Functional Programming similar to static dataflow except that unlike static dataflow it allows the FIFO channels to be buffered. CAPSIM, a popular block diagram simulator for DSP systems [Messerschmitt84] uses dynamic dataflow as its computation model.

- *Synchronous Dataflow* [Lee87]

This is a subset of Functional Programming where a process repeatedly consumes a statically known number of data items from each input channel, and produces a statically known number of data items on each output channel. This is a generalization of the notion of *well-behaved* dataflow graphs. Modelling data dependent computation is inefficient in this model. However, a big advantage of this model is that a cyclic schedule for the execution of processes can be statically determined.

- *Hardware-C* [Ku90]

Proposed by researchers at Stanford for behavioral specification of clock synchronous ASICs for synthesis, it allows a static set of processes that communicate through unbuffered channels like in CSP, as well as through a shared medium.

- *CSIM* [Schwetman86]

It is a process-oriented discrete event simulator that models a system as a dynamic set of pro-

cesses that communicate through events and message queues that can have multiple readers and writers. The events are like buffered channels of depth one where the writer overwrites if the buffer is full. Message queues are infinitely buffered FIFO channels.

- *SILAGE* [EDC90]

It is a popular language for specification of DSP systems. The metaphor used by it is that of *signal-flow graphs* used in signal processing. It is distinctive principally for syntactic reasons - it is otherwise isomorphic to static dataflow.

5.1.2 Model Used in this work

Our choice for a high-level model for application-specific systems was driven by the following requirements. First, it should be able to naturally express the kind of application-specific system we are most interested in, namely embedded systems that continually interact with their environment in real-time. Second, it should be suitable for rapid-prototyping of the application-specific system as a mix of dedicated and general-purpose hardware and software. Third, the model should be easy to simulate. Last, its behavior should be easily analyzable so that the specification may be manipulated and transformed by CAD tools for synthesis. Since our initial goal is to create a vertically integrated framework for rapid-prototyping, most attention was given to the first three requirements. The model itself should encourage modular composition of the system, reusability of system level modules, unified treatment of hardware and software modules, and uniform communication between the modules.

The model adopted for our rapid-prototyping framework views an application-specific system as a static, hierarchical, network of processes that execute concurrently, and interact using a well-defined communication mechanism based on FIFO channels. Each process has several input and output ports, and a channel connects an output port to an input port. A process can send data samples at an output port, receive samples at an input port, or wait non-deterministically for one of a set of ports to become ready for communication. The processes make these port operations in a sequential fashion, but otherwise may have finer granularity parallelism. In fact the model does

not specify any particular language to be used for describing the functionality of the processes themselves. Since no unified language exists for describing all the different types of processes, nor is such a language likely to be available in the near future, allowing a multitude of languages to be used for this purpose is a reasonable choice. Any reasonable language, or set of languages, with appropriate extensions to access the ports may be used for this purpose - this may include imperative languages like C, VHDL, and Hardware-C, or applicative languages like SILAGE.

Channels are characterized by a buffer depth and a data type. A buffer depth of zero indicates synchronous or unbuffered communication. A buffer depth larger than zero indicates asynchronous communication. Infinite buffer depth is allowed for simulation purposes.

The ports are characterized by a data type and a port protocol. The port protocols specify the behavior when a channel is full or empty. For an output port it can be *block on full*, *overwrite on full*, and *ignore on full*. For an input port it can be *block on empty*, *previous on empty*, and *ignore on empty*. The port protocols are fixed, and cannot change dynamically. The *overwrite on full* and *previous on empty* protocols have been included to handle many low-level hardware interfaces.

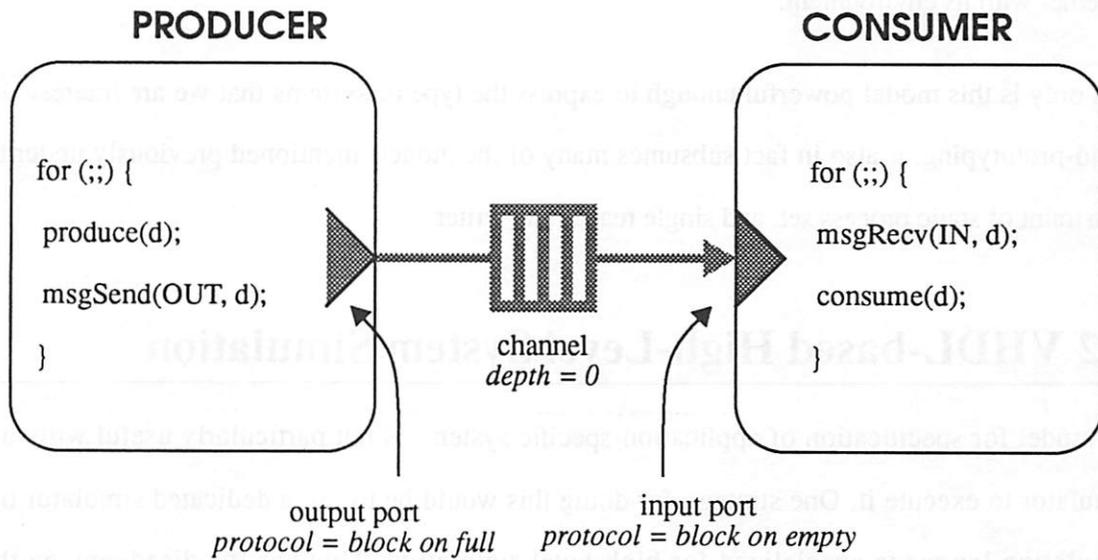
Such a process network can be viewed as a parameterized block diagram which is a very natural paradigm for thinking about the class of systems of interest to us. There are two facets to such a block diagram or process network: the description of the blocks themselves, and the description of block interconnection. This can be viewed as a netlist with the blocks coming from a library. Each block may have multiple implementations, each with the same port interface. This is similar to the approach used in GABRIEL [Lee89] and Ptolemy [Ptolemy91], except that underlying computation semantics are quite different. In GABRIEL the computation model is that of synchronous dataflow, where, as described in the previous sub-section, each block repeatedly consumes a statically known number of data items on each input, and produces a statically known number of data items on each output. Such a system is capable of doing only determinate computation. For example, it is not possible to describe a system that waits for data items on

multiple asynchronous inputs and services them in the order they arrive. Such non-determinism or asynchrony is easily expressed in the process network model. Unlike GABRIEL, Ptolemy is designed to support multiple computation models so that, in theory, it can subsume the process network model. However, at present, no such computation model is supported in Ptolemy.

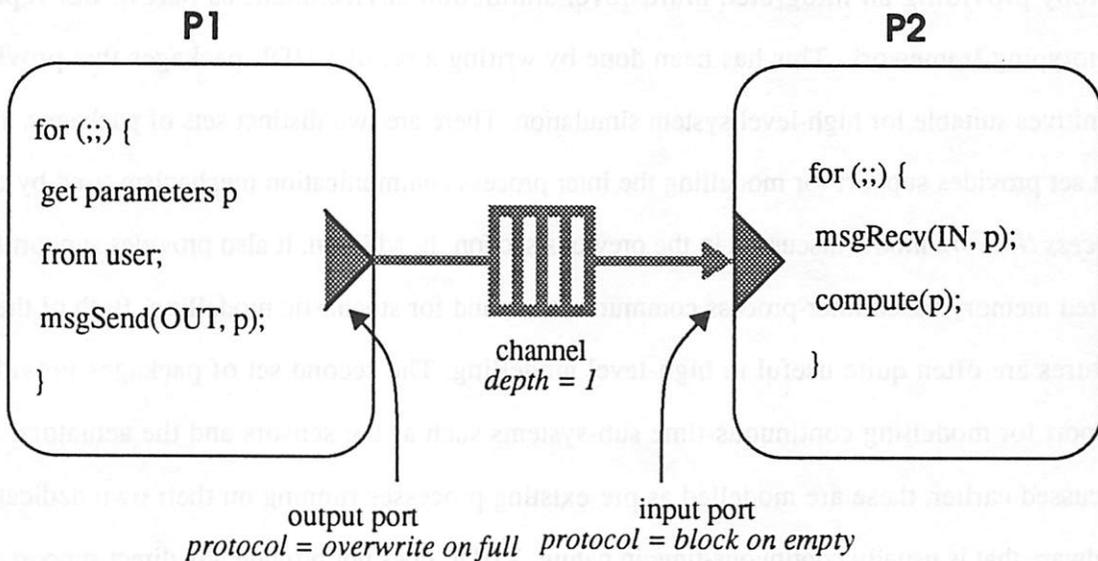
Figure 5-1 shows two simple example systems described in the process network formalism, with the process behavior described in pseudo-C. The first example demonstrates CSP-like synchronous (unbuffered or handshaked) communication between a producer process and a consumer process. The blocking protocol together with a channel buffer depth of zero result in the two processes rendezvousing in time for data transfer to occur. The second example illustrates the use of *overwrite on full* and *previous on empty* port protocols to allow *safe* updating of parameters used in P2 according to user input received by P1. For example, P2 may be a process controlling a motor using PID control, and the parameter therefore is a structure with three coefficients for the P, I, and D parts of the controller. The process network shown ensures that the entire parameter structure is updated atomically and asynchronously. The computation in P2 is never delayed, and uses the latest available set of parameters.

An attractive feature of the process network model is that it does not make any distinction between hardware and software. As mentioned in Chapter 4, the notion of processes is equally applicable to software processes running on a programmable processor, and to dedicated hardware modules that can be viewed as processes running on their own processors. Therefore the process network model can not only be used to model abstract behavior, but also the architecture of the system in terms of its hardware and software organization.

Another advantage of this model is that the sensors, actuators and mechanical subsystems using which the system interacts with the physical world can be viewed as pre-existing processes executing on their own dedicated electrical or mechanical hardware in parallel with the rest of the system. This *Process Network* model thus provides an abstract representation for the entire system



(i) Process Network for a Producer-Consumer System with CSP-like Synchronous (Unbuffered) Communication



(ii) Process Network for a System with a User-Interface Process P1 Asynchronously and Atomically Updating Computation Parameters for a Process P2.

Figure 5-1 : Simple Example Systems Described as Process Networks

together with its environment.

Not only is this model powerful enough to express the type of systems that we are interested in rapid-prototyping, it also in fact subsumes many of the models mentioned previously under the constraint of static process set, and single reader and writer.

5.2 VHDL-based High-Level System Simulation

A model for specification of application-specific systems is not particularly useful without a simulator to execute it. One strategy for doing this would be to use a dedicated simulator or a simulation language specialized for high-level simulation. This has the disadvantage that selectively modelling parts of the system at lower levels of detail is ruled out *a priori*. We instead decided to simulate our model of high-level system specification on top of a VHDL substrate, thereby providing an integrated multi-level simulation environment as part of our rapid-prototyping framework. This has been done by writing a set of VHDL packages that provide primitives suitable for high-level system simulation. There are two distinct sets of packages. The first set provides support for modelling the inter-process communication mechanism used by our *Process Network* model discussed in the previous section. In addition, it also provides support for shared memory based inter-process communication, and for stochastic modelling. Both of these features are often quite useful in high-level modelling. The second set of packages provides support for modelling continuous-time sub-systems such as the sensors and the actuators. As discussed earlier, these are modelled as pre-existing processes running on their own dedicated hardware that is usually continuous-time in nature. VHDL does not provide any direct support for describing the behavior of such sub-systems, and a set of VHDL packages were written to simplify this task.

In the following discussion we often refer to *template packages*. This refers to a preprocessing based strategy that we have used to bypass the limitations imposed by the lack of *generic packages*

(packages parameterized by a data type) and *function pointer* type. Actual instances of these template packages are generated using a simple preprocessor that customizes the package for a particular user-defined data type or function using simple text substitution. We have so far found this to be the most clean way to overcome these problems.

5.2.1 VHDL Package for Simulating the Process Network Model

A VHDL package template called *MSGPACK_type* has been written that provides primitives to support modelling a system according to the Process Network model presented earlier. It allows VHDL processes to communicate using special signals that implement the FIFO channel based communication mechanism. The package template is parameterized by the data type *type* carried by the channel, and an instance of the package needs to be generated for each channel data type.

The channels are implemented as a separate VHDL entity that connects with ports in the source and the destination processes using signals of a special data type called *MSG_type*. One such channel entity needs to be used for every channel in the process network. In our framework this is done automatically on the process network which is entered by the user either schematically or by using a special parameterized structure description language. The channel entity is parameterized by a single VHDL generic for specifying the buffer depth. The channel entity has a third port where it outputs data about the channel state. This is used for collecting statistics about the data traffic through the channel.

The VHDL processes that model the computation nodes in the process network model access the channel through *msgSend()* and *msgRecv()* operations on the corresponding VHDL ports of type *MSG_type*. These operations can be done in one of several modes corresponding to the port protocol as discussed in section 5.1.2. For writing to an output port, the permissible modes are *block on full* (BMODE), *overwrite on full* (OMODE), *ignore on full* (IMODE), and *error on full* (EMODE). For reading from an input port, the permissible modes are *block on empty* (BMODE),

previous on empty (PMODE), *ignore on empty* (IMODE), and *error on empty* (EMODE). The *error on full/empty* mode is provided only for simulation purposes. Two other important procedures are *waitAny()* and *waitAll()*. They take a list of input or output ports, and return only when any or all of them are ready for communication. In addition, several other utility routines are provided that enable monitoring of the channel state from these processes. These utility routines are meant strictly for simulation purposes.

The key issue in the implementation of this package was the implementation of the *channel*, as defined by the process network model, using VHDL *signals*. As already indicated earlier, a separate VHDL entity is used to model the buffering in the channel. This buffer is connected to user processes using VHDL signals of type *MSG_type*. This data type is really a composite type, and can be viewed as a bundle of primitive VHDL signals. Besides the signal to carry the data sample, there are signals to implement a 4-phase handshaking between the buffer process and the sender or receiver process. In addition, there are signals to transmit channel information data to the sender or the receiver. The ports of type *MSG_type* must be of mode *inout*. The reason for this is that some of the control signals that implement the handshake move in a direction opposite to that of the data. Every signal of type *MSG_type* therefore has two ports of mode *inout* connected to it - one from the buffer side, the other from the side of the sender or receiver process. A complex resolution function is required to implement the handshaking. The complexity comes because there of the need to disambiguate inside the resolution function as to which driver is connected to the buffer, and which to the sender or the receiver. This connection may be through a series of port-signal associations, which complicates things even further. We use an algorithm based on maintaining information about how far a signal is from the ports in the buffer process and the sender or receiver processes. This information is used to identify the drivers inside the resolution function. Figure 5-2 shows a simplified picture of the primitive signals that constitute a signal of type *MSG_type*, and lists the various functions available to the sender and the receiver processes.

Although this package is meant to simulate the process network model used in our rapid-

prototyping framework, using an appropriate combination of channel type, write mode, and read mode, it can also be used to simulate many other similar system models, some of which were discussed in section 5.1.1. This is of course under the constraint of a static set of processes, and channels with single reader and writer. Table 5-1 shows some examples of this.

Channel Type	Write Mode	Read Mode	Comments
Unbuffered	Block on Full	Block on Empty	Same as CSP [Hoare85]
Unbuffered-FIFO	Any Mode	Block on Empty	Similar to Kahn's Model [Kahn74]
Unbuffered	Overwrite on Full	Previous on Empty	Like a Wire or VHDL <i>Signal</i>
FIFO of depth 1	Overwrite on Full	Block on Empty	Like CSIM [Schwetman86] <i>event</i>
Unbuffered	Any Mode	Block on Empty	Like CSIM <i>message</i>

Table 5-1 : Examples of Emulating Various IPC Models with the *MSG* Package

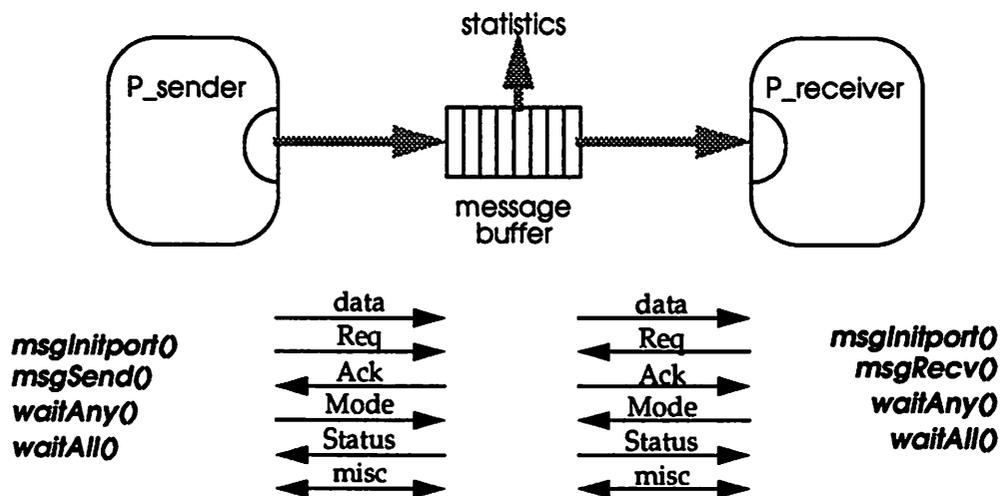


Figure 5-2 : Vhdl Implementations of Channels for the Process Network Model

Support for Shared Memory based IPC

In many modelling situations using shared memory as a form of IPC is a quick and dirty, but efficient, solution. Although shared memory can be expressed in the framework described above, it is not very efficient. Therefore, a special VHDL package template, called `SHMPACK_type`, has been written that allows modelling of shared data objects. The shared memory is implemented as a special VHDL signal of type `SHM_type`, where *type* represents the data type of the shared memory. A preprocessor is used to generate an instance of the package for a specific data type. Readers and writers of a shared memory can access it either as a global signal, or using signals connecting ports. The shared memory behaves like a true multi-port memory with one object of the specified data type. A VHDL process uses procedures `shmRead()` and `shmWrite()` to do atomic read and write on the shared data object. These atomic reads and writes take zero physical time, and simultaneous reads or writes by more than one process result in them being arbitrarily interleaved.

Following is the VHDL code for the header of the package. Basically a VHDL signal of type `SHM_type` has a time-stamp attached to it in order to achieve a *last-driven value* resolution. In other words such a signal always has the latest value driven on it, thus effectively giving an *atomic register* [Lamport] containing a datum of type *type*. A true multi-ported memory containing multiple atomic registers is easily constructed by using an array of VHDL signals of type `SHM_type`.

```
--
-- File: shm_<type>.vhd
--
-- This is a package to implement shared memory (pools) of type <type>
-- for my process network. Multiple readers and writers are allowed
-- with atomic read and write capability to the entire <type> data
-- structure. In case of simultaneous writes (even if to
-- different fields of the shared memory), only one (undefined)
-- will take effect. No built-in mutual-exclusion capability is
-- provided for critical sections - a separate process needs
-- to be created with the desired arbitration scheme. This
-- decision was taken to allow arbitrary schemes.
--
```

```

package SHM_<type> is
  subtype SHM_DATA is <type>;
  type SHM_DATA_VECTOR is array(NATURAL range <>) of SHM_DATA;
  type SHM_PORT_<type>_1 is record
    D : SHM_DATA;
    T : TIME;
  end record;
  type SHM_PORT_<type>_1_ARRAY is array (NATURAL range <>) of SHM_
PORT_<type>_1;
  function srf(v: SHM_PORT_<type>_1_ARRAY) return SHM_PORT_<type>_1;
  subtype SHM_PORT_<type> is srf SHM_PORT_<type>_1;
  procedure shmWrite(d: in <type>; signal p: out SHM_PORT_<type>);
  procedure shmRead(d: out <type>; signal p: in SHM_PORT_<type>);
end SHM_<type>;

package body SHM_<type> is
  -- resolution function that uses the time-stamp field T to
  -- achieve last-driven resolution
  function srf(v: SHM_PORT_<type>_1_ARRAY) return SHM_PORT_<type>_1 is
    variable result : SHM_PORT_<type>_1;
    variable newest_i : NATURAL;
  begin
    for i in v'RANGE loop
      if v(i).t >= v(newest_i).t then
        newest_i := i;
      end if;
    end loop;
    return v(newest_i);
  end srf;

  -- procedure to write to the memory
  procedure shmWrite(d: in <type>; signal p: out SHM_PORT_<type>) is
  begin
    p.d <= d;
    p.t <= NOW;
  end shmWrite;

  -- procedure to read the memory
  procedure shmRead(d: out <type>; signal p: in SHM_PORT_<type>) is
  begin
    d := p.d;
  end shmRead;
end SHM_<type>;

```

Shared memory inevitably leads to the problem of resource contention, and a low-level solution to this problem is using *binary semaphores*. A VHDL package called SEMPACK has been written that implements a restricted form of *binary semaphores*. The restriction is that the semaphore must be *given* by the same process as has *taken* it. Such a semaphore can only be used for mutual exclusion purposes, and not for signalling. When used together with the SHM_type package, it can

implement shared protected data structures. Two types of semaphores are provided: `SEMAPHORE_PRIO` and `SEMAPHORE_FCFS`. These are VHDL data types, and a semaphore object is a VHDL signal of one of these types visible to all interested VHDL processes. `SEMAPHORE_PRIO` implements a priority scheme for waking-up waiting processes. `SEMAPHORE_FCFS` uses a first-come-first-served scheme for waking-up processes. The processes operate on semaphore objects using procedures `semTake()` and `semGive()` which implement the acquisition and release of a semaphore by a process. The `semTake()` procedure also allows modelling a time-out capability.

The VHDL code implementing the `SEMPACK` package is presented in the following paragraph. The package header declares VHDL data types corresponding to the two types of semaphores, `SEMAPHORE_PRIO` and `SEMAPHORE_FCFS`. Both these data types are record data types with fields to store the value of the semaphore, the priority of the process accessing the semaphore, and, in the case of `SEMAPHORE_FCFS`, a field to store a time-stamp. The resolution functions `srfl` and `srf2` implement the priority-based and first-come-first-served policies respectively for waking up the processes waiting to take a semaphore. This is accomplished by selecting the process with the highest priority (in the case of `srfl`) or with the oldest time-stamp (in the case of `srf2`). Finally a set of over-loaded functions (multiple functions with the same name) to *give* and *take* the semaphores are defined. Two versions of the function `semGive` are defined corresponding to the two types of semaphores. Four versions of the function `semTake` are defined - two for each type of semaphore. For each semaphore type, one version of `semTake` is a “blocking” version where the process waits until the semaphore becomes available. The second version takes an additional parameter specifying a time-out period corresponding to the maximum time for which the process waits while trying to acquire a semaphore - the success in acquiring the semaphore is indicated by the boolean flag `f` returned by this “non-blocking” version of `semTake`.

```
--  
-- File: sem.vhd  
--
```

```

-- binary semaphore package
--
-- semaphores are useful primitives for concurrent programming
--
-- The intent in this package is that semaphore objects are
-- statically declared as signals visible to all the interested
-- processes. Various useful operations are defined on the
-- semaphore object to allow synchronization of concurrent
-- processes. Together with the SHM package, one can implement
-- inter-process communication, although using the MSG package
-- is cleaner. The main use foreseen for the SEM package is
-- for critical section (resource contention) problems.
--
-- The semaphores are binary semaphores. When a process takes
-- a semaphore that is empty, that process will suspend until
-- the semaphore is full again. Any number of processes may be
-- waiting on the same semaphore. At most one process may acquire
-- the semaphore at one time. When the semaphore becomes full,
-- the highest priority process is given the semaphore.
--
-- Two types of semaphores are defined:
--
-- SEMAPHORE_PRIO:
-- the highest priority process is run
-- SEMAPHORE_FCFS:
-- the process with longest wait time is run
-- priority is used to break ties.
--
-- Caution: only a process that has the semaphore should give it back.
-- A semaphore is initially full. Unlike SHM and MSG objects,
-- SEM cannot be ports. The reason is that if they are made
-- ports then there is actually a hierarchy of interconnected
-- SEMAPHORE objects and the arbitration algorithm fails in such
-- a distributed case.
--

package sem is
  subtype PRIORITY is NATURAL;
  type SEMVALUE is range 1 downto 0;
  type SEMAPHORE_PRIO_1 is record
    V: SEMVALUE;
    P: PRIORITY;
  end record;
  type SEMAPHORE_FCFS_1 is record
    V: SEMVALUE;
    P: PRIORITY;
    T: TIME;
  end record;
  type SEMAPHORE_PRIO_1_ARRAY is array(NATURAL range <>)
    of SEMAPHORE_PRIO_1;
  type SEMAPHORE_FCFS_1_ARRAY is array(NATURAL range <>)
    of SEMAPHORE_FCFS_1;
  function srf1(v: SEMAPHORE_PRIO_1_ARRAY) return SEMAPHORE_PRIO_1;
  function srf2(v: SEMAPHORE_FCFS_1_ARRAY) return SEMAPHORE_FCFS_1;
  subtype SEMAPHORE_PRIO is srf1 SEMAPHORE_PRIO_1;
  subtype SEMAPHORE_FCFS is srf2 SEMAPHORE_FCFS_1;

```

```

-- take a semaphore
procedure semTake(signal s: inout SEMAPHORE_PRIO; p: in PRIORITY);
-- take a semaphore if available within a timeout period
procedure semTake(signal s: inout SEMAPHORE_PRIO; p: in PRIORITY;
  t: TIME; f: out BOOLEAN);
-- give a semaphore
procedure semGive(signal s: inout SEMAPHORE_PRIO);

-- take a semaphore
procedure semTake(signal s: inout SEMAPHORE_FCFS; p: in PRIORITY);
-- take a semaphore if available within a timeout period
procedure semTake(signal s: inout SEMAPHORE_FCFS; p: in PRIORITY;
  t: TIME; f: out BOOLEAN);
-- give a semaphore
procedure semGive(signal s: inout SEMAPHORE_FCFS);
end sem;

package body sem is
function srf1(v: SEMAPHORE_PRIO_1_ARRAY) return SEMAPHORE_PRIO_1 is
  variable f : BOOLEAN := false;
  variable i_sel : NATURAL;
begin
  -- resolution function to implement a priority policy
  --
  -- find the maximum priority process
  for i in v'RANGE loop
    next when v(i).p=0;
    if f=false then
      i_sel := i;
      f := true;
    else
      next when v(i).p<v(i_sel).p;
      assert v(i).p/=v(i_sel).p
        report "SEM error: simultaneous equal priority accesses!"
          severity FAILURE;
      if v(i).p>v(i_sel).p then
        i_sel := i;
      end if;
    end if;
  end loop;
  if f=false then
    return SEMAPHORE_PRIO'(1,0);
  else
    return SEMAPHORE_PRIO'(0,v(i_sel).p);
  end if;
end srf1;

function srf2(v: SEMAPHORE_FCFS_1_ARRAY) return SEMAPHORE_FCFS_1 is
  variable f : BOOLEAN := false;
  variable i_sel : NATURAL;
begin
  -- resolution function to implement a first-come-first-serve
  -- policy
  --
  -- find the oldest request with non-zero priority

```

```

-- break ties according to priority
for i in v'RANGE loop
  next when v(i).p=0;
  if f=false then
    i_sel := i;
    f := true;
  else
    assert not(v(i).p=v(i_sel).p and v(i).t=v(i_sel).t)
      -- cannot resolve a tie
    report "SEM error: simultaneous equal priority accesses!"
      severity FAILURE;
    if (v(i).t<v(i_sel).t) or (v(i).t=v(i_sel).t and
      v(i).p>v(i_sel).p) then
      i_sel := i;
    end if;
  end if;
end loop;
if f=false then
  return SEMAPHORE_FCFS'(1,0,NOW);
else
  return SEMAPHORE_FCFS'(0,v(i_sel).p,v(i_sel).t);
end if;
end srf2;

```

```

procedure semTake(signal s: inout SEMAPHORE_PRIO; p: in PRIORITY) is
begin

```

```

  if (s.p=p) then
    -- semaphore is already taken by the current process
    assert false
    report "semTake: bad priority or repeated takes"
      severity FAILURE;
  elsif (p=0) then
    assert false
    report "semTake: bad priority 0"
      severity FAILURE;
  else
    while true loop
      if (s.p/=0) then wait until s.p=0; end if;
      s.p <= p;
      wait until s.p/=0; -- this will take 0 time
      exit when s.p=p;
      s.p <= 0;
    end loop;
  end if;
end semTake;

```

```

procedure semTake(signal s: inout SEMAPHORE_PRIO; p: in PRIORITY;
t: TIME; f: out BOOLEAN) is
  variable timeout : TIME;
begin

```

```

  if (s.p=p) then
    -- semaphore is already taken by the current process
    assert false
    report "semTake: bad priority or repeated takes"
      severity FAILURE;
  elsif (p=0) then

```

```
    assert false
      report "semTake: bad priority 0"
      severity FAILURE;
  else
    timeout := NOW + t;
    while true loop
      if (s.p/=0) then wait until s.p=0; end if;
      wait until s.p=0 or NOW=timeout;
      exit when s.p/=0;
      s.p <= p;
      wait until s.p/=0; -- this will take 0 time
      exit when s.p=p or NOW=timeout;
      s.p <= 0;
    end loop;
  end if;
  f := (s.p=p);
end semTake;

procedure semGive(signal s: inout SEMAPHORE_PRIO) is
  variable p : PRIORITY;
begin
  if (s.v=1) then
    -- semaphore is already full
    assert false
      report "semGive: semaphore is already full"
      severity FAILURE;
  elsif (s.p=0) then
    -- semaphore priority is bad
    assert false
      report "semGive: semaphore already has 0 priority"
      severity FAILURE;
  else
    s.p <= 0;
    wait until s.p=0; -- this will take 0 time
  end if;
end semGive;

procedure semTake(signal s: inout SEMAPHORE_FCFs; p: in PRIORITY) is
begin
  if (s.p=p) then
    -- semaphore is already taken by the current process
    assert false
      report "semTake: bad priority or repeated takes"
      severity FAILURE;
  elsif (p=0) then
    assert false
      report "semTake: bad priority 0"
      severity FAILURE;
  else
    s.p <= p;
    s.t <= NOW;
    wait until s.p=p and s.v=0;
  end if;
end semTake;

procedure semTake(signal s: inout SEMAPHORE_FCFs; p: in PRIORITY;
```

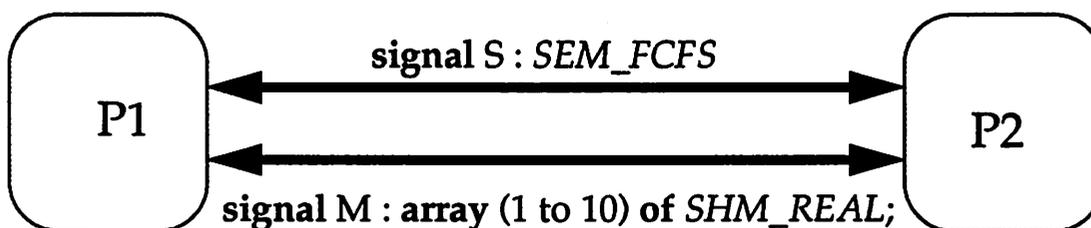
```

    t: TIME; f: out BOOLEAN) is
begin
    if (s.p=p) then
        -- semaphore is already taken by the current process
        assert false
            report "semTake: bad priority or repeated takes"
            severity FAILURE;
    elsif (p=0) then
        assert false
            report "semTake: bad priority 0"
            severity FAILURE;
    else
        s.p <= p;
        s.t <= NOW;
        wait until s.p=p and s.v=0 for t;
        if (s.p=p and s.v=0) then
            f := true;
        else
            f := false;
        end if;
    end if;
end semTake;

procedure semGive(signal s: inout SEMAPHORE_FCFS) is
    variable p : PRIORITY;
begin
    if (s.v=1) then
        -- semaphore is already full
        assert false
            report "semGive: semaphore is already full"
            severity FAILURE;
    elsif (s.p=0) then
        -- semaphore priority is bad
        assert false
            report "semGive: semaphore already has 0 priority"
            severity FAILURE;
    else
        p := s.p;
        s.p <= 0;
        s.t <= NOW;
        wait until s.p/=p;
    end if;
end semGive;
end sem;

```

The SHM_type and SEMPACK packages together let the user express most shared memory situations. Figure 5-3 shows a simple example of using the two packages to model two processes accessing a shared resource.



P1: process begin

```

. . .
semTake(S);
-- use the shared resource
shmWrite(data,M(addr));
semGive(S);
. . .

```

end process;

P2: process begin

```

. . .
semTake(S);
-- use the shared resource
shmWrite(data,M(addr));
semGive(S);
. . .

```

end process;

Figure 5-3 : A Simple Example Demonstrating the Use of the *SHM* and *SEM* VHDL Packages

Support for Stochastic Models

Modelling stochastic behavior is often very important in high-level modelling. VHDL does not provide a built-in set of random number generators. The lack of a good VHDL math library made doing this entirely in VHDL not a very attractive option. Instead, the C interface capability of the MCC VHDL simulator [MCC91] was used to provide a package with a set of random number generators for a variety of commonly used distributions, such as normal, poisson, uniform, erlang, hyperexponential etc. The functions provided by the package have been modelled after those provided in CSIM [Schwetman86]. Following is the VHDL code for the header of the package. The body of the package as currently implemented uses non-standard constructs provided by the MCC VHDL simulator for accessing foreign functions in C, and is therefore not of general validity and is not presented here.

```

package random is
  --erlang - return random from Erlang(u,v)

```

```

function erlang(u,v : in REAL) return REAL;
--expntl - return random from Exp(x)
function expntl(x : in REAL) return REAL;
--hyperx(u,v) - hyperexponential with mean u, variance v
function hyperx(u,v : in REAL) return REAL;
--normal(u,s) - normal with mean u, standard dev. s
function normal(u,s : in REAL) return REAL;
--prob - return random from Uniform[0,1)
function prob return REAL;
--random - return integer random from Equiprob(i1,i2)
function random(i1,i2 : INTEGER) return INTEGER;
--uniform - return random from Uniform[ x1, x2 )
function uniform(x1,x2 : in REAL) return REAL
--srandom - set the seed of the random number generator
function srandom(s : INTEGER) return INTEGER;
end random;

```

5.2.2 Modelling Continuous-Time Subsystems using VHDL

As already mentioned sensors, actuators, electromechanical components, and analog electronic subsystems are integral parts of many application-specific systems. These are all continuous-time dynamical systems. In most cases the environment with which the system interacts is also continuous-time in nature. For example, in order to do a meaningful high-level simulation of the robot controller, one needs to simulate its operation together with the servo-amplifiers, the robot arm and the physical world. Further, the sensory subsystems of the controller itself are internally require modelling of continuous time parts. In short, the ability to model continuous-time subsystems together with rest of the application-specific system is extremely important for realistic high-level simulation. Unfortunately VHDL does not provide any built-in support for such modelling.

There has been some recent work done in doing SPICE-like low-level simulation of analog circuits within VHDL [Zhou91]. This is however not very useful for high-level system modelling. Our requirements are for a high-level behavioral modelling of commonly encountered continuous-time sub-systems. A set of coupled, non-linear, time-varying ordinary differential equations (ODEs) can easily express such behavior. In many cases, such as the robot arm mechanics and the electric motors driving them, it is also the most natural representation. In other cases, such as for analog electronic subsystems, it is sufficiently user friendly for expressing the overall high-level

functionality.

There are two possible approaches to solving this problem. One would be to link a simulator, such as SIMNON [Åström82], that is capable of simulating dynamical systems expressed as ODEs to a VHDL simulator. Multi-simulator integration environments such as Ptolemy [Ptolemy91] together with foreign-simulator interface provided by VHDL simulators, such as MCC VHDL Simulator [MCC91], may make this an attractive solution in the future. The approach taken in our environment is a more traditional one. Since VHDL is also a programming language, one can just write dynamical system simulators within VHDL for the continuous-time parts of the overall system. In other words, a simulator for continuous-time dynamical systems described using ODEs is embedded in the discrete-event VHDL simulator.

Algorithm for Continuous-Time Simulation in VHDL

The conventional method for simulation of continuous-time systems is based on treating the various interconnected continuous-time blocks together as one system described by a single set of coupled ODEs. This set of ODEs is then solved using numerical integration techniques with adaptive time-steps. Unfortunately this method is not feasible inside VHDL because of two problems. First, there is no easy and portable way to collapse the various interconnected continuous-time blocks into a single set of ODEs. This is a result of the strict modularity imposed by VHDL as well as the lack of any control over the simulator itself. Second, using adaptive time steps may require the ability to roll back the time if the error is too large. This too is not possible from inside a VHDL model.

In light of the above constraints, the only practical alternative is to use an algorithm that will solve the continuous-time dynamical system in a distributed fashion. The basic strategy is that corresponding to each continuous-time subsystem there is a VHDL process that implements a dedicated simulator for simulating the dynamics of just that subsystem, and a special synchronization mechanism is used so that the network of these VHDL processes effectively

solves the continuous-time dynamical system.

The dynamics of each continuous-time subsystem is represented in the *state-space* formalism:

$$\frac{d\bar{s}}{dt} = f(\bar{s}, \bar{u}, t) \quad (\text{EQ 1})$$

$$\bar{v} = g(\bar{s}, \bar{u}, t) \quad (\text{EQ 2})$$

where \bar{s} is the state vector, \bar{u} is the input vector, and \bar{v} is the output vector.

Each such subsystem is modelled by a VHDL process that solves the above equations numerically. This is done using two distinct steps that are performed repeatedly. Using (EQ 1), standard numerical techniques, the state vector at time ΔT into the future is calculated. A series of small, adaptive, time steps may be taken to accomplish this. ΔT itself depends on the dynamics of the model as well as the dynamics of the inputs. After advancing the time, the output vectors are recalculated using the new state vector, and the current input vector. This step is not as simple as it appears to be. The problem is that the different continuous-time blocks update their outputs asynchronously. These outputs may form inputs to other such blocks in the coupled system. It is essential to make sure that all these signals get settled at a given time step before the time is advanced. We use the following algorithm to accomplish this: each subsystem re-evaluates its outputs every time any of its inputs changes at a given time step. If there are no *algebraic loops* in the interconnected system, then the data flow graph formed by (EQ 2) of all the subsystems taken together is acyclic, and this process is guaranteed to terminate. In fact, if the VHDL simulator maintains the processes in a topologically sorted order, then there will not be any unnecessary re-evaluations. If there are *algebraic loops*, then the above process is equivalent to a distributed relaxation algorithm for solving a non-linear fixed-point problem, and convergence is no longer guaranteed. The presence of such algebraic loops is usually an indication of poor modelling. The following simplified pseudo-code demonstrates the above algorithm. The procedure *odesolve()* uses the derivative function *f()* to calculate the state at the next time step.

```

t := NOW; tnext := t + deltaT;
L1: while TRUE loop
  -- solve for output vector u at the
  -- current time instant t using a
  -- distributed iterative algorithm
  L2: while TRUE loop
    v <= g(s, u, t);
    usave := u;
    wait on u for deltaT;
    exit L2 when NOW>t;
  end loop L2;
  -- calculate state at tnext
  odesolve(s, s, t, tnext, ...);
  if NOW<tnext then
    wait for tnext-NOW;
  end if;
  t := tnext;
  tnext := NOW;
end loop L1;

```

The only remaining problem now is that of calculating the state vector at time ΔT in to the future. Many numerical techniques are available in the literature for this. We use a fifth-order Runge-Kutta method with the monitoring of local truncation error to adapt the step size. The method is a good and robust method applicable in a wide class of problems. It keeps the error within a certain bound by taking a series of steps of appropriate sizes.

Implementation of VHDL Packages for Continuous-Time Modelling

The strategy outlined in the previous section has been implemented in VHDL. The lack of generic or template packages as well as function pointers in VHDL make an elegant implementation difficult. As shown by (EQ 1), the ODE solver routine inherently depends on the derivative function $f()$. This makes it impossible to write a single VHDL package that works everywhere. The package itself needs to be customized according to the derivative function $f()$. To overcome this problem a *template package* has been written from which a particular instant of the package is generated for a given $f()$ using a simple preprocessor program.

The package implements in VHDL the Runge-Kutta subroutines described in [Press88]. The template package is called *ode_<derivs>* where <derivs> is replaced by the name of the actual derivative function during the process of instantiation. The package makes available two functions

to the user:

a. *odesolve_rkdumb_<derivs>*

This function uses fourth-order Runge-Kutta method to advance in *nstep* equal increments from *t1* to *t2*.

b. *odesolve_rksmart_<derivs>*

This function advances from *t1* to *t2* with accuracy *eps* using as series of fifth-order Runge-Kutta steps.

Using one of these functions a continuous-time model is written according to the pseudo-code presented in the previous sub-section. Following is the VHDL code for the header of the *ode_<derivs>* package template:

```
--
-- File: ode_<derivs>.vhd
--
-- template package
library util; use util.types.all;
package ode_<derivs> is
  -- routines to solve ODE of the form dy/dx = f(x,y)
  -- the user supplied procedure
  -- <derivs>(x: in REAL; y: in REALVEC; dydx: out REALVEC; par: in
REALVEC);
  -- evaluates the derivative function f(). par is a vector of reals
  -- used to pass arbitrary parameters to <derivs>()

  -- Given y1 at x=x1 calculate y2 at x=x2 using 4-th order Runge-Kutta
  -- method to advance x in nsteps equal increments from x1 to x2
  -- y1 and y2 may be bound to the same array.

  procedure odesolve_rkdumb_<derivs>(y1: in REALVEC; y2: out REALVEC;
    x1,x2: in REAL; par: in REALVEC; nstep: in POSITIVE := 1);

  -- Given y1 at x=x1 calculate y2 at x=x2 with accuracy eps using
  -- 5-th order Runge-Kutta method with adaptive step-size control
  -- based on monitoring of local truncation error. h1 is the guessed
  -- first step size, and hmin is the minimum step size (>=0)
  -- y1 and y2 may be bound to the same array.

  procedure odesolve_rksmart_<derivs>(y1: in REALVEC; y2: out REALVEC;
    x1,x2: in REAL; par: in REALVEC; eps,h1,hmin: in REAL);
end ode_<derivs>;
```

In order to simplify using the package even further, the special case of linear (but possibly time-varying) systems is already implemented. In such a case the derivative function *f()* can be represented by an array of coefficients so that a general purpose VHDL procedure can be written.

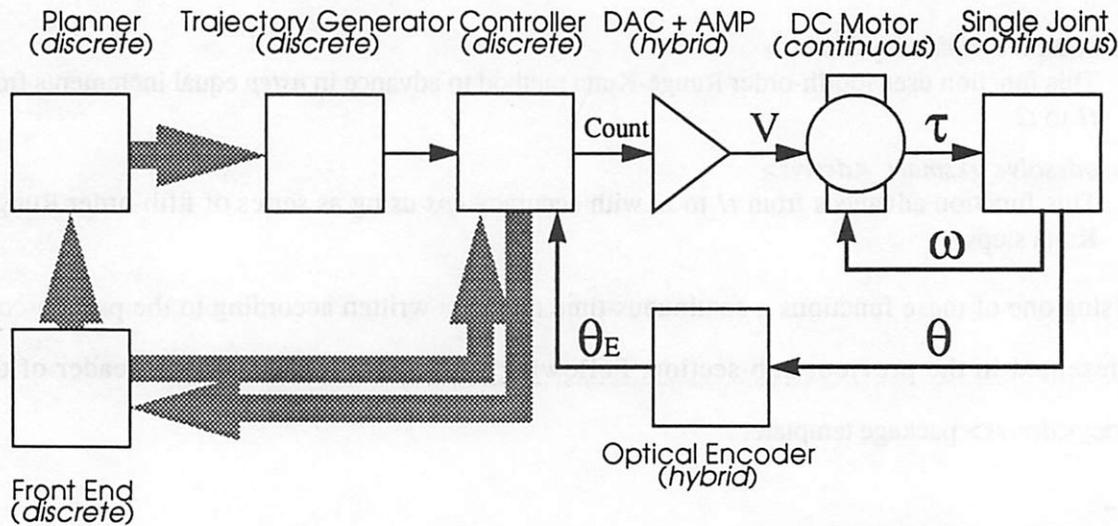


Figure 5-5 : A Simple Example of a Mixed-Mode System

For linear systems:

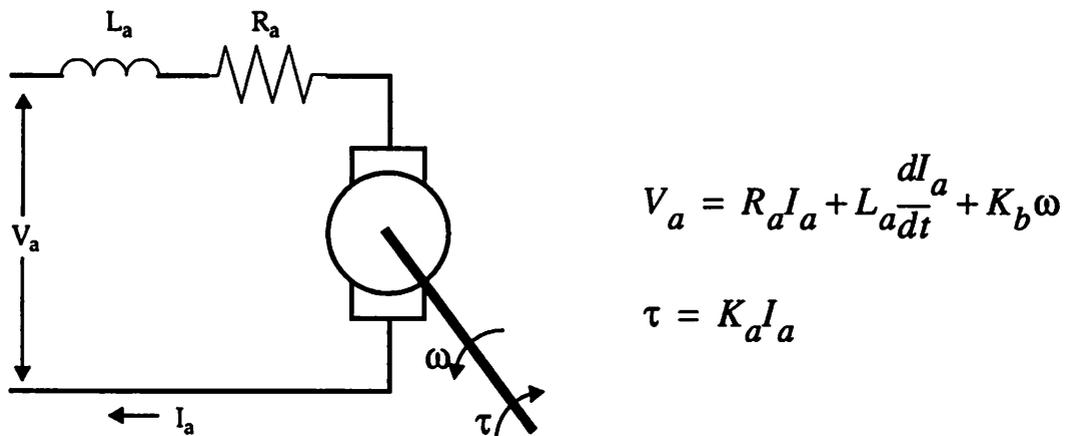
$$\frac{d\bar{s}}{dt} = A\bar{s} + B\bar{u} \quad (\text{EQ 3})$$

$$\bar{v} = C\bar{s} + D\bar{u} \quad (\text{EQ 4})$$

The package `ode_linderiv` provides functions `odesolve_rkdumb_linderiv` and `odesolve_rksmart_linderiv`. Using these functions a VHDL entity called `LTI` has also been provided. It implements a generic continuous-time *Linear Time-Invariant* block. The constant coefficient matrices A , B , C , and D , are passed as generics to the entity. Figure 5-4 shows the use of the `LTI` entity in modelling a DC motor.

5.3 Summary

In order to illustrate how the various packages discussed above are used in a complete high-level system simulation, Figure 5-5 presents a very simplified high-level block diagram of a robot controller. The intent is to show that even such a simple system is composed of blocks that require dramatically different modelling techniques. The DC motor is modelled as a linear time-invariant



```

entity DCMOTOR is
  generic( Ra, La, Ka, Kb: REAL;
    deltaT : TIME);
  port( Va, Omega: in REAL;
    Ia, Tau: out REAL);
end;
library CONT;
architecture STRUCTURAL of DCMOTOR is
  use CONT.lti_hdr.all;
begin
  M: LTI
    generic map( deltaT => deltaT,
      M => 2, N => 1, P => 2,
      A => (1=> (1=> -Ra/La)),
      B => (1=> (1=> 1.0/La, 2=> -Kb/La)),
      C => (1=> (1=> 1.0), 2=> (1=> Kb)),
      D => (1=> (1=> 0.0, 2=> 0.0),
        2=> (1=> 0.0, 2=> 0.0)),
      S0 => (1=> 0.0) )
    port map( u(1) => Va, u(2) => Omega,
      v(1) => Ia, v(2) => Tau );
end;

```

Figure 5-4 : Modelling a D. C. Motor in VHDL Using the Continuous-Time LTI Entity

continuous-time system, whereas the robot arm is described by a set of non-linear time-varying differential equations. On the other extreme, the planner, the trajectory generator, the front-end, and the controller represent blocks that are naturally described in terms of processes and inter-process communication channels. The optical encoder, and the digital-to-analog converter and amplifier are hybrids because they interface between the discrete event and the continuous-time parts of the system.

Using the packages described previously, a high-level simulation of such a system is very easily accomplished. In addition, using the native capability of VHDL for low-level digital hardware simulation, it is very easy to model parts of the system at the level of the actual hardware.

One conclusion that can be drawn from the discussion in the preceding sections is that VHDL presents many syntactic as well as semantic obstacles to doing such high-level system simulation elegantly. Nevertheless doing such simulation on top of a VHDL substrate offers the tremendous advantage of being able to mix high-level simulation with low-level simulation - something which is not feasible with dedicated high-level simulators. This provides a very powerful multi-level mixed-mode simulation environment. The various packages described in this paper demonstrate one way of accomplishing this goal.

CHAPTER 6

ARCHITECTURE GENERATION

The architecture of a system is its decomposition into hardware and software building blocks or modules, the specification of the behavior of these modules and the structural interfaces between them. Architecture generation refers to the process of producing a suitable architecture for the system starting from the specification of its behavior. In the framework presented here it would refer to the task of finding the best, according to some metric, hardware and software architecture for the process network description.

Architecture generation is largely an unsolved problem even for single chip designs, and it is a much more difficult problem at the system level. There is no formalized theory to relate the behavioral description of a system to its structure and therefore approaches used for logic and behavioral synthesis for chip designs are not directly applicable to system-level design. Formal hardware description languages and boolean equations based descriptions that suffice for chip-level hardware designs result in too much detail in case of a system design and render the task of realizing the system hopelessly complex.

However, a more important reason appears to be that the design of system architecture is still largely an art. There are few architecture models, or styles of hardware and software organization, for application-specific systems that are formalized enough so that they can be transformed and optimized by CAD tools. In contrast, in case of ASICs the architecture models are relatively well-understood as are the techniques to optimize them and the associated performance and cost metrics.

It is not very worthwhile to try automating the architecture generation process without first developing a model of system architecture that is both realistic and amenable to computer-aided module generation techniques. Therefore a major part of the research effort was devoted to studying and developing a system level hardware and software architecture model which is useful in a broad range of real systems. The architecture model is such that the process network description of the system can be easily mapped and hopefully automatically, and the model itself can be physically generated using the module generation techniques presented in the earlier chapters.

6.1 Approaches to Architecture Generation

One can classify the various approaches to architecture generation along two primary dimensions. The first dimension is the restriction on the architecture search space. This distinguishes the various approaches on the basis of factors such as computation and timing model of the input specification (e.g. data-flow versus event-driven models) and implementation style (e.g. hardwired implementation versus software for programmable components versus a mix of the two, single chip versus MCM versus board, synchronous versus asynchronous etc.). The second dimension is the degree of automation of the search mechanism which can range from fully manual to fully automated. The usefulness of an architecture generation approach depends not only on the efficacy of its search mechanism but also on the restrictions placed on the search space. Highly automated

approaches with too restrictive an architecture model - such as the behavioral synthesis approaches used for single chip designs - are not sufficient for most systems.

As mentioned in Chapter 1, this thesis is targeted at real-time reactive systems implemented as a mix of hardwired and software programmable components at the board level. This naturally restricts the search space somewhat - but the problem remains far more complex than for ASIC or pure software designs. Since not much previous work exists on architecture definition for such board-level systems, it would be useful to look at the architecture generation approaches adopted when the search space is different, such as for ASICs or for pure software implementations. The following discussion is therefore organized according to the degree of automation of the architecture generation process, and within each category examples are presented for not just board-level systems but also for chip and software domains.

6.1.1 Manually Specified Architecture

This is almost the universal approach currently used for defining the system architecture. Even in case of single chips, despite the availability of silicon compilers and behavioral synthesis systems, manually specified architectures are still widely used, particularly for high-performance or area-sensitive chips such as general-purpose microprocessors. A system designer using this approach can nevertheless use the hardware and software module generation aids described earlier. Considerable time is required for this approach, and is therefore not satisfactory from a rapid-prototyping perspective.

6.1.2 Fixed Architecture

Perhaps the simplest approach to automating the process of architecture definition is to use a fixed architecture. This approach has been used for ASIC design in early silicon compilers, such as LAGER-I [Rabaey86], where the behavioral description of the algorithm was mapped to a processor with a fixed architecture consisting of a fixed datapath and a microcoded controller. The

problem of silicon compilation is thereby reduced to that of generating the controller - a problem which is equivalent to that of software compilation. The fixed architecture approach is often used for high-performance implementations of ASICs for narrowly defined application domains where the optimal architectures are known. FIRGEN [Jain91], a compiler for FIR filters built on top of LAGER silicon assembler, is a case in point. Starting from the frequency domain specification it generates a FIR filter with a fixed architecture. Implementation of an algorithm in software on a programmable signal processor is also a trivial example of this approach.

The fixed architecture approach is also very common at the system level. Any system based on a general-purpose computer belongs to this category, at least from the perspective of hardware organization. A widely used approach to implementing a real-time reactive system is as a software system running on multiple single-board computers connected using a standard bus, such as the VME bus, together with off-shelf I/O cards. There are other similar multi-processors with different, and often software configurable, interconnect topologies. Figure 6-1 shows organization

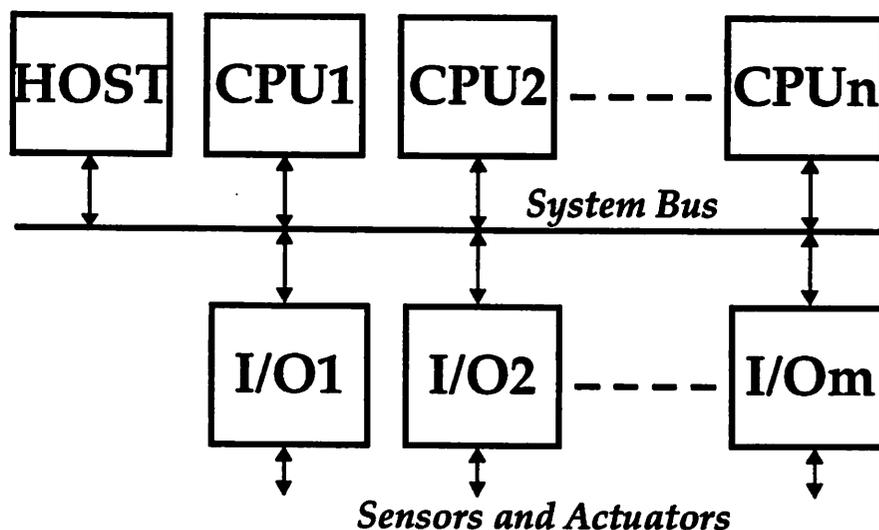


Figure 6-1 : Typical System Based on General-Purpose Multi-Processor Architecture

of a typical system based on such a general-purpose multiprocessor architecture. With a fixed hardware organization the problem of system architecture definition reduces to that of a distributed software system. There are tools which can partition the system specification to software for each of the processors. One such tool is McDAS [Hoang92] which can map a DSP algorithm described in the dataflow language SILAGE to multiprocessors with a wide variety of interconnect topologies. Other similar tools are GRAPE [Lauwereins90] and GABRIEL [Lee89].

This approach is quite restrictive since it is typically based on general-purpose off-shelf processors/computers. The efficiencies of a more customized solution are thus not available.

6.1.3 Architecture Template

Architecture template based approaches are a compromise between the inflexibility of the fixed architecture approach of the previous section and the difficulty of the synthesis approach described in the following section. The basic feature of this approach is that while the global architecture is fixed, many of the individual modules are allowed to be synthesized or user specified. The fixed global architecture simplifies the synthesis process while the flexibility in the architectures of the individual modules allows customization for performance. An example of this approach is the C-to-Silicon Compiler which is part of LAGER [Shung91]. All the ASICs generated by it have the same top-level architecture based around a microcoded FSM controller, an arithmetic datapath, an address generation datapath, a boolean condition flag FSM and an I/O unit. However, the structure of the two datapaths can be user specified, or generated by an even higher level tool, and the compiler retargets the microcode generator according to the structure of the datapaths. Cathedral-II [Rabaey88] is another silicon assembler which follows a similar architecture template philosophy.

The architecture template approach is attractive at the system level as well. As shown in Figure 6-2 the system hardware organization follows a template based on multiple application-specific boards which are individually customized. The software is then mapped onto the hardware template. The details of the architecture template - hardware and software organization - are

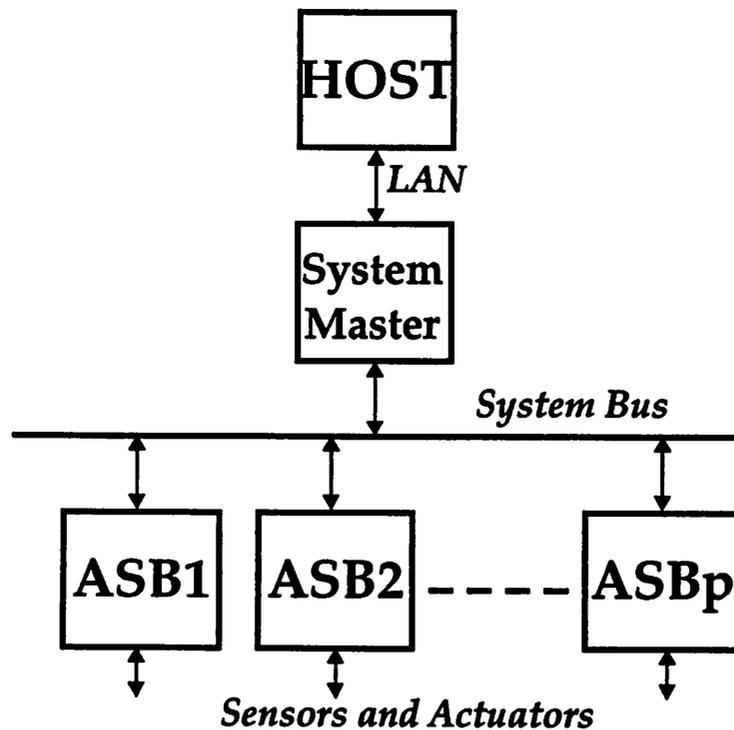


Figure 6-2 : System Architecture Using Application-Specific Boards

described later in this chapter.

The approaches taken by MICON and Vulcan-II, discussed in Chapter 1, can also be classified as an approach based on architecture templates. MICON uses an expert system to generate single-board computers based around commercial microprocessors. The computers designed by MICON consist of a microprocessor, ROM, SRAM or DRAM, cache memory if supported by the processor, serial and parallel I/O ports, standard bus interface, and support circuits such as address decoding, clock generation etc. The search space is organized as a tree or a lattice which reflects the top-down structural decomposition and the implementation choices available at any stage - in effect this lattice structure defines an overall parameterized architectural template. Given user input, such as the choice of processor, amount and type of memory, number and type of I/O ports etc., and constraints on performance, such as the clock speed, reliability etc., the expert system

traverses the search space (the architecture template) using rules entered by experts to come up with a solution that satisfies the design constraints. The search strategy in MICON is fully automated although the architecture space is restricted. Similarly Vulcan-II uses an architecture template consisting of a single microprocessor and multiple ASICs, and then uses an automatic partitioning approach to generate the precise hardware and software organization. The architecture model of Vulcan-II deserves particular attention because, as is the case here, Vulcan-II is also targeted at a mixed hardware-software implementation. The architecture model of Vulcan-II was studied in depth in Chapter 1.

6.1.4 Automatic Architecture Generation

Fully automatic architecture generation has proven feasible only in cases where the search space is sufficiently restricted - there are no examples of this approach at the system level. For example, there are architecture synthesis tools for single ASIC design. However, none of these tools can claim to give the optimum architecture because invariably these tools place further restrictions on the ASIC design by having a certain *model* of architecture which is reflected in the synthesized architecture. Example of such architecture models are the clock synchronous hardware model based on multiple communicating finite state machine controllers used by OLYMPUS [Micheli90], and the clock synchronous architecture model of a single finite state machine controller and a datapath consisting of one or more heterogeneous functional units with register files at inputs and the outputs feeding the register files that is used by HYPER [Rabaey91].

Experience at chip level suggests that automatic architecture generation works best in a well defined application domain with a representation language and architecture model suited for that domain. A unified language and architecture model that works for all domains appear infeasible, and is not a worthwhile strategy to pursue for achieving automatic architecture generation at the system level because most systems are characterized by multiple heterogeneous domains. Instead, divide-and-conquer approaches similar to that adopted by Ptolemy (see Chapter 1) for simulation

appear more likely to succeed. Effort should be directed towards defining a small set of computation models, or domains, that together suffice to describe most systems, developing automatic architecture generation techniques for each of these domains, and, most importantly, rigorously defining the interface between two different domains at the specification level as well as the architecture level. In Ptolemy “worm-holes” provide the unified simulation interface mechanism between parts of a system (“universes”) that belong to different domains. The process network representation described in the previous chapter provides a similar, rigorously defined, interface between parts of a system using the notion of channels. The processes themselves may be defined in languages best suited for naturally expressing the respective computations - in fact the process network representation does not define any language for this. Research should be directed towards developing a small set of languages to represent the computation within the processes, and automatic architecture generation or synthesis tools for each one of them. Together with the techniques described in this chapter for implementing the channels for inter-process communication and synchronization, such research can provide a solution to the problem of system-level automatic architecture generation.

6.2 Co-Design of Hardware and Software

A distinctive characteristic of system level design is the use of a mix of dedicated hardware as well as software programmable hardware. Although it is certainly possible to realize the functionality of a system either purely as software running on programmable components such as microprocessors, or purely by imbedding the algorithms in dedicated hardware as done in some ASICs, the most cost-effective solution usually lies between the two extremes. The example systems in Chapter 1 amply demonstrate this point. This mix of dedicated and software programmable hardware is dictated not just by cost-effectiveness but also by the sheer ability to realize the functionality in current technology. Logically complex control oriented tasks such as

protocol processing are often feasible only in software whereas real-time signal processing and sensor I/O are often possible only in hardware.

A key problem in the architecture definition of systems using a mix of dedicated and software programmable component is that of *Hardware-Software Co-Design*. Given the system specification one has to:

- decompose the system functionality into parts implemented using dedicated hardware components and parts implemented as software for programmable hardware components.
- the organization and implementation of the dedicated as well as programmable hardware modules, and of the software running on the programmable hardware modules.
- communication and synchronization between the various parts of the system whether implemented as dedicated hardware or as software.

The first problem is that of partitioning. It involves transforming the system specification such that a good partitioning into hardware and software components can be obtained and then actually partitioning it. The second problem involves finding out a good model or template according to which the system hardware and software should be organized. Chapters 3 and 4 have already dealt with the related low level module generation issues. The third problem involves finding out the appropriate mechanism for communication and synchronization between the different parts of the system. These three problems are actually closely related. For example the partitioning depends on the architecture model and the cost of the available communication and synchronization mechanisms. The solutions to these problems is described later in the chapter in the form of an architecture template, the communication mechanism, and a methodology for partitioning which is currently manual but can be made automatic.

The distinction between what constitutes a software implementation and what constitutes a dedicated hardware implementation is not always clear. The strict definition of software is that it is the information needed to program a general-purpose computing device to do a specific task.

According to this the bit pattern stored in the memory locations of a Xilinx FPGA to implement, say, a FIR filter could be regarded as software. However, in this thesis, the term software implementation is being used only for traditional processors that are programmed using assembly language instructions that are stored in RAM or ROM. Computation structures programmed into FPGAs using bit patterns stored in memory locations are considered as configured hardware. Core-processors with fixed on-chip micro-code are also considered to be hardware because despite the use of a compiler to generate the micro-code, the entire processor needs to be fabricated again in order to alter the program.

6.3 Template Mapping Based Approach to Architecture Definition

The approach used here is based on mapping the system specification in the form of a network of processes as described in Chapter 5 to an architecture template. This approach was chosen because a major goal of this work was to have a complete top-down system design methodology and a template based approach appeared to be the best compromise between practicality and degree of automation.

Figure 6-3 shows an overview of the architecture definition approach, and the main focus of this chapter is on the template used to realize the system architecture. As elaborated in the following subsections, the system represented as a network of processes as described in Chapter 5 is partitioned on to a parameterized architecture template to produce a structure of the system in terms of specifications for hardware and software modules, and the communication and synchronization between them. Each process in the process network description is mapped either to a *software process* running on a programmable processor module, or to a *hardware process* running on a dedicated hardware module. The system is thus partitioned according to the coarse grained parallelism explicit in the process network description.

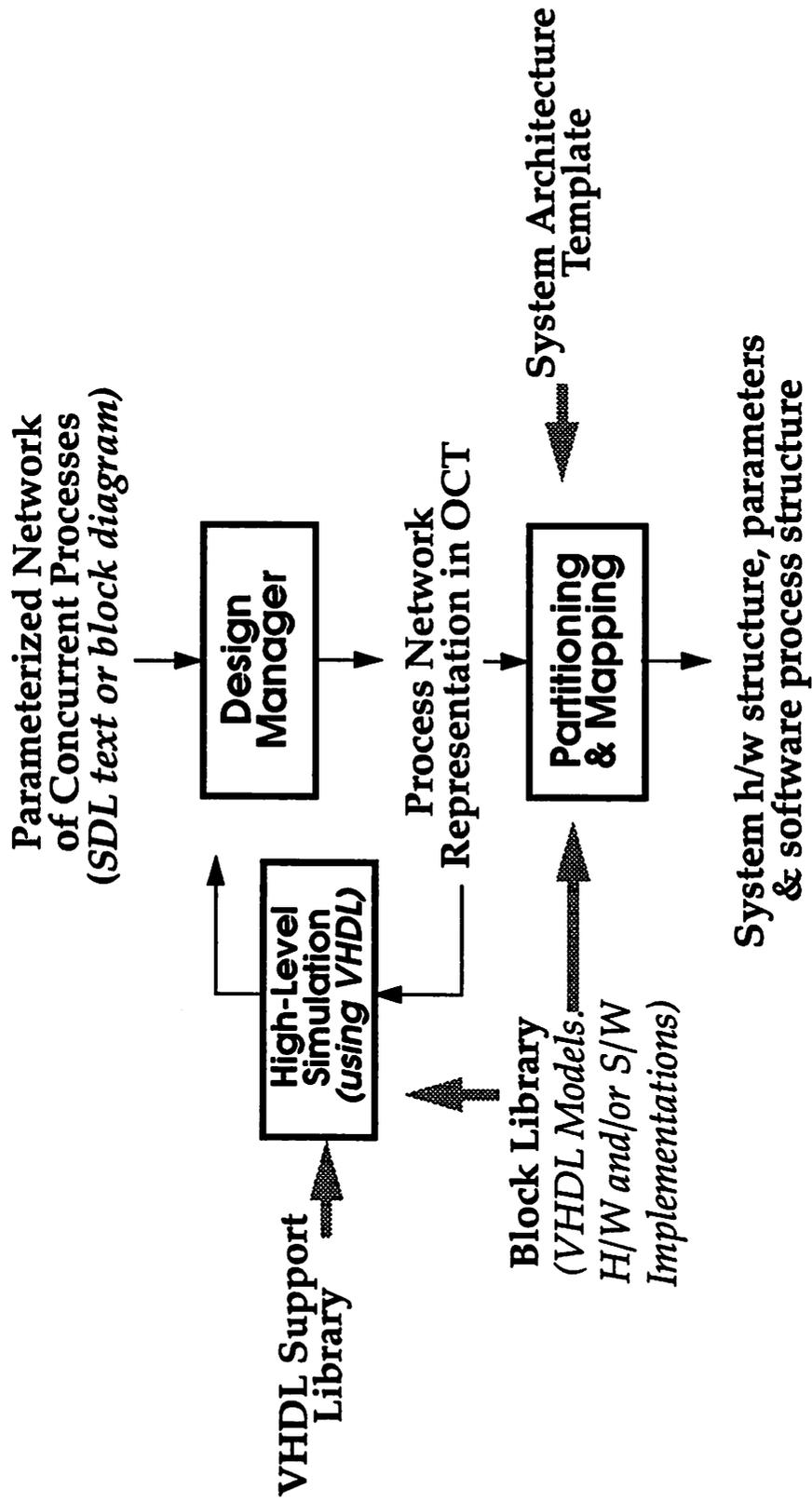


Figure 6-3 : Template Based Approach to System Architecture Definition

The following sections describe in detail the architecture template and the rationale behind its choice, the partitioning methodology, and mechanisms for communication and synchronization between the *hardware processes* and *software processes*.

6.4 The Layered System Architecture Template

Based on an analysis of the architecture of many existing real-time reactive systems a parameterized template for the architecture of such systems was chosen as the basis for the architecture definition approach presented here. As shown in Figure 6-4, the architecture template chosen has a layered, distributed hardware structure with a hierarchical bus organization for increased bandwidth. However before going into the specifics of this architecture a study of various alternatives for system architecture is done below in order to justify the choice of this particular architecture.

6.4.1 Alternative Models of System Architecture

The goal in defining the architecture template was to come up with a scalable family of architectures that allows for a systematic interconnection of an arbitrary number of heterogeneous processing nodes that can be based on dedicated hardware as well as software programmable hardware modules. Of course the architecture template needs to be able to provide sufficient computation and I/O capability for systems of interest such as described in Chapter 1. Allowing heterogeneous processing modules which can be dedicated hardware as well as software programmable will allow the resulting architectures to take advantage of the extensive suite of module generation tools and libraries presented in Chapters 3 and 4. The capability to systematically connect these processing modules is needed to both simplify the problem of communication and synchronization between these modules and the problem of exploring this family of architectures to select a good one.

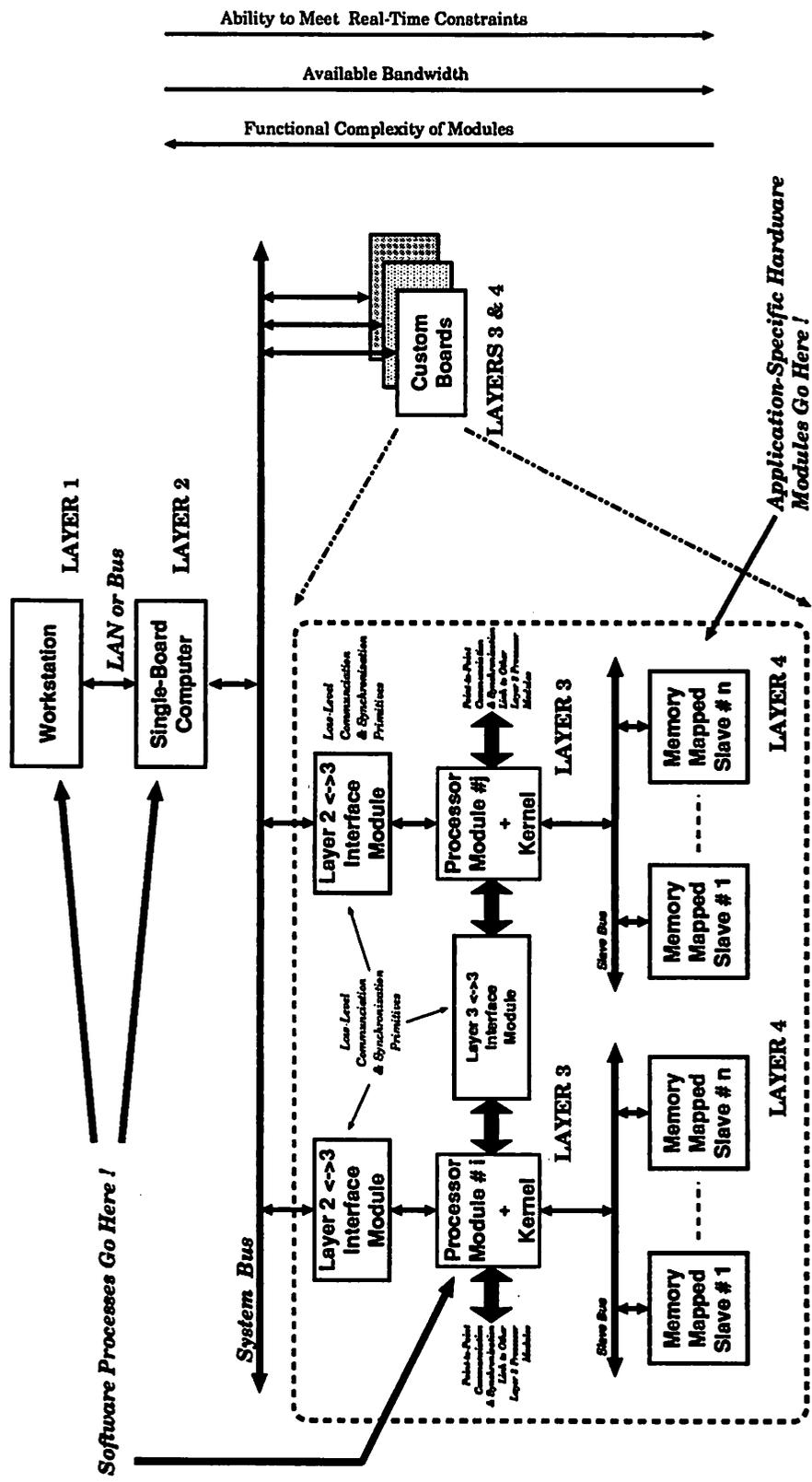


Figure 6-4: A Layered Architecture Template for Systems

Existing Architecture Models for Purely Software-Based Systems

Most system level architecture models in literature deal only with the homogeneous software programmable processing nodes. Although such architecture models are not applicable to the problem in this thesis, nevertheless they can provide examples of useful interconnect topologies as well software organization.

An interconnect network lets any two processors communicate whenever they need to. Of course providing full connectivity for N processors will require $N(N-1)$ point-to-point connections which is too costly. Therefore a variety of interconnect topologies with varying mix of partial connectivity and multi-access (as opposed to point-to-point) links have been used in existing systems based on multiple software programmable processors. The key metrics to evaluate a interconnect topology are its diameter d (the maximum distance between two processors using the shortest path), its connectivity k (the minimum number of different paths between two nodes), its flexibility, and its communication delay. The total communication delay (T_D) is itself composed of the computation time required to set up and consume the data (T_{PROC}), the time spent while waiting to be sent due to contention or synchronization (T_{QUEUE}), and the transmission time ($T_{TRANSMIT}$). Of these three components T_{PROC} depends on speed of the processors, T_{QUEUE} is a function of the access protocol (e.g., point-to-point versus multi-access), and $T_{TRANSMIT}$ depends strongly on the spatial topology and the communication bandwidth and protocol. It is desirable to have a topology that has a small diameter, a high connectivity, a high flexibility, a small communication delay, and a low cost.

Although the generic term *processor interconnect network* is being used in this discussion, the issues remain substantially the same whether the *processor* is an active computation device or an I/O device or even a storage memory device.

a. Point-to-Point Interconnect Networks

These networks are characterized by being composed of a number of point-to-point

communication links between pairs of processors. A fully connected network would be characterized by $N(N-1)$ links for N devices or processors and is very costly for large N . Therefore partially connected topologies such as arrays (1-D, 2-D and 3-D) and hypercubes are commonly used. If the pattern of data flow in the system directly maps on to the actual topology of the network then such point-to-point networks provide very efficient solutions. These networks are easy to implement because the communication links are point-to-point, although sophisticated routing algorithms may be required if connectivity $k > 1$. For topologies such as hypercube the network interface of any individual processor has to change as the number of processors changes which can be an impediment to modularity.

b. Switching Networks

These interconnect networks are characterized by a multi-access switching structure that handles routing of data between processors. The switching structure itself is characterized by the bandwidth and number of simultaneous communication links it can support. The switch needs to be able to provide an adequate bandwidth so as to avoid becoming a bottleneck. A very high speed switch with sophisticated data routing algorithm may be required if the number of processors is large. Examples of switching networks include:

- *Star Switch* which uses $O(N)$ switches while allowing any one pair of processors to communicate simultaneously with a $O(1)$ delay.
- *Crossbar Switch* which uses $O(N^2)$ switches while allowing any disjoint $N/2$ processor pairs to communicate simultaneously with a $O(1)$ delay.
- *Butterfly Switch* which uses $O(N * \log N)$ switches while allowing a restricted set of $N/2$ processor pairs to communicate simultaneously with $O(\log N)$ delay.

c. Single Shared Bus

This is the simplest and probably the most widely used topology. Simple multi-processors based on multiple single-board computers on a standard bus such as the VME bus are very commonly used for embedded real-time software systems. All the

processors communicate by time-sharing the common multi-access bus which is a broadcast medium that is accessed according to some arbitration protocol. The bus bandwidth is shared between all the active communication links, some fraction of which is wasted due to the inevitable protocol overhead. The shared bus topology is characterized by a diameter $d=1$, connectivity $k=1$, a high flexibility, and a high T_D because of a high T_{QUEUE} which increases as the number of processors that can initiate bus transactions. The bus interface for any individual processor is independent of the number or type of other processors on the bus - a certain level of uniformity and modularity is implicit in the bus structure.

One of the key design consideration for a shared bus architecture is the arbitration protocol used to allow orderly access to the bus by multiple processors. It has a direct impact on T_{QUEUE} and therefore on the utilization of bus bandwidth. Many bus arbitration strategies are available that provide different trade-offs between implementation complexity and waiting time T_{QUEUE} . The various bus arbitration strategies can be classified into two categories: *single-master* and *multiple-master*.

In the case of single-master bus-arbitration strategies only one device on the bus is allowed to initiate a bus transaction, the other devices are slave devices that wait for the master to service them. This has the advantage that no bus arbitration is needed because only one device can initiate transactions. However some strategy is required to minimize the time a slave has to wait before being serviced. There are two aspects to this problem - one is how to indicate to the master that a slave needs servicing, and second is the policy according to which the master should service slaves that are ready. Polling and interrupts are two techniques for solving the first problem. Least recently serviced, first come first served, priority servicing (daisy chaining), round robin are some of the common servicing policies.

In the case of multiple-master bus arbitration more than one device can initiate a transaction and therefore a policy is needed according to which the bus is granted to

devices that need to initiate a bus transaction. Round robin, priority based (daisy chaining) and first come first served are common arbitration policies. Note that in the case of multiple-masters there may in addition be multiple slave devices with each master possibly servicing multiple slave devices and, though less common, each slave device being serviced by more than one master. Techniques similar to those in the case of single-master case can be used to service the slaves even in the multiple-master case.

d. Hybrid Networks

The three approaches discussed above have different sets of positive and negative features and work well in different situations. Therefore network topologies that are variations on the above themes or combinations of these themes are often used to get improved performance. For example, a hypercube topology might be combined with a 2-D mesh in order to obtain good performance on a wider class of algorithms. Similarly, processors might use a global hypercube interconnection topology but use a local bus structure to communicate with I/O and memory devices. In general the interconnection network may have different structures at different levels of granularity. The layered architecture template used in this thesis also belongs to this category of hybrid networks.

Existing Architecture Models for Mixed Hardware-Software Systems

There are few formal architecture models described in literature for systems that contain both dedicated hardware and software programmable hardware. One example is the architecture model of the Vulcan-II system for hardware-software co-design that was discussed in detail in Chapter 1. The model in Vulcan-II, based on a single processor and multiple-ASICs that communicate using a shared memory on a multi-master bus, is targeted at much smaller scale systems than those that are of interest here, and appears to be motivated primarily towards automating the partitioning problem. An unfortunate effect of this is that the architecture model has many weaknesses,

outlined in Chapter 1, that make it difficult to implement. Indeed, the architecture model of Vulcan-II remains to be proven on any real design.

However, an advantage of the Vulcan-II architecture model is that it is extremely simple, and therefore allows an automated partitioning approach. This probably just reflects a different design decision than from the one made here - the usability of the architecture model for actual applications was deemed critical.

6.4.2 Details of the Layered Architecture Template

As already mentioned the chosen architecture template shown in Figure 6-4, has a layered, distributed hardware structure with a hierarchy of busses for increased bandwidth while retaining the simplicity and uniformity of module interfacing provided by busses.

There are four layers in the architecture template. The bottom two layers of the hierarchy are spanned by custom boards. Each custom board has one or more software programmable processor modules based around programmable microprocessors or digital signal processors. Each of these processor modules runs a real-time multi-tasking OS kernel that is configured according to the hardware resources. This provides the ability to dynamically schedule in an event-driven fashion the tasks mapped to a given processor module. Each processor module in turn coordinates a number of application-specific slave modules which can be either software programmable or dedicated hardware modules. These slave modules in the lowest layer of the hierarchy are the only place where dedicated custom hardware modules can appear. The custom boards spanning the bottom two layers of the hierarchy in turn sit on a back-plane bus such as the VME bus, and are slaves to another processing module which is the bus master. This processing module belongs to the second layer of the hierarchy and also runs under the control of a real-time OS kernel. Typically this processor module is an off-shelf single-board computer using a standard real-time OS kernel such as VxWorks. The layer 3 processing modules on the custom boards interact with the master processing module on layer 2 through a bus interface module that is required to support

the hardware and software communication and synchronization primitives discussed in the previous section. For example, one such interface module is a dual-port RAM based VME interface that provides the layer 3 processor with a block of memory that is also accessible to the layer 2 processor. In addition, the module also provides mail-box interrupts and hardware semaphores using which the required higher level communication and synchronization primitives are implemented in software. The layer 2 processor module is in turn slave to a conventional workstation (for example, a SPARCstation) which constitutes the topmost layer in the hierarchy. The communication between layer 1 and layer 2 is typically over a network such as ethernet or FDDI that supports high-level protocols such as TCP/IP.

According to the taxonomy of interconnect networks presented in the previous section, the topology of the layered architecture model can be classified to be a hybrid interconnect topology having features of both a point-to-point tree network, and a shared bus network. The hierarchical bus organization is actually a tree-like structure where the nodes of the tree are clusters of programmable and dedicated processors. Within each cluster the processors are connected using a single shared bus architecture. Some of these processors act as gateways between clusters at different levels of the tree. The processors on the bus within a cluster can in general be either bus masters or bus slaves. Typically the processor that connects to a higher level cluster is the bus master and the other processors are bus slaves. These bus slave processors may in turn be gateway processors that are bus masters in a lower level cluster.

An advantage of such a tree structured hierarchy of busses is that the peak bandwidth is much greater than that for a single bus while much of the simplicity and uniformity of the single shared bus architecture is retained. Many real-time reactive systems follow a multi-level hierarchical control strategy with a high communication bandwidth requirement between tasks at the same level and a relatively lower communication bandwidth requirement between levels. The hierarchical bus organization naturally supports such hierarchical control strategies. For example, the standard hierarchical control architecture for integrating multiple sensors in a multi-robot

environment called the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) [Albus89] can be implemented efficiently as a hierarchy of busses.

One feature of the architecture template omitted from the above description is that the layer 3 processor modules are also allowed to have direct point-to-point communication links between them. This allows the layer 3 processors to communicate with each other with a high bandwidth without making the single layer 2 processor a bottleneck. A result of this is that the topology of the architecture template is not strictly a tree. Further, the topology of these point-to-point links in layer 3 can be arbitrary and is application dependent. However, in the current implementation, a maximum of two such links is allowed for each processor. This means that the topology of this layer 3 network is such that the processors are clustered into independent groups where the processors within the group are connected in a ring or a linear array using these point-to-point links. These clusters are defined according to the needs of the application. Another point to observe is that with this enhancement the communication path between two processors is no longer unique - a property which is true for a pure tree structure. For example, two layer 3 processors on the same cluster can communicate either using the layer 2 processor as an intermediary, or can communicate using a path that is routed through other processors in the cluster. It is not obvious which path is better as it depends on the implementation and the relative lengths of the two paths. This further complicates the partitioning problem, but was considered desirable for performance reasons. If the clusters are small in size, as will be the case just because of board technology limitations, then it is almost always better to use these dedicated point-to-point links.

6.5 Communication and Synchronization Mechanisms for the Layered Architecture Template

The architecture template described in the previous section specifies only the organization of the dedicated and software programmable processors to which the processes in the process network

description of the system are mapped. The other crucial aspect of this problem is the communication and synchronization between these processes that are mapped to an instance of the layered architecture template. The processes in the process network are connected using the FIFO channel mechanism as described in Chapter 5, and the channel interconnect pattern in the process network can be arbitrary. This arbitrary channel interconnect pattern needs to be embedded in the tree-like physical interconnect network of the layered architecture template.

There are two problems that occur in this embedding. First, two processes that are adjacent in the process network (i.e. connected using a channel) may not be mapped to processors that are adjacent (i.e. having a physical communication link) in the layered architecture template. The channels between such processes therefore need to be routed using a path going through other intermediate processors. As mentioned in the previous section this problem gets complicated by the point-to-point links between the layer 3 processors because the path is no longer unique. One way to make the path unique is to always give preference to a path which stays within layer 3 instead of the one that gets routed through the layer 2 processor. It should also be noted that layer 4 processor modules can never be intermediate processors on a path because they are the leaf nodes of the architecture template. Therefore the intermediate processors through which a communication path is routed is always a software programmable processor. Given this the implementation strategy is very simple - since each software programmable processor runs a multi-tasking kernel, special routing tasks are created on these processors, one for each path being routed through the processor. These routing tasks just help move data from one neighboring processor to another. One implication of this is that routing a path through an intermediate processor may require substantial computational resources in addition to the I/O bandwidth. Careful partitioning and intelligent use of dedicated point-to-point links is therefore extremely important.

The second problem related to embedding the channels of the process network in an instance of the architecture template is that of the actual implementation of the required channel behavior. This

important issue is the focus of the rest of this section. Its importance is due to the fact that the protocol behavior of the FIFO channels and ports in the process network, as described in Chapter 5, is defined at an abstract level to simplify description of the system. However, this behavior is at too high a level to be directly implemented in hardware. Implementing the channels themselves requires a judicious use of low-level communication and synchronization primitives, both hardware and software. Deciding what are the appropriate low-level primitives for efficient implementation of the channels in the process network, in the context of the architecture template defined earlier, is the problem being addressed in this section.

There are two aspects to the problem of implementing the channel abstraction of process interaction - *Communication* and *Synchronization*. Communication deals with the physical transfer of data. Synchronization is required to accomplish the data transfer while meeting the protocol constraints - it is the mechanism through which constraints are placed on the actions being taken by the two communicating processes to perform the data transfer. For example, it is through the synchronization mechanism that the port protocols discussed in Chapter 5 can be enforced.

6.5.1 Communication and Synchronization in Software

There is a very rich literature dealing with the issues of communication and synchronization, and many formal techniques have been developed and used in distributed software systems and operating systems. These ideas equally applicable to hardware as well, although traditionally hardware designers have not treated the problem in as formal a fashion as the software designers. Hardware implementations often make assumptions about the relative speed of the interacting processes, or their synchronization to a global clock. This may suffice for the design of a particular system but the inherent lack of modularity of such approaches make them difficult to use in a system design environment based on reusable libraries.

Communication can be accomplished by one of the two basic techniques - by using shared

memory accessible by the sender and the receiver, or by sending a message from the sender to the receiver over a network. The nature of the underlying hardware has a major role in deciding which of the two techniques is better suited in an application. If the processors on which the communicating processes are running are tightly coupled, such as on a VME bus, or are the same processor, then shared memory is usually better suited because the hardware itself provides memory that is accessible by both the processes involved in a communication. On the other hand, if the two processors are separated by a network such as ethernet, then message passing is usually better suited because the hardware itself is intrinsically message oriented. It is usually inefficient to simulate shared memory when there is none.

The two basic mechanisms of communication have their own distinct synchronization requirements. In the case of shared memory based communication two types of synchronization are required - mutual exclusion and condition synchronization. Mutual exclusion type synchronization is needed to enable the processes to perform atomic operations on a shared object, for example operations on a data structure stored in the shared memory. Condition synchronization is needed to coordinate the actions of the communicating processes when the state of a shared object is inappropriate for a particular operation. A process attempting that operation needs to be delayed until the state of the shared object changes to an acceptable one as a result of actions by the other process. For example, a process writing data into a buffer area in the shared memory may have to be delayed if the buffer is full.

In the case of message-passing-based communication, synchronization is a natural outcome of the act of communication - a message can be received only after it has been sent. This lets the receiver make some assertion about the state of the sender. In addition, if the message passing is synchronous (unbuffered) then it represents a point in time where the execution of the sender and the receiver are synchronized. The message received therefore corresponds to the current state of the sender, and the sender can also make assertions about the state of the receiver.

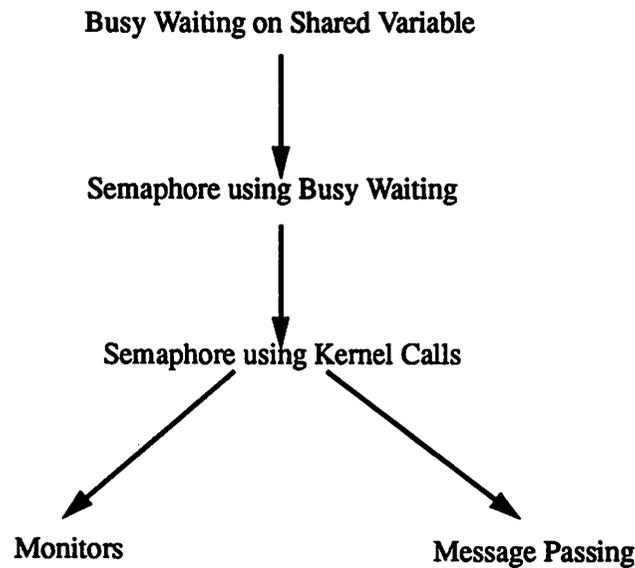


Figure 6-5 : Synchronization Techniques in Software Systems and Their Relationships

Many primitives have been developed for implementing synchronization in software systems. Figure 6-5 shows the relationship between some of these primitives. These techniques include simple busy waiting or spin-lock based synchronization, semaphore using busy waiting, semaphore based on system calls to a kernel which schedules processes, monitors, message passing, remote procedure call, etc. A good discussion of the details and relative merits of these techniques can be found in [Andrews83][Ben-Ari90].

6.5.2 Applicability of Communication and Synchronization Techniques in Software to Dedicated Hardware

The techniques for communication and synchronization in software systems provide a good conceptual framework in which to implement the channels in the architecture template. However it is not possible to use the same implementation techniques because the software techniques were mostly developed with general-purpose computers in mind. Their implementations make assumptions that are valid for general-purpose computers but may be inefficient for an application specific system.

For example, the implementation of the techniques presented in the previous section usually require the shared memory to satisfy certain properties. The most common model of the memory is that of a collection of *atomic* registers or locations. Overlapping reads and writes to such memory locations by two processes are equivalent to an arbitrary interleaving of the two accesses. The hardware implementation of a memory with such atomically accessible locations is done using an external hardware arbiter. This does not matter in general purpose MIMD computers with large shared memories implemented using single-ported RAMs - the single access port or bus to the memory requires arbitration anyway, and therefore all accesses to the memory are inherently atomic irrespective of the address. However, in the case of dedicated communication interfaces true multi-ported memories may be used. In such cases, the accesses on the multiple ports are completely asynchronous and can proceed simultaneously in time. An arbiter is required to resolve access contention only when the locations being simultaneously accessed on the different ports have the same addresses. This not only adds to the hardware complexity but also has an adverse effect on all memory access times because the arbiter has to check for simultaneous access to the same location even when the two processors are accessing different locations in the same memory.

Thus for hardware simplicity it is desirable that the communication and synchronization primitives be implemented using strategies that can directly work with such multi-port memories. There do exist communication and synchronization algorithms that can work with weaker models of memory such as Lamport's bakery algorithm for mutual exclusion which can work with a memory composed of *safe* registers. A safe register is one where in the case of a overlapping read and write accesses, the write executes correctly and the read is allowed to return any value in its allowed range. A write-write contention is not allowed. However common multi-port memories without an external arbiter, such as the IDT7025 dual-port memory, are an even more permissive form of shared memory - they are not even safe. In the case of overlapping access to the same memory location from two ports, both accesses may fail. A write may result in corrupt data being written, and a read may return any value. This is too weak a memory model to implement communication

and synchronization between the two processes and additional hardware support is needed. One of the examples in the next section will describe a hardware module for low-level inter-processor communication and synchronization that is based around a true dual-port memory with the above properties.

6.5.3 Implementation of the Channels in the Architecture Template

As should be obvious from the discussion in the previous section, the techniques used for communication and synchronization depend heavily on the underlying hardware. In the case of our architecture template the interfaces between the various layers are quite different, and therefore need to be treated separately. The following subsections describe the implementation of the channels depending on where the sender and the receiver processes are located.

Case 1: Sender and Receiver both on Layer 1 Workstation

At present this case is not being used although an implementation has been done by a co-researcher [Lee91] using the light-weight process library from SUN. The basic idea is that all processes from the process network get mapped as light-weight processes, or threads, inside a single UNIX process (heavy-weight process). The light-weight processes are distinguished from the heavy-weight UNIX processes by the fact that all the light-weight processes inside a heavy-weight process share the same address space and system resources such as file descriptors. In contrast, heavy-weight processes work in separate address spaces. As a result communication, synchronization, and context-switching are much cheaper in the case of light-weight processes than in the case of heavy-weight processes - hence the names *light-weight* and *heavy-weight*. The light-weight process library provides several primitives for communication and synchronization between the light-weight processes. These primitives include monitor and synchronous message passing. In addition, all the light-weight processes share a common address space. The channel

objects for communication between the light-weight threads are then constructed using these primitives.

A drawback of the current implementation is that whenever any of the light-weight processes attempts to do an I/O from outside the UNIX process then all the light-weight processes also block. This problem is really an artifact of the fact that the light-weight processes are not scheduled by the underlying UNIX operating system, but nevertheless causes inefficiencies. An alternate implementation is a mix of light-weight processes and heavy-weight UNIX processes, or the use of an operating system that explicitly supports light-weight processes.

Case 2: Sender on Layer 1 Workstation and Receiver on Layer 2 Processor (and vice versa)

The layer 1 and layer 2 processors communicate over a local area network which is inherently a message passing based medium. The channel objects need to be implemented using one of the standard communication protocols such as UDP or TCP, or higher level protocols such as RPC protocol from Sun Microsystems. For the sake of programming simplicity the current implementation is based around RPC. The basic idea is that for each channel going between the two processors (as a result of multiple processes being mapped to them), a buffer is maintained to store the data on one of the processors. The remote processor then accesses the buffer using RPC calls corresponding to the channel port operations.

One variable in the implementation is the processor on which the buffer is maintained. Four strategies are possible: always on layer 1 processor, always on layer 2 processor, the sender processor, and on the receiver processor. The choice of the strategy depends on the relative compute power of the two processors, and on the available communication bandwidth. Currently, however, all the buffers are located on the layer 2 processor because of two reasons. First, the layer 2 processor runs a real-time kernel, VxWorks in the current implementation, where all processes or tasks are inherently light-weight, and therefore cheap and efficient. Second, as mentioned earlier in

case 1, the current strategy of mapping all layer 1 processes as light-weight processes in a single UNIX process is not entirely satisfactory. Therefore putting all buffers on the layer 2 processor allows future modification of the implementation of case 1 as the RPC interface allows an inherent modularity.

Case 3: Sender and Receiver both on Layer 2 Processor

This is similar to case 1 except that the layer 2 processor uses a real-time kernel. As a result all processes or tasks are inherently light-weight and efficient, and directly scheduled by the kernel. The basic strategy is similar to that of case 1 - all processes mapped to the layer 2 processor are mapped as individual tasks scheduled by the real-time kernel. The VxWorks kernel that is currently used provides many different primitives for inter-process communication and synchronization, including a variety of semaphores, message queues and pipes. In addition, all processes share a common address space. The channel object is easily implemented by using a shared buffer together with appropriate semaphores to arbitrate access to it.

Case 4: Sender on Layer 2 Processor and Receiver on a Layer 3 Processor (and vice versa)

This is one of the most interesting cases. Like case 2 the sender and the receiver are on different processors. However, more importantly, the layer 3 processor resides on an application specific board. This means that unlike case 2 one has control over both the hardware and the software driving the communication link between the two processors. This feature can be exploited to provide explicit hardware support for efficient implementation of the channel objects.

In the architecture template the layer 2 processor, which is typically an off-the-shelf single board computer, is a master capable of initiating transactions on a backplane bus such as the VME bus. Since it is off-shelf, the layer 2 processor can not be modified in hardware. Therefore the hardware support for implementing the channels between the layer 2 processor and a layer 3 processor is integrated with the layer 3 processor. This also provides scalability as the number of layer 3

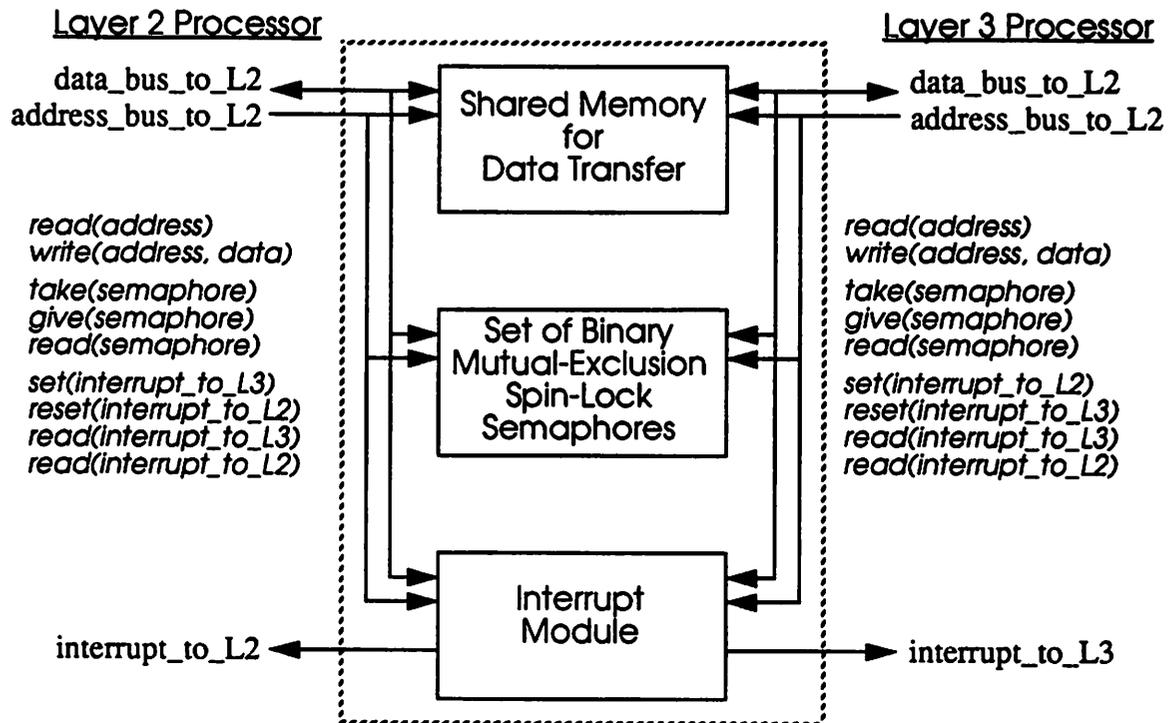


Figure 6-6 : Required Functionality of the Layer 2 ↔ Layer 3 Memory Mapped Communication and Synchronization Interface

processors changes - each layer 3 processor comes with its own support for the channel objects. This support hardware together with driver software essentially provides the lowest layer of communication and synchronization between the two processors. In order to efficiently layer the channel objects on top of it this lowest communication and synchronization layer is required to exhibit certain behavior.

Figure 6-6 shows the abstract functionality that must be provided by the communication and synchronization module in the layer 3 processor for interaction with the layer 2 processor. This black box is capable of supporting more than one channel between the two processors. It has two facets: a set of internal resources using which multiple channel objects are maintained, and a set of required operations available to both processors through accesses to memory mapped locations. The exact implementation of this black box is immaterial, and can be a mix of hardware and software, as long as it provides the following features:

-
- a. A block of shared data transfer memory resource which is used to actually buffer the data being transferred over the channels. There are two operations defined on this memory for both processors: *read(address)* and *write(address,data)*. These operations from the two processors are allowed to be executed in an overlapped fashion, and the behavior when the operations to the same address overlap is undefined.

This is a very permissive memory model (it is not even *safe*) and is implementable very cheaply. It may be implemented by a true dual-ported memory or by sharing access to a single ported memory.

- b. A memory mapped block that implements a set of spin-lock binary semaphores (busy-waiting binary semaphores based on a shared variable) for mutually exclusive access to data structures stored in the shared data transfer memory. This resource is required to support the following abstract operations on each mutual exclusion spin-lock binary semaphore: *take* a semaphore, *give* a semaphore, and *read* a semaphore. Note that these semaphores are used only for mutual exclusion between the two processors which means that the same processor takes and gives the semaphore.

There are two obvious ways of implementing this. One is to have a memory composed of *safe* binary registers and use Lamport's algorithm for mutual exclusion [Ben-Ari90], or have a memory composed of *atomic* registers and use Peterson's or Dekker's algorithm [Ben-Ari90] for mutual exclusion.

The second approach is to provide hardware assistance to simplify this process. This may be done by using a shared memory that allows atomic read-modify accesses. However the best approach probably is to have a true dual-ported memory whose locations are special mutual-exclusion elements, also called hardware semaphores. Some modern multi-port RAMs include a small, separately addressable, set of such hardware semaphores. These hardware semaphores provide three operations: read, clear, and request to set. The requests to set the hardware semaphores are internally handled in a first-come first-served fashion by a fair arbiter. Using these operations on the hardware semaphores it is trivial to implement mutual-exclusion spin-lock binary semaphores.

- c. An interrupt block which is used to send interrupts from one processor to the another. The interrupts are essentially binary flags that are set by the sender processor, reset by the receiver processor, and readable by both the processors. This allows one processor to asynchronously indicate a change in its state to the other processor. The processors themselves are responsible for handling these interrupts. This allows channel communication to be done in an event-driven fashion without busy-waiting. The following abstract operations are supported by this block: set the interrupt flag of the other processor, reset the interrupt flag to self, read interrupt flag of the other processor, and read interrupt flag to self. In addition, it is guaranteed that the set and reset operations on an interrupt flag will not overlap. This can easily be implemented by using shared boolean registers together with tri-state buffers.

Many implementations of this black-box behavior are possible. An example implementation for such a communication and synchronization interface between layer 2 and layer 3 is described in section 6.7.1. Given such a black box an efficient implementation of the channel objects is done

with the aid interrupt handlers and special I/O tasks running under the multi-tasking kernels on the two processors.

Case 5: Sender and Receiver both on the same Layer 3 Processor

This is a pure software case like cases 1 and 3. The built-in features of the particular real-time kernel running on the layer 3 processor are used to implement a channel object and the routines to access it. At present the only implementation available is that for the SPOX kernel running on the TMS320C30. Since the VDI kernel for the DSP32C module is not a multi-tasking kernel, this case is meaningless for it.

The implementation strategy under SPOX is similar to that under VxWorks in case 3. The only difference is that SPOX provides a different set of inter-process communication and synchronization primitives. The primitives in SPOX are monitors, conditions, and software interrupts. In addition, all processes share the address space. The channel object is efficiently implemented using the monitor and the condition variables together with a shared memory buffer.

Case 6: Sender and Receiver on different Layer 3 Processors

There are two sub-cases of this case., If the two layer 3 processors that are involved do not belong to the same cluster of layer 3 point-to-point links, then the channel is routed through the layer 2 processor using the technique of case 5 above.

If the two layer 3 processors involved in the communication have a direct link then explicit hardware support similar to case 4 is used to implement the channel objects - in fact the same interface block described in case 4 is used in this case too with multiple instances of it organized in a ring or a linear array.

Case 7: Sender on a Layer 3 Processor and Receiver on a Layer 4 Processor (and vice versa)

This too is an interesting case because it involves two different processors, one of which - the layer 3 processor - is a programmable processor running a real time kernel, whereas the other processor - the layer 4 processor - is a dedicated processor. The dedicated layer 4 processor can be a hardwired processor, or a software programmable processor executing just one fixed program.

The main consideration in the implementation of channels for this interface was to keep the design of dedicated hardware modules and ASICs simple. This rules out requiring these modules to have a complex microprocessor-like bus interface capable of initiating read and write transactions with an external memory. In fact most ASICs have memory mapped slave interface ports using which an external host processor initializes them and communicates with them. Such memory mapped slave interface precludes using a shared memory based hardware module for implementing the channel objects for communication with the layer 3 processor - this would require an intervening DMA like device with (or inside) every layer 4 module.

The solution adopted is to maintain the channel object in software inside the layer 3 processor. A special I/O task or/and interrupt handler (which is basically another task scheduled without the control of the kernel) is associated with each layer 4 module for moving data between it and the channel objects stored in the layer 3 processor. In effect, these special I/O tasks and interrupt handlers are a software wrapper around the raw memory mapped slave modules to make them appear to have multiple distinct ports each of which can communicate using the channels.

Figure 6-7 shows an abstract view of the implementation. The precise nature of the slave interface ports on each layer 4 module is also defined as part of the architecture template, and is presented in detail in section 6.7.1.

Case 8: Sender and Receiver both on the same Layer 4 Processor

This case is not allowed because each layer 4 processor module is a dedicated module so that only one process can be mapped to it. This is obviously true in the case of a dedicated hardware module, and is required to be true if a programmable processor module is used as a layer 4 processor.

Case 9: Sender and Receiver on different Layer 4 Processors

In this case the channel is routed through the layer 3 processors that are masters to these layer 4 processors. If the masters are different and do not belong to the same cluster, then the channel needs to be routed through the layer 2 processor.

As is obvious, this can be extremely inefficient, particularly if the sender and the receiver do not have the same master. For high-speed communication the layer 4 slaves may want to communicate directly using their own hardwired link with custom protocol. An example of this might be a link

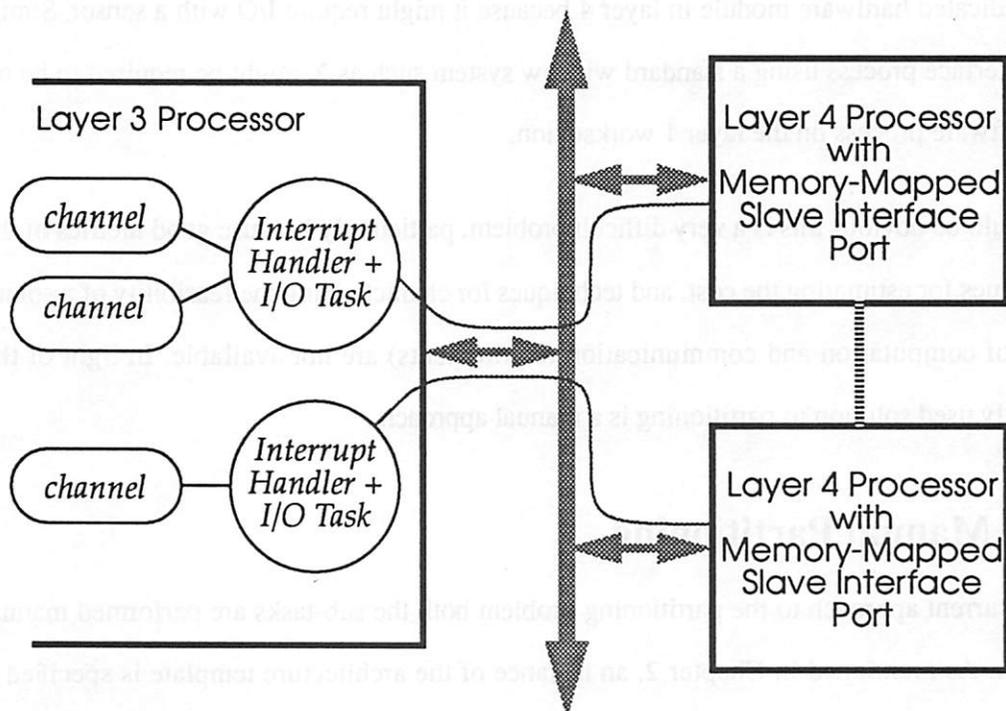


Figure 6-7 : Abstract View of the Communication and Synchronization Interface between Processing Modules on Layers 3 and 4

carrying high speed video data.

6.6 Partitioning of the System Specification

Given the system architecture template, and a description of the system as a process network, the following mutually dependent sub-tasks need to be performed during partitioning:

- a. Finding an instance of the architecture template suited for the system. This involves finding the number and types of processors in layers 3 and 4.
- b. Finding a mapping of the processes in the process network to the chosen instance of the architecture template.

These two sub-tasks have to be accomplished with the goal of a minimal cost solution that meets the computation and communication requirements. Implementation constraints also need to be taken into account in this process - certain processes in the process network may be implementable only on specific types of processors. For example, a process might be required to be implemented as a dedicated hardware module in layer 4 because it might require I/O with a sensor. Similarly, a user interface process using a standard window system such as X might be required to be mapped as a software process on the layer 1 workstation.

As should be obvious this is a very difficult problem, particularly because good metrics of the cost, techniques for estimating the cost, and techniques for characterizing the feasibility of a solution (in terms of computation and communication requirements) are not available. In light of this, the currently used solution to partitioning is a manual approach.

6.6.1 Manual Partitioning

In the current approach to the partitioning problem both the sub-tasks are performed manually by the user. As mentioned in Chapter 2, an instance of the architecture template is specified by the user in a SAIL (System Architecture Intermediate Language) file which is an intermediate form for storing the architecture information according to the layered architecture template. This file is used

as the central repository of all architecture related information, and is also used by the run-time software environment to control the hardware. It contains information about all the processors on every layer, and the processes mapped to them. The syntax of the SAIL file is lisp-like. An architecture is a lisp property list - i.e. a list of attribute value pairs - with four special attributes *:layer1*, *:layer2*, *:layer3*, and *:layer4*. The value of these attributes are possibly nil lists of processors in layers 1, 2, 3, and 4 respectively. Each processor is in turn described as a property list, with certain special attributes being required for processors at each layer. Such attributes include name of the processor (the *:name* attribute), the type of the processor (the *:type* attribute), the hosts of the processor (the list-valued attribute *:hosts*), the peers of the processor (the list-valued attribute *:peers*) etc.

An advantage of this syntax is that the policy is inherently extensible - as the tools mature more required attributes can be easily defined while retaining the same parser that is currently in use. A library of routines is available to parse a SAIL file and access various attributes. Also, as the architecture crystallizes, more and more attributes get defined. This file is also used by the run-time software system to configure the software environment.

Essentially a SAIL file views the system to be composed of two types of entities: *processes* and *processors*. The *processes* are connected arbitrarily, although the interconnections follow the channel mechanism specified by the process network model. The *processors* are connected according to the interconnection topology and interface modules allowed by the layered architecture template. The *processors* can either be dedicated hardware modules in layer 4, or software-programmable hardware modules in layers 1, 2, or 3. A layer 4 dedicated hardware module can implement only one specific process. A software-programmable module in layer 1, 2, or 3, on the other hand, can simultaneously implement multiple processes. Partitioning maps the *processes* to the *processors* taking this constraint into account.

Following is a sample SAIL file corresponding to a simple hypothetical system that is shown in

Figure 6-8. The system follows the architecture template, and has a single custom board containing two TMS320C30 processor modules running SPOX kernels in the layer 3, and no modules in layer 4. It uses a SPARC-based SUN workstation in layer 1, and a Heurikon HKV30 single board computer running VxWorks kernel in layer 2. The corresponding SAIL file, as shown below, defines each of these four processors. The two layer 3 processors are assumed to be connected to each other, and to the layer 2 processor, through dual-port RAM based communication and synchronization links. The characteristics of these communication links are described by the attribute *:ipc* of the processors in the lists of host or peers of a processor.

```

; SAIL file for the hypothetical system in Figure 6-8
(
; required header
:header (:format SAIL :comment "speech system")

; layer 1 processor is a SPARC workstation running SunOS
:layer1 (
  ( :name pajaro :type sparc_sunos )
)

; layer 2 processor is a MC68020 running VxWorks

```

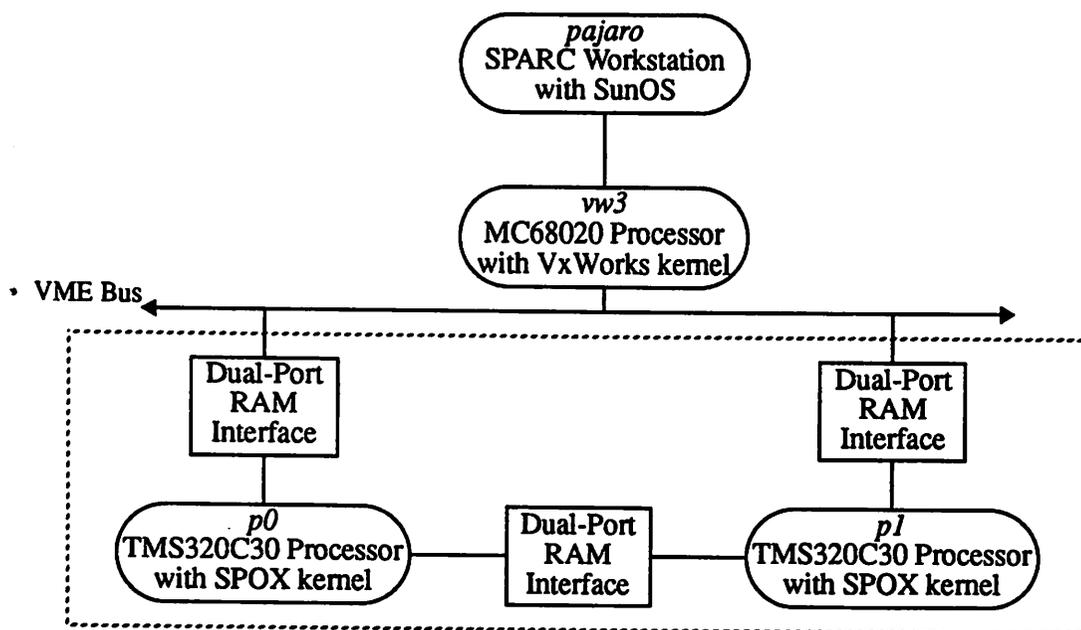


Figure 6-8 : A Hypothetical System Using the Architecture Template to Demonstrate the Syntax of a SAIL File

```

:layer2 (
  ( :name vw3 :type m68k_vw )
)

; layer 3 has two processors - both are TMS320C30 processor modules
; running SPOX kernel. The two processor modules are connected to
; each other, and to the layer 2 processor, using dual-port RAM based
; communication links
:layer3 (
  ; first processor ...
  (
    :name p0 :type c30_spox
    :vmeaddr_base 0 :vmeaddr_type 24 ; its VME Bus attributes
    :srambase 512K :sramsize 256K ; its local memory
    ; description of its host - the layer 2 processor
    :hosts (
      (
        :name vw3
        ; description of the interface to the host
        :ipc (:dpram_num 1
              :local_view (:dpramport left :intrnum 0)
              :remote_view (:dpramport right :intrnum 5)
            )
      )
    )
  )
  ; description of its peers - other layer 3 processors that
  ; are connected to it
  :peers (
    (
      :name p1
      ; description of the interface to the peer
      :ipc (:dpram_num 1
            :local_view (:dpramport left :intrnum 1)
          )
    )
  )
)
; a similar description for the second layer 3 processor ...
(
  :name p1 :type c30_spox
  :vmeaddr_base (* 12 2KW) :vmeaddr_type 24
  :srambase 512K :sramsize 256K
  :hosts (
    (
      :name vw3
      :ipc (:dpram_num 1
            :local_view (:dpramport left :intrnum 0)
            :remote_view (:dpramport right :intrnum 5)
          )
    )
  )
  :peers (
    (
      :name p0
      :ipc (:dpram_num 1
            :local_view (:dpramport right :intrnum 1)
          )
    )
  )
)

```

```
    )
  )
)
; there are no layer 4 modules in this simple example
:layer4 ()
)
```

6.6.2 A Strategy for Automated Partitioning

The manual approach used currently leaves the partitioning problem unsolved. A more systematic approach to partitioning is desirable. Even if it proves impractical to automate the entire partitioning process, it may be feasible to automatically map the process network given a particular instance of the architecture template, or to just evaluate the cost of a partitioning specified by the designer.

Greedy approaches similar that used by Vulcan-II [Gupta92a] for a single processor architecture may work even for the complicated architecture template used by us. The somewhat simpler problem of mapping the process network given an instance of the architecture template can be viewed as a problem of binding processes to processors. One greedy approach would be to start by assigning all processes to the lowest (highest number) layer that they can be implemented on. This lowest layer is determined by the following strategy - a process is first assigned to a dedicated hardware module in layer 4 if it can at all be implemented in hardware. Otherwise it is mapped to a layer 3 processor, the layer 2 processor, and the layer 1 processor in that order. This would give, in some senses, the fastest solution. Next, processes can be migrated one by one in a greedy fashion to a higher layer as long as the computation and I/O limitations are not exceeded. Thus compute-bound processes in layer 4 can migrate to one of the upper layers as software processes. This greedy process can be continued until no improvement in board area takes place.

6.6.3 Role of Process Network Transformation in Partitioning

In the discussion so far it was assumed that the granularity of the process network description is unchanged during the partitioning process. In effect the functional decomposition implied by the process network description is retained during implementation. This can often result in an inefficient solution, or may result in no solution being found. For example, processes running on a programmable processor have associated costs of context switch and inter-process communication. Therefore coalescing smaller processes that are communicating with each other into a single process may improve performance. Similarly, part of a large process may be carved out as a separate process that is run on a separate processor.

This notion of decomposition and coalescing of processes in the process network can be generalized and formalized into the transformation of a process network such that the resulting process network has the same functionality, and the constituent processes communicate using FIFO channels with the same restricted set of protocols as were discussed in Chapter 5.

An example of such a process network transformation is shown in Figure 6-9 where two processes in a client-server relationship are coalesced into one. This is essentially equivalent to converting a remote procedure call into a local one. Several other such transformations can be formulated. For example, another possible transformation is to replace a synchronous (unbuffered) communication channel by an asynchronous (buffered) channel as the latter is often cheaper to implement.

Currently no mechanism is provided in the framework to do these transformations on the process network, whether manually or automatically. The reason is that a pre-requisite for doing such transformations is that the precise computation being done by each process in the process network also needs to be modeled. So far we have been concentrating solely on formalizing the black-box behavior, or the communication behavior, of the processes with a view towards automating the interfacing between the processes. Unfortunately we still do not have a unified formal representation of the computation being done inside each process in the process network. The

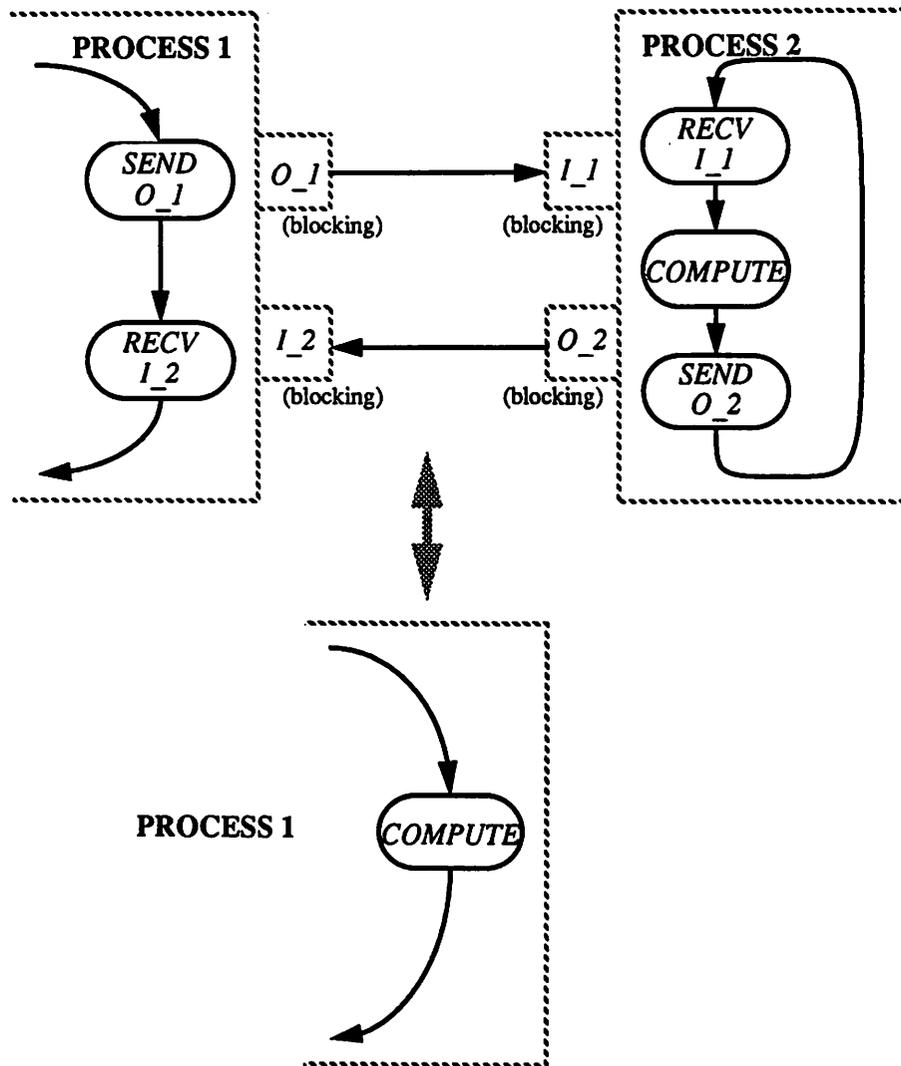


Figure 6-9 : Coalescing a Server Process into a Client Process, and vice versa

process network only represents the communication pattern, and the processes themselves are specified in terms of their implementations as C code (in case of software implementations), or as a structural or a behavioral description of a hardware implementation. This is really a reflection of the fact that no currently available representation was found to be powerful enough to handle all the computation models needed for the systems of interest in this thesis. The search for a unified representation that can handle all computation models, and from which either hardware or software can be generated, has been left for future investigation.

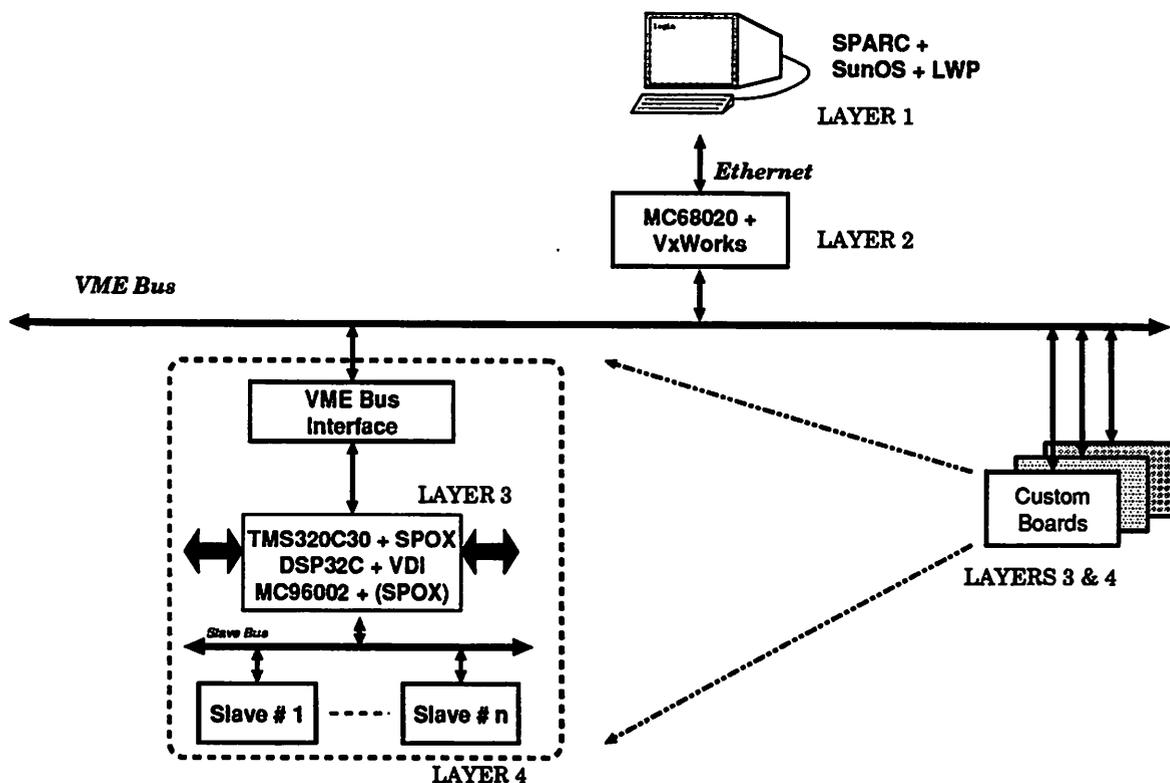


Figure 6-10 : Currently Available Implementation Choices for the System Architecture Template

6.7 Implementation of the Architecture Template

In order to be able to implement instances of the architecture template within the CAD framework several hardware modules and software libraries have been developed that allow complete systems to be prototyped according to the architecture template. This section describes the current capabilities of this effort.

Figure 6-10 shows the currently feasible implementations of the architecture template. The only allowed layer 1 processor is currently a SUN SPARCStation. At layer 2 the supported processor modules are the HKV2F and HKV30 single board computers both of them running the VxWorks real-time multi-tasking kernel. At layer 3, which resides on the custom boards, the supported processor modules are a versatile TMS320C30 based module (discussed in Chapter 3), and

simpler modules based on DSP32C and MC96002 processors. The TMS320C30 module is managed by a customized version of the SPOX real-time multi-tasking kernel, while the DSP32C module uses VDI, a real-time foreground-background kernel. At present no real-time kernel is supported on the MC96002 module. At layer 4 a variety of dedicated hardware modules are available in the library, including A/D, D/A, optical communication module, RS232C module, optical encoder module for position and velocity sensing. A simple DSP32C based processor module is also available for use as a layer 4 processor. In addition, the module generators described in Chapter 4 can be used to develop application-specific modules for layer 4 in a short time.

The current physical interconnection medium between layer 1 and layer 2 processor modules is the ethernet. Similarly, between layer 2 and the layer 3 processor modules the currently supported interconnect medium is the VME bus. Effort is underway to support SBUS as an alternate choice at this level. The interconnection between a layer 3 processor module and the layer 4 modules slave to it is described in terms of a custom bus protocol which is described in detail in the following subsection.

6.7.1 Implementation Restrictions and Guidelines

As a result of the experience with the development of the current suite of hardware and software modules for supporting the architecture template, several guidelines or restrictions have also been developed with which the new modules need to adhere.

Layer 1 Processor Modules

The only restriction on the layer 1 processor module is that it should be a processor with a 32-bit data type that supports UNIX and X, and be able to communicate with a layer 2 processor using TCP/IP and SUN's RPC protocols. The precise physical medium of communication is in theory

immaterial - it can be ethernet, FDDI, VME, SBUS etc. - although the current layer 2 modules support only ethernet.

Layer 2 Processor Modules

A layer 2 processor module should be a 32-bit processor running VxWorks, which is currently the only supported real-time kernel at layer 2. The processor module should be able to read and write 32-bit memory locations shared with each layer 3 processor over a backplane bus such as the VME bus. In addition, it should be able to handle interrupts from the layer 3 processors.

Layer 3 Processor Modules

A layer 3 processor should be able to do 32-bit read and write to a memory shared with the layer 2 processor, and also do 32-bit read and write to layer 4 slave processors that are memory mapped to it. It should be able to handle interrupts generated by the layer 4 processors.

Interface Between Layer 3 and Layer 2 Processor Modules

A layer 3 processor module appears as a memory mapped slave to the layer 2 module across a backplane bus. The layer 3 processor module is expected to provide certain basic host interface functionality over this memory mapped interface. This functionality, which is just a concrete realization of the abstract black box behavior of the layer 2 \leftrightarrow 3 interface described in section 6.5.3, includes:

- a. Shared memory area that is visible to both the layer 2 and layer 3 processors. It should be possible for the layer 3 processor to execute a program stored in this shared memory when the processor is reset.
 - b. Shared memory locations that provide atomic read and write accesses for software implementation of spin-lock semaphores using Peterson's algorithm. Alternatively, and preferably, hardware semaphores may be provided.
 - c. Ability to halt and start the layer 3 processor from the layer 2 processor by writing to a bit in a memory location.
 - d. Mail-box interrupt to the layer 3 processor, i.e. a memory location which generates an interrupt to the layer 3 processor on being accessed by the layer 2 processor.
-

-
- e. Mail-box interrupt to the layer 2 processor, i.e. a memory location which generates an interrupt to the layer 2 processor on being accessed by the layer 3 processor.
 - f. Ability to monitor the status of mail-box interrupts from both the sender, and the receiver.
 - g. Ability to reset the mail-box interrupt by the receiver.
 - h. Ability to send and receive at least one single-bit flag in each direction via memory mapped locations.

A reference implementation of such an interface is provided in the library that can be used with most 32-bit processors and most backplane busses. The reference implementation, shown in Figure 6-11, is based around using a 32-bit wide true dual-port RAM which also provides special memory-mapped hardware semaphore locations. The mail-box interrupts, flags, and reset functionality is provided by a pair of PLDs. These PLDs, called *v2tigen* and *t2vigen* in the library, can be used in isolation to implement other such interface modules with different types of shared memory. For example, a single-ported SRAM may be used together with external arbiter to provide this capability. Some processors, such as the DSP32C and the MC96002, have special ports through which a host is allowed atomic access to the processor memory. These ports can be used to implement a cheap though slower shared memory.

Interface Between Layer 4 and Layer 3 Processor Modules

A layer 4 processor module appears as a memory mapped slave device to the layer 3 processor module across a slave bus whose protocol is defined as part of the architecture template. The layer 3 and layer 4 processor modules are expected to adhere to this protocol. This requires the following basic functionality:

- a. Every layer 4 module can have one or more *ports* through which it connects to the slave interface bus of the layer 3 processor modules. Each such slave port can have signals from the following set of signals:
 - a data bus $D[0:N_D-1]$ where $0 \leq N_D \leq 32$
 - an address bus $A[0:N_A-1]$ where $0 \leq N_A \leq 13$
 - active low output enable signals $OEN[0:N_{OEN}-1]$ where $N_{OEN} \geq 0$
 - active low write enable signals $WEN[0:N_{WEN}-1]$ where $N_{WEN} \geq 0$
-

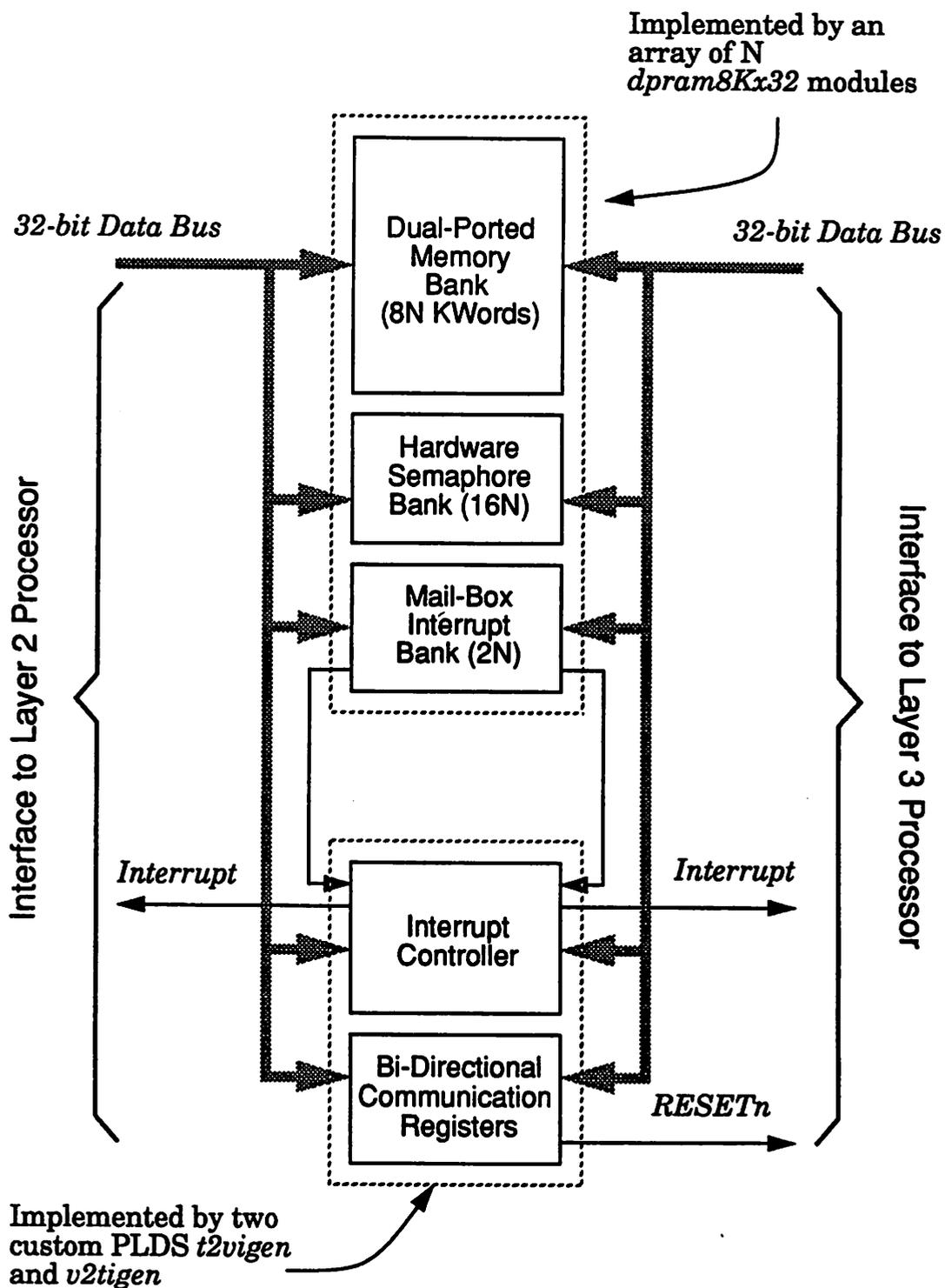


Figure 6-11 : Reference Implementation of the Interface between Layer 2 and Layer 3

-
- active low chip select signals $CSN[0:N_{CSN}-1]$ where $N_{CSN} \geq 0$
 - active low buffer read signal $RDBUFN[0:N_{RDBUFN}-1]$ where $N_{RDBUFN} \geq 0$
 - active low register write signal $WRREGN[0:N_{WRREGN}-1]$ where $N_{WRREGN} \geq 0$
 - active low reset signal $RESETN$
 - active low access completion signal $RDYN$
 - active low interrupt $INTRN$

In the above list the choice of address bus width to be a maximum of 13 may seem rather arbitrary and restrictive, particularly for applications such as vision and image processing that deal with large quantities of data. The number was decided based on what can be efficiently supported in hardware across commonly available processors while also being sufficient for typical applications. Further more, image and vision applications that need to access large quantities of data, for example a pixel from a single frame, access them according to some address-generation algorithm, as opposed to true random-access. Therefore such cases can easily be handled even with a limited address bus by integrating an address-generation unit into the layer 4 slave module. Hardware module generation techniques of Chapter 4 can often be used to automatically generate such address-generation units from a high-level specification.

- b. Every layer 3 processor module must have a slave bus controller block using which it can generate the control signals of the types listed above for all the slaves attached to it. The ALOHA interface synthesis system may be used to synthesize such a block. A parameterized slave bus controller module is also provided in the library which can be used with most processors. Given suitable parameter values it can generate the OEN , WEN , CSN , $RDBUFN$, and $WRREGN$ signal for the layer 4 slaves connected to the processor.

The basic idea behind this interface is that a slave port is viewed as a window through which the layer 3 processor can access memory locations inside the layer 4 slave module. The interface has been defined with a view to simplify the implementation. The signals are defined to cover three commonly encountered types of memory locations. SRAM-like multi-word memory is accessed through the appropriate combination of CSN , OEN and WEN . Positive edge-triggered registers as commonly found in PLDs and TTL devices are written through $WRREGN$. Tri-state buffers often

used to monitor status signals are read using the RDBUFN signal. CSN, WRREGN and RDBUFN can be viewed as enable signals associated with blocks of memory with different timing characteristics. Each signal of the type CSN, RDBUFN, and WRREGN is characterized by the size of memory block enabled by it, and the access delay for locations in that memory block, provided the delay is fixed. This delay information is by the slave bus controlled to provide a sufficient number of wait states. If the access delay to the memory block associated with a CSN, WRREGN, or RDBUFN signal is not fixed then the slave is required to generate a handshake acknowledge on the RDYN output. In addition to the signals for memory access, a slave port must contain a signal RESETN to reset it. The signal is guaranteed to be low on power-up and on system initialization. Finally, the slave port can also contain an interrupt output to be handled by the layer 3 processor.

6.8 Summary

The issues in architecture generation at the system level were presented in this chapter, together with a description of the architecture template-based approach adopted in this thesis. The problem is not yet well defined, so that the current approach is still based on manual partitioning and mapping. Instead the research was concentrated on developing a system architecture model that encompasses dedicated as well as software programmable hardware in a formalized fashion. In addition, communication and synchronization mechanisms for efficient implementation of the FIFO channels and the port protocols used in the process network description have been investigated.



CHAPTER 7

USING SIERA FOR DESIGNING SYSTEMS

This chapter describes the different ways in which the tools and libraries in SIERA that were described in the previous chapters can be used for designing systems.

7.1 The Three Entry Points to SIERA

Although SIERA provides a complete vertical path for system design, there are several different ways of using the tools, libraries, and software that is part of SIERA. As is evident from the discussion so far, different facets of this design framework are automated to different extents. Lower level hardware module generation utilities are mostly automated and quite general purpose in applicability. The software module generation is not automatic, but provides a disciplined framework in which to write modular software together with extensive run-time support software. The high level specification of a system as a process network, and implementation according to the architecture template, are not general purpose, although they do have wide applicability as demonstrated by two system examples in the following chapters.

In light of the varying degree to which different parts of this framework are automated and are of general applicability, it is inevitable that different system designs use the framework to different extents. Moreover, the loosely coupled modular organization of the framework makes it possible to only use those tools and libraries that are useful. On the basis of experience with several board-level system designs the following three distinct ways of fruitfully using this framework have been found:

- a. As a physical board-level design environment for custom boards with dedicated hardware, and whose hardware organization is completely known. A good example of using SIERA in such a fashion is the design of the robot peripheral board mentioned in Chapter 8 (next chapter). SIERA allows changes in the hardware organization to be incorporated easily, and also allows rapid exploration of alternate hardware organizations.
- b. As a design environment for concurrent development of hardware and software for custom boards that have both dedicated hardware modules as well as embedded software programmable processors. The architecture of the board is arbitrary, but pre-determined. No example boards exist yet that were designed using this approach.
- c. As a complete system design environment with support for high-level specification and simulation, architecture generation using manual partitioning onto an architecture template, hardware module generation, and software organization. The robot system presented in the next chapter is a good example of this approach.

In the following three sections a description of the design flow using SIERA in each of these three roles is presented.

7.2 Designing a Custom Board with Dedicated Hardware

In this case SIERA is like a silicon assembler for ASICs, except that it is targeted at board level hardware. Using techniques described in Chapter 3 boards with architectures customized to a specific application can be generated rapidly.

Even in this limited role SIERA provides a design environment which is more sophisticated than that provided by commercial board placement and routing tools. In commercial tools, such as the ones from Racal-Redac or OrCAD, the board design is done schematically at a low level of

abstraction as a hierarchical netlist of individual chips. In contrast, in SIERA there is an extensive library of parameterized sub-system modules, module generators, and tools for a variety of placement styles. This enables quick design iteration in case of changes in the design architecture.

Figure 7-1 shows the design flow in this case. The user expresses the design as a hierarchical netlist composed of parameterized sub-systems from the central library or personal libraries, individual chips, and behaviorally specified modules. The netlist is usually expressed by the SDL language, although a less versatile schematic interface is also available. Next the design manager DMoct is run on the root of the design. DMoct traverses the design hierarchy, and runs appropriate structure-processors and layout-generators as needed. Typically the root of the design has a layout-generator attached to it that generates input for a foreign router in the form of a flat and fully placed net-list of individual board-level components. The next step is that of routing for which a foreign router is used. The output of this process is a set of GERBER files describing the geometry of each layer of the board. A GERBER previewing tool can convert these files into an OCT physical view. After a satisfactory set of files is available, they are sent for board fabrication.

7.3 Designing a Custom Board with Dedicated Hardware and Software Programmable Processors

This is quite similar to the previous case except that the custom board may also use parameterized software-programmable processor modules from the library. This allows processor modules to be embedded together with dedicated hardware in a custom fashion. Together with bus interface modules, such as the VME or SBUS interface, this also allows the ability to rapidly prototype relatively general purpose single-board computers.

The process of the design of the board itself is the same as in Figure 7-1. However, in addition, a variety of software modules are available for use with programmable processor modules that were used in the board design. As described in Chapter 4, this includes multi-tasking kernels, library of

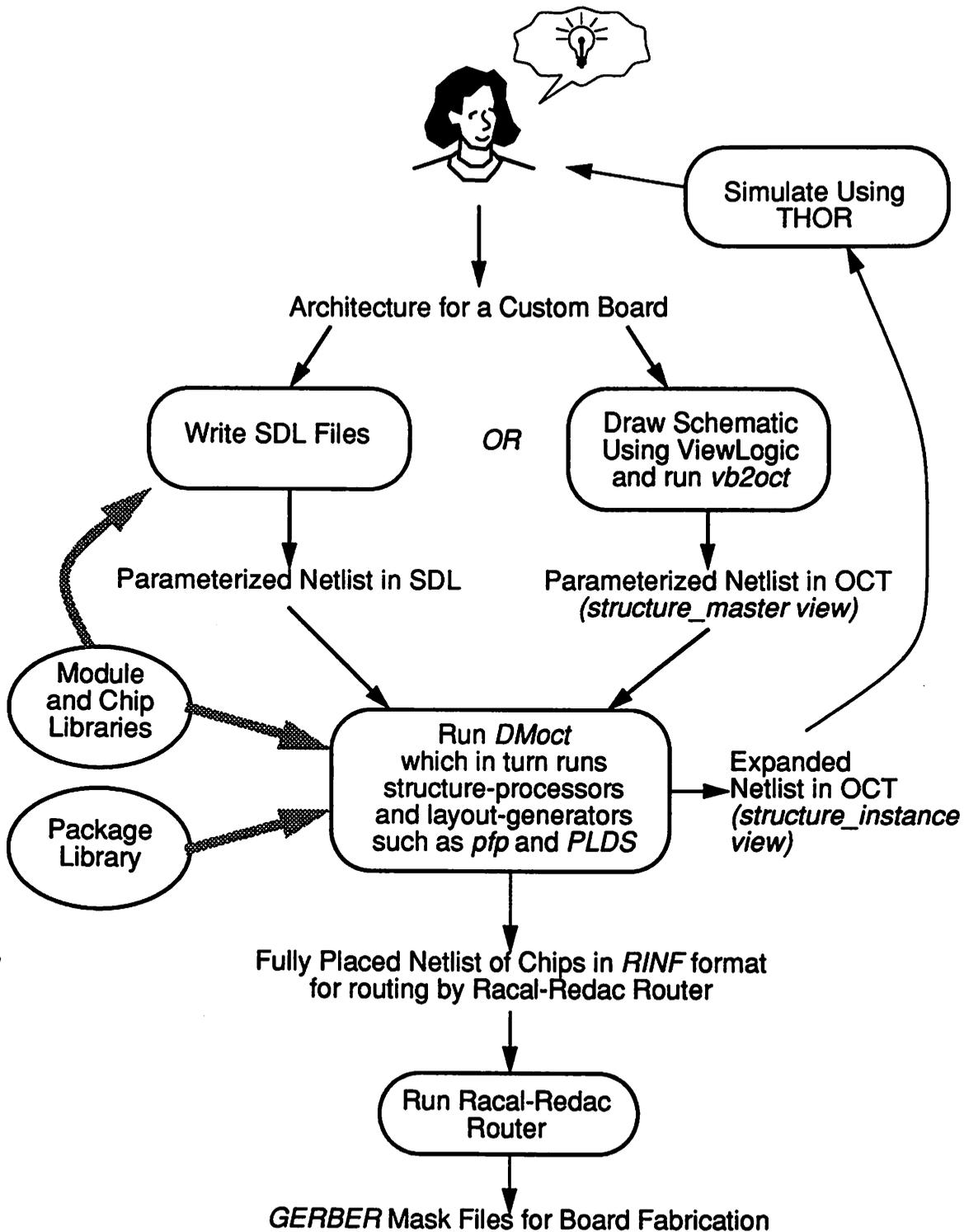


Figure 7-1 : Currently Implemented Design Flow for a Custom Board with Dedicated Architecture without using the System Architecture Template

inter-process communication and synchronization, object-file loaders, and other run-time utilities. However, since the board architecture can be arbitrary, and need not follow the architecture template described in Chapter 6, the user will still have to provide the interconnection between these software modules and hardware modules. Nevertheless, the availability of these reusable software modules, kernels, and tools simplifies the task.

7.4 Designing a System According to the Architecture Template

In this mode SIERA attempts to provide a complete vertical path for designing systems. As is evident from the discussion in the previous chapters, the entire path is not yet automated and the approach used for some higher level problems are not universally applicable to all types of systems. Still, support is provided for all the phases of the system design cycle, as shown in Figure 7-2.

The first step in the system design process is to specify and model it at a high level using the process network model of computation. In order to arrive at such a description the system functionality is decomposed into coarse-grained concurrent entities. These concurrent entities become the processes in the process network description, and communicate using the channel paradigm discussed in Chapter 5. Some of these concurrent entities may correspond to the environment in which the system operates and are not meant to be generated. Thus, for example, the robotic mechanism being controlled by a robot control system can be viewed as one such environmental process.

Next the behavior of each of these concurrent entities - or processes - is described for simulation purposes. Currently this can be done using two approaches. The first is to describe the process behavior in VHDL using the VHDL package described in Chapter 5. An alternative is to use the process domain that has been added to Ptolemy by a co-researcher [Lee91] and describe the

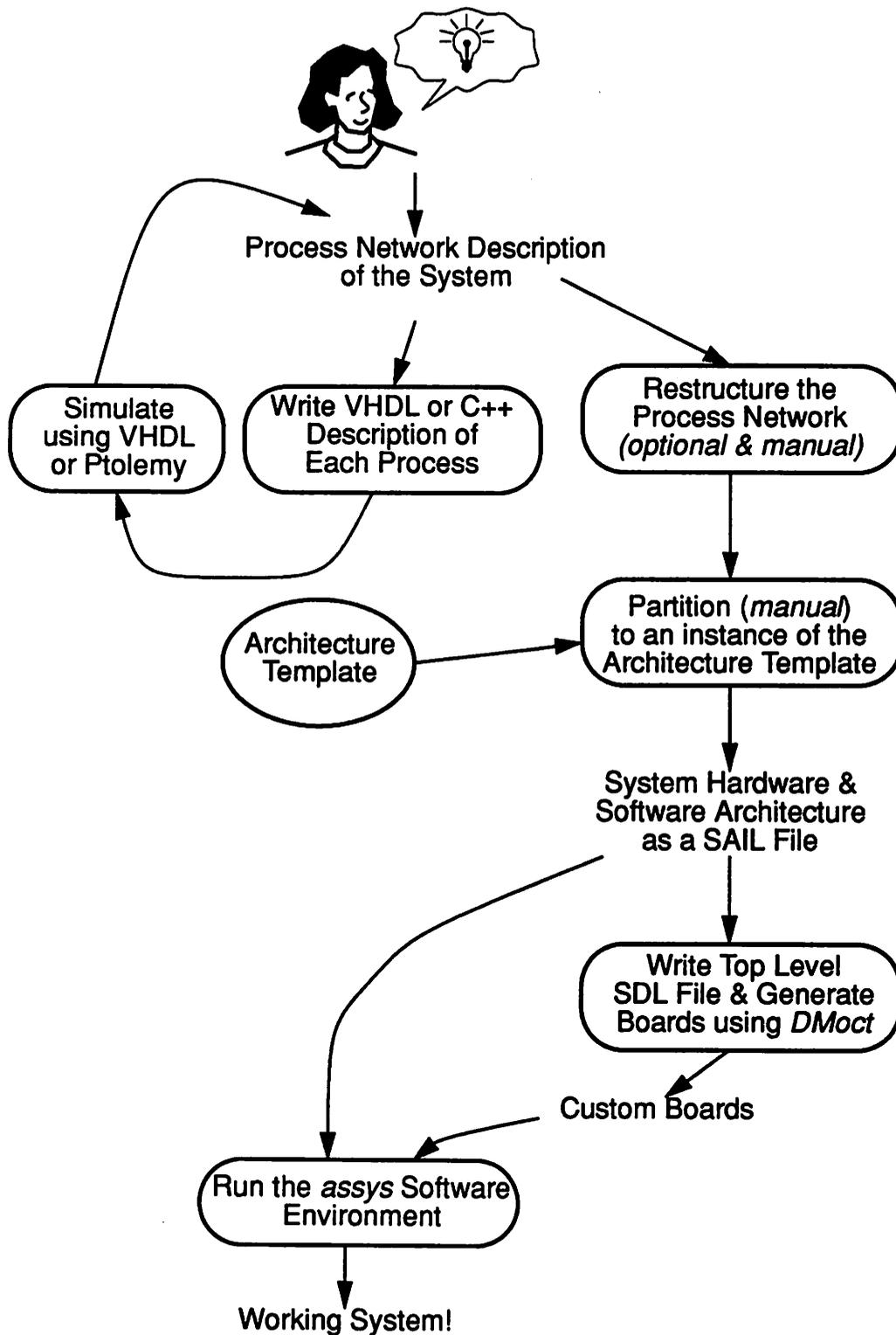


Figure 7-2 : Using SIERA for the Entire Top-Down Design of a System

processes using C++. Both these implementations require the process behavior to be strictly sequential. The VHDL approach allows the environmental processes to be modelled using the stochastic modelling package and the continuous time modelling package described in Chapter 5. In the case of Ptolemy the environmental processes can be modelled using any of the other domains of Ptolemy, thus providing a mixed simulation capability as soon as the process domain is fully integrated into the Ptolemy framework. The simulation at this level allows the functionality of the system to be checked without any detailed timing constraints.

After simulation the process network needs to be mapped to an instance of the architecture template described in Chapter 6. However, as mentioned there, it might be desirable to restructure the process network first by using transformations such as coalescing a server process into a client process and sizing the channel buffer depths. The process of mapping itself involves choosing an instance of the architecture template and then annotating each process in the process network graph to be either a dedicated hardware process or a software process mapped to one of the processors on the layered architecture template. This process is completely manual, and involves taking two things into account. First, some of the processes may have implementation restrictions. For example, a process that involves communicating with sensors or actuators will have to be mapped as a layer 4 slave. On the other hand a user-interface process involving X window graphics will require to be run on the layer 1 workstation. As described in Chapter 6, a reasonable strategy for doing the partitioning is to first map all un-constrained processes to the lowest layer possible, and then greedily move them to higher layers as long as the number of layer 4 and layer 3 modules continue to decrease (and hence the cost of the custom boards).

The result of this partitioning is the hardware and software architecture of the system encoded as a SAIL (System Architecture Intermediate Language) file. As elaborated in Chapters 2 and 6, a SAIL file describes an instance of the architecture template - it contains information about all the processors on every layer, and the processes mapped to them.

The SAIL file contains enough information about the processor modules being used in the various layers and the parameters of their interconnect links to enable generation of the top-level SDL files for each of the custom boards in the system. This step remains to be automated, and currently the designer has to write the top-level SDL file for each board. This, however, is a straight-forward procedure as the overall structure of all the boards is the same except for the modules in layer 4. This is accomplished by the fact that three building blocks for boards following the architecture template are already provided in the module library.

The top-level SDL file for a custom board that follows the architecture template looks like the example in Figure 7-3. The three building blocks are: *brdGen*, *ipcnw*, and *slave_manager*. The *brdGen* block is a generic board that contains multiple independent processor modules attached to a bus. The number, type, and configuration of each of the processors can be specified using parameters. The bus connects these processor modules to a layer 2 processor. At present the only supported bus is the VME bus though, of course, the modular approach being taken would allow other interface modules to be designed and integrated. The supported processor modules in this configuration are the TMS320C30 based module, and the DSP32C based module. The MC96002 module can also easily be incorporated into the *brdGen* module, although this has not been done yet. Corresponding to each processor in the *brdGen* module there is a main bus, and a slave bus that comes out. The main busses are connected using the *ipcnw* module. This module allows sets of processors to be connected in a ring or a linear array - the number of processors is a parameter. Finally, the *slave_manager* module is a parameterized slave bus interface module that generates the necessary signals to interface layer 4 slave modules conforming to the interface structure described in Chapter 6.

Using these three building blocks in a fixed configuration all a user has to do is to write the top level SDL file that contains the layer 4 slaves. This procedure is being automated so that the top level SDL file is generated automatically from the SAIL file. Note, however, that the SDL files for

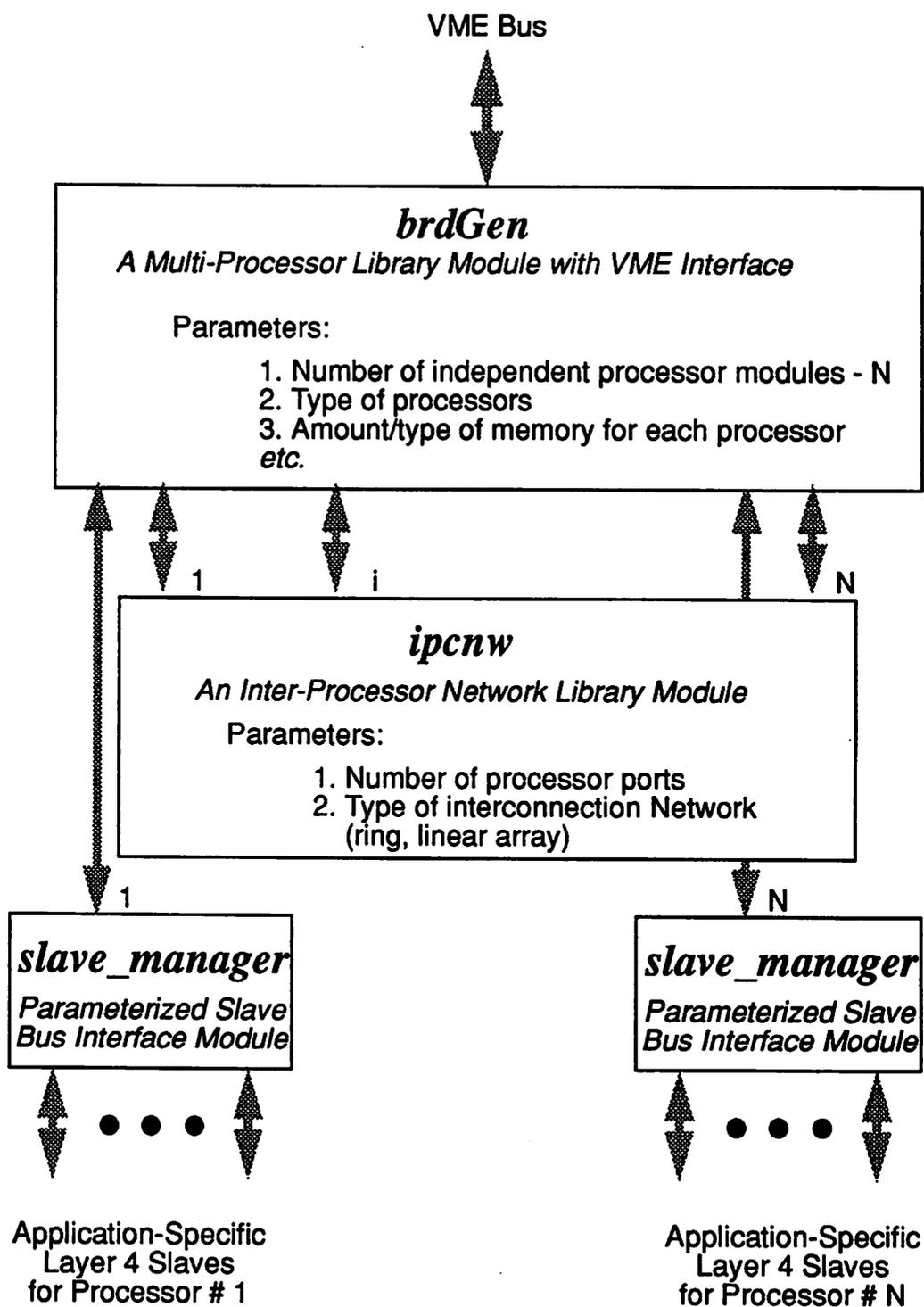


Figure 7-3 : Top-Level SDL File for a Custom Board in a System following the Architecture Template

the layer 4 slaves still need to come from the library, or be provided by the user. Once the top level SDL file for each board has been generated, the boards themselves can be generated by running *DMoct*.

The same SAIL file that is used to create the SDL files is also used by the run-time software described in Chapter 4. Currently the run-time software assumes a SPARC processor on layer 1, and a MC68020 running the VxWorks OS on layer 2. A special function *assSysInitialize(SAIL_file_name)* is executed on the layer 2 processor. This function reads the SAIL file, recursively bootstraps and initializes all the slave processor, and starts the various I/O servers. The application software modules are then loaded and executed by using the command *assL3ProcAppLoadAndRun(processor_name, object_file)*.

7.5 Summary

This chapter described the use of SIERA in three distinct fashions ranging from a board with known, dedicated architecture to the design of a complete system following an architecture template for the organization of hardware and software.

CHAPTER 8

MULTI-SENSORY ROBOT CONTROL SYSTEM

This chapter presents the first example of a system designed using the framework presented in this thesis. The system is a multi-sensory robot control system, and was the primary driver application of much of the research in this thesis. The entire design cycle of this system as well as the resulting implementation are described here. Besides being an exercise of a computer-aided system design methodology, the resulting control system has many interesting architectural features that are worth presenting on their own merit, like the extensive use of dedicated hardware in the form of custom boards and custom chips. This is largely a result of the ease of prototyping and fast turnaround time offered by the framework described in earlier chapters for the design of such custom hardware and the associated software drivers.

8.1 System Requirements

The first task in the design of any system is to decide what is required from the system - it is only then that one can specify the system functionality. Sometimes the requirements themselves are not

known, but in most cases the requirements are constrained by the environment surrounding the system, and the goal for its design.

The requirements for the multi-sensory robot control system were defined primarily by its environment (robot, sensors, and user), and by a suggestion from my advisor that I should “try to make the robot do something neat ... like play ping-pong!”. Based on these a set of requirements for the robot controller were defined, and are enumerated below.

8.1.1 Environmental Constraints

The environment of the robot controller consists of two things - the robot arm + sensors, and the human user (or operator) - each of which places certain constraints on what the robot controller is required to do. The robot controller can be viewed to have two ports. Through one port a user commands the robot to do a certain task using an interactive or a language-based interface. Through the other port the controller receives the sensor readings and drives the actuators (motors and relays) in the robot.

Robot Arm + Sensors

We view the robot together with the sensors as one black box which communicates with the controller over a port. However, the black box is really composed of several distinct and concurrently operating physical entities, and the port communication is itself made up of several logically distinct channels of communication.

There are three distinct classes of entities that are present in the black box. First is the mechanical device or the *Robotic Mechanism* which is being used to carry out the task of physically manipulating the environment (like assembling a car or scraping paint etc.). The robot controller drives this mechanical assembly through *Actuators*, like motors and relays, which form the second class of entities, The third class of entities are *Sensors* which provide information about the internal state of the robotic mechanism, and about its interaction with the physical world.

The Robotic Mechanism:

The robot mechanism is a six rotational degree-of-freedom commercial robot arm from Panasonic (model PanaRobo VI). Kinematically this robot is similar to the classic PUMA robot. Each of the six joints are geared joints, and in addition the joints have brakes. A custom made open-shut two-pronged pneumatic gripper mechanism is attached at the end of the robot.

The Actuators:

The geared joints of the Panasonic arm are driven by d. c. motors that are housed in the arm itself. A servo-amplifier box that came with the robot (and is therefore considered part of the robot arm) contains the low-level electronics to drive the motors. The amplifiers are H-bridge amplifiers whose inputs have to be generated by the controller. Each set of inputs is composed of four signals which take on values of 0 V (asserted) or 15 V (disasserted). These inputs together specify the voltage that needs to be applied - one input is asserted when the voltage is negative, the second is asserted when the voltage is positive, the third is disasserted if the voltage is positive and is a pulse width modulated waveform indicating the voltage magnitude if the voltage is negative, and the fourth input is similar to the third one except that it is active when the voltage is positive.

The brakes to the joints are driven by relays that are also part of the servo-amplifier box. The input to each of the relay is also a one bit active-low signal where low is 0 V and high is 15 V. As currently configured, applying the brakes to any one of the joints results in brakes being applied to all the joint.

The air-flow to the pneumatic gripper is also controlled by a relay which is attached to the arm itself together with necessary drive electronics. The input to the gripper drive electronics is a single-bit TTL level signal that needs to be provided by the controller.

The Sensors:

Following are the sensors that are available:

a. Joint Position Sensors

These are highly accurate optical encoders connected to the motor end of the gear drive train at each joint. They allow measurement of the relative position of each joint by outputting TTL level quadrature waveforms (a pair of square waves phase shifted by +90 degrees or -90 degrees - where the phase shift indicates the direction of rotation and a certain fixed number of pulses correspond to one rotation). In addition, a single pulse is generated on a third signal at the same (but unknown) position in every rotation of the motor.

b. Joint Origin Sensors

These are relatively inaccurate potentiometer based sensors that generate a TTL level signal that is low on one side of some mechanically fixed absolute origin of the joint, and high on the other side. The transitions of this signal (modulo gear backlash) are accurate to within one motor rotation.

c. Joint Limit Sensors

The same potentiometer sensor that generates the absolute origin indicator is also used to generate two TTL level signals that are asserted low when the joint position exceeds some mechanically determined limits on the two ends of its range of motion (< 360 degrees).

d. Force/Torque Wrist

This is a strain gauge based wrist that is attached between the gripper and the robot arm to measure the contact forces and torques between the robot arm and the physical objects being manipulated by the gripper. This is a commercial sensor that comes with its own signal processing box. The controller communicates with this box instead of directly to the strain gauges in the wrist.

The interface has two separate ports. One is a RS232 serial link using which the entire functionality of the force/torque sensor is accessible. Configuration parameters have to be down-loaded at the beginning, and then the sensor box supplies a force-torque vector at some sample rate (100 Hz maximum). There is a second parallel port that allows data to be transmitted from the sensor box at a higher rate (400 Hz maximum).

e. Proximity Sensor

This is a sensor mounted on the gripper that signals when the gripper is sufficiently close to some object. This is done by measuring the reflection of a LED from the object surface. The output is a TTL level single bit signal that is asserted when an object is in close proximity to the sensor.

f. (Optional) Vision Sensor

A vision system capable of tracking objects in a 2D plane using Radon Transform based algorithms is available as the result of a co-researcher's work [Baringer91]. The system is based on a set of identical custom boards that consist of four DSP32C processor modules each, and conform to the layered architecture template and the low-level communication and synchronization interface specification discussed in Chapter 6. A software process running on layer 2 collects data from processes running under VDI on each of the DSP32C modules, and generates the position of an object in real-time as the object moves through the field of view. A new position vector (X, Y, ROT) is generated every video frame (30 ms).

User Interface

The user interfaces with the controller through a UNIX workstation running X. This allows a mix of interactive keyboard or mouse or even joy-stick based control, as well as a language based interface. The user interface is based on RPC following the strategy described in Chapter 4. Basically the user-interface is composed of one or more UNIX processes that generate RPC calls for a server process that needs to be part of the controller. The RPC calls are generated as a result of interactive user actions or by a program written by him. The high-level (task-level) planning is considered to be part of the user interface.

8.1.2 Goal

The suggestion given by my advisor was scaled down and it was decided to ignore the high level task planning needed to make the robot do intelligent things like play ping-pong. Instead the goal was reformulated to concentrate on the lower level control of the robot. The goal for the system was defined to be:

Control in real-time the position of the Panasonic robot arm and the forces applied by it to the physical world incorporating data from the joint encoders, force-torque wrist, and the proximity sensor. In addition, it should be capable of using the tracking data generated by the 2-D vision sub-system so that it too can be incorporated at a later stage.

8.2 Algorithms

There are two distinct phases of operation of the robot: initialization and control. During initialization the robot is calibrated with respect to its environment, and during control the robot is made to follow a desired position and force trajectory.

Although initialization is a one time process, it is complicated enough to deserve special attention,

and even dedicated hardware to simplify it. The problem arises due to two reasons: use of relative position sensors, and the lack of precise knowledge about the robot kinematic parameters. The optical encoders used to measure the joint angles accurately are inherently relative position sensors - they allow position to be measured with respect to some position, and do not have an absolute origin. The potentiometer based joint origin sensors, on the other hand, measure the absolute origin, but are inaccurate. During initialization an elaborate process is used for each joint to establish an accurate absolute origin. This involves moving each joint slowly in one direction while monitoring the absolute origin sensor and the index pulse of the optical encoder. The index pulse immediately before or after the absolute origin can be used to establish an accurate absolute origin. This still leaves unknown the position of the robot relative to the physical world. Techniques based on external sensors, such as a camera, may be used to accomplish this automatically. A simpler technique is used in this system - a table of joint origin offsets, in terms of encoder counts, has been built up by manual calibration of the robot arm with respect to its environment.

The control phase of operation requires continuous monitoring of the sensors, and generation of inputs for the actuators (joint motors and pneumatic gripper relay). Robotic control algorithms have become far more complicated than the simple PD joint controllers of the past. State of the art systems incorporate both force and position controllers (sometimes both running at the same time [Craig79][Khatib87]). Position control is generally used when the links of the robot form an open kinematic chain, i.e. the robot does not touch anything. When, however, the robot eventually contacts an object, the robot forms a closed kinematic chain until the object is lifted or freed.

For tasks in which the robot forms a closed kinematic chain, force control is almost a necessity [Whitney]. For example, the robotic task of scraping paint from a surface cannot be easily performed using a PID controller. Any error in the system would be catastrophic for a PID controller. If the surface were too far from the robot-held tool, there would be no contact force. On the other hand, if the surface were too close, the integral term of the controller would apply the

absolute maximum force against the surface. The ability to specify as a control input the desired force against the surface solves this problem.

Many types of force controllers have been proposed [Whitney]. The type chosen for this system is a form of force control called impedance control [Hogan], in which the force applied by the robot end-effector is proportional to the displacement from its goal position. This system is similar to a spring, where $F = K\Delta X$. The basic advantage of this force control strategy over others is that it displays some position stability (away from any singularity). Further, this type of controller is well suited to tasks such as peg-in-hole insertions using a remote center of compliance as proposed by [Whitney82]. Figure 8-1 shows an extremely simplified process network view of the system.

Linearization of the robot control algorithm is based on the calculation of complicated inertial, Coriolis, and gravitational terms, and these terms have to be updated at a reasonably high frequency. There are also numerous frame transformations and unit conversions that must be performed inside the control loop. At a higher level, the trajectory control process must continuously monitor the proximity sensor, and update the controller inputs. Certainly, the hardware architecture must be designed to facilitate all these high bandwidth tasks.

8.3 System Architecture

The resulting architecture of the robot control system is shown in Figure 8-2. It is an instantiation of the layered architecture template described in Chapter 6. The top layer 1 is based around a SUN workstation, and software processes for user interface and high-level path planning are mapped to it. The next layer 2 is based around a MC68020 based single-board computer running the VxWorks kernel, and communicates with the workstation across an ethernet. It in turn coordinates two VME bus slave custom boards that form layers 3 and 4. One is the robot controller board, and the other a vision sensing board. The controller board in turn communicates with a custom peripheral board using a fiber-optic link. This peripheral board interfaces with the joint motors and

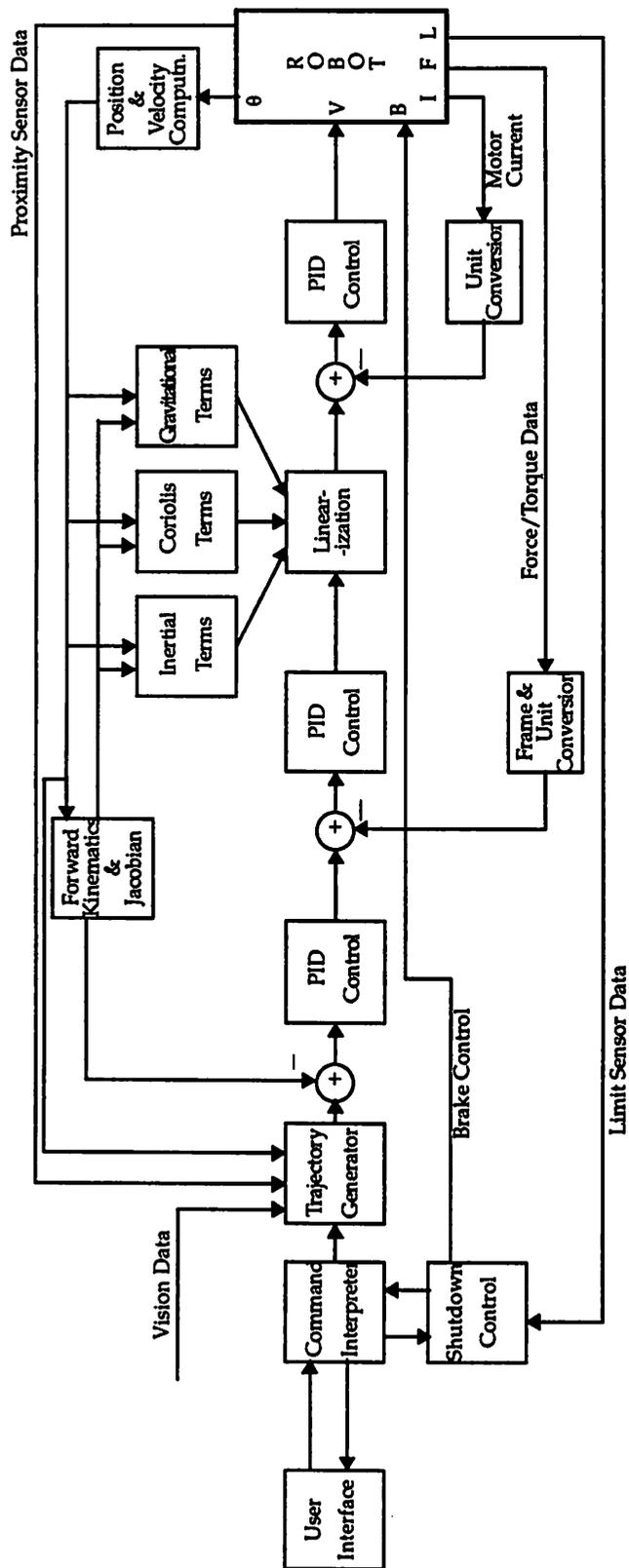


Figure 8-1 : Functional Decomposition of the Robot Control System

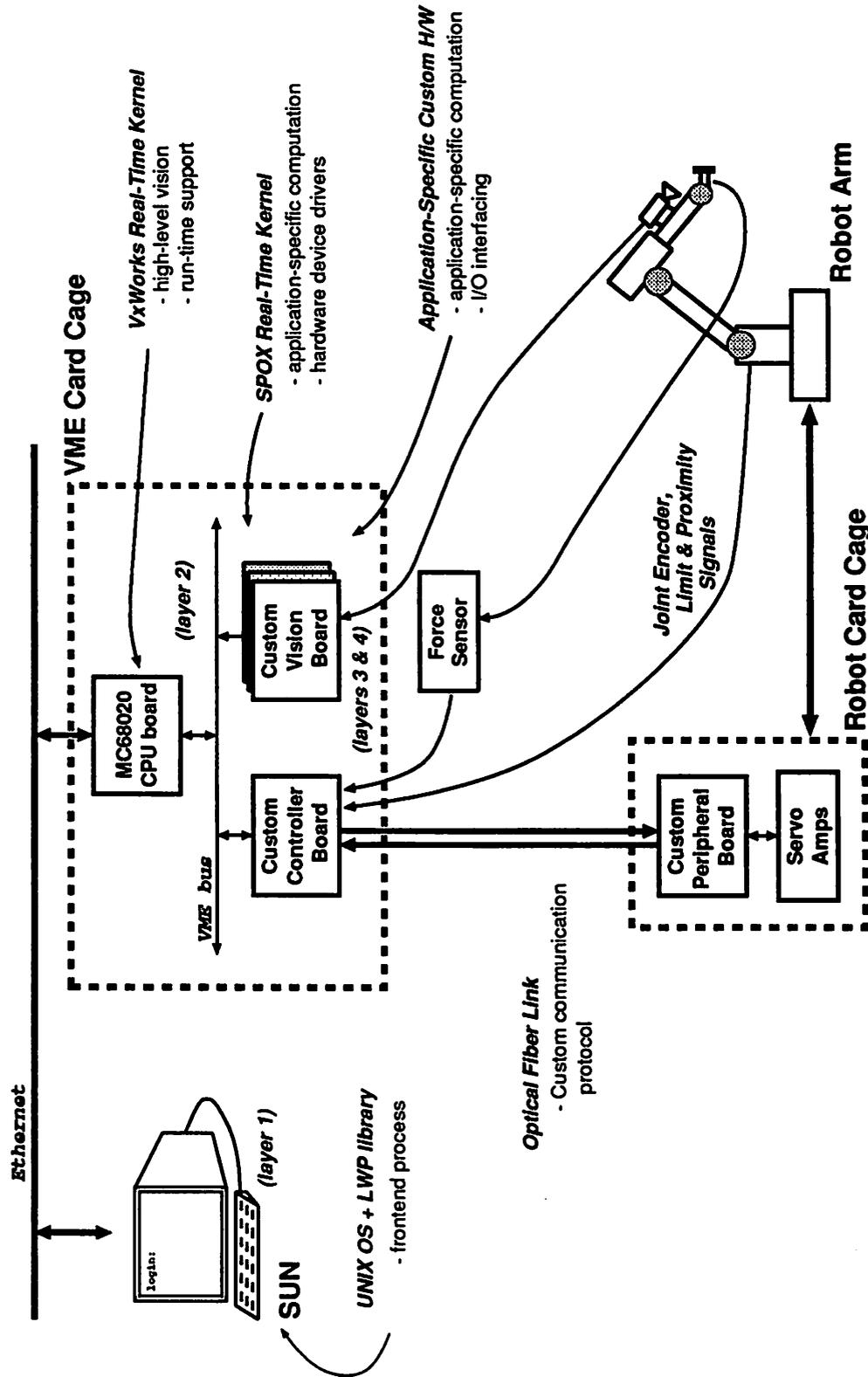


Figure 8-2 : Architecture of the Multi-Sensory Robot Control System following the Architecture Template of Chapter 6

current sensors in the robot arm. The controller board also communicates with force and position sensors, and the vision board communicates with a camera.

This custom architecture is in sharp contrast to architectures based around a general-purpose homogeneous shared-memory MIMD multi-processor with off-shelf I/O boards that are typical of most state-of-the-art commercial as well as research robot control systems [Chen86][Ish-Shalom88][Narasimhan88][LYMPH]. The custom architecture approach offers much improved performance in a more compact package. The tasks in a robot control system not only have high real-time computation requirements, but also need extensive specialized I/O capabilities. This restricts the controllers based around general purpose machines with limited I/O capabilities to simple control mechanisms, or to non-real-time algorithm test benches at best.

8.4 Hardware Organization

The two custom boards were generated using the board-level module generation tools and libraries described earlier. The parameterized library modules and the module generation tools allow variations of these boards to be generated in a short time with computation and I/O resources tailored for different algorithms, sensors, or robots.

The controller and the peripheral custom boards are described in the following sections. The use of the various hardware and software module libraries and tools is also illustrated.

8.4.1 Controller Board

This is a custom board to which the processes related to the control law, position, velocity, force, and current sensing, and motor drive are mapped. It spans layers 3 and 4 of the architecture template discussed in Chapter 6. As shown in Figure 8-3, it has two processor modules in layer 3. Each of these is based around 33 MHz TMS320C30, a powerful floating-point signal processor. Both the processor modules are instantiated with 1 Mbyte of fast zero wait state SRAM, and use

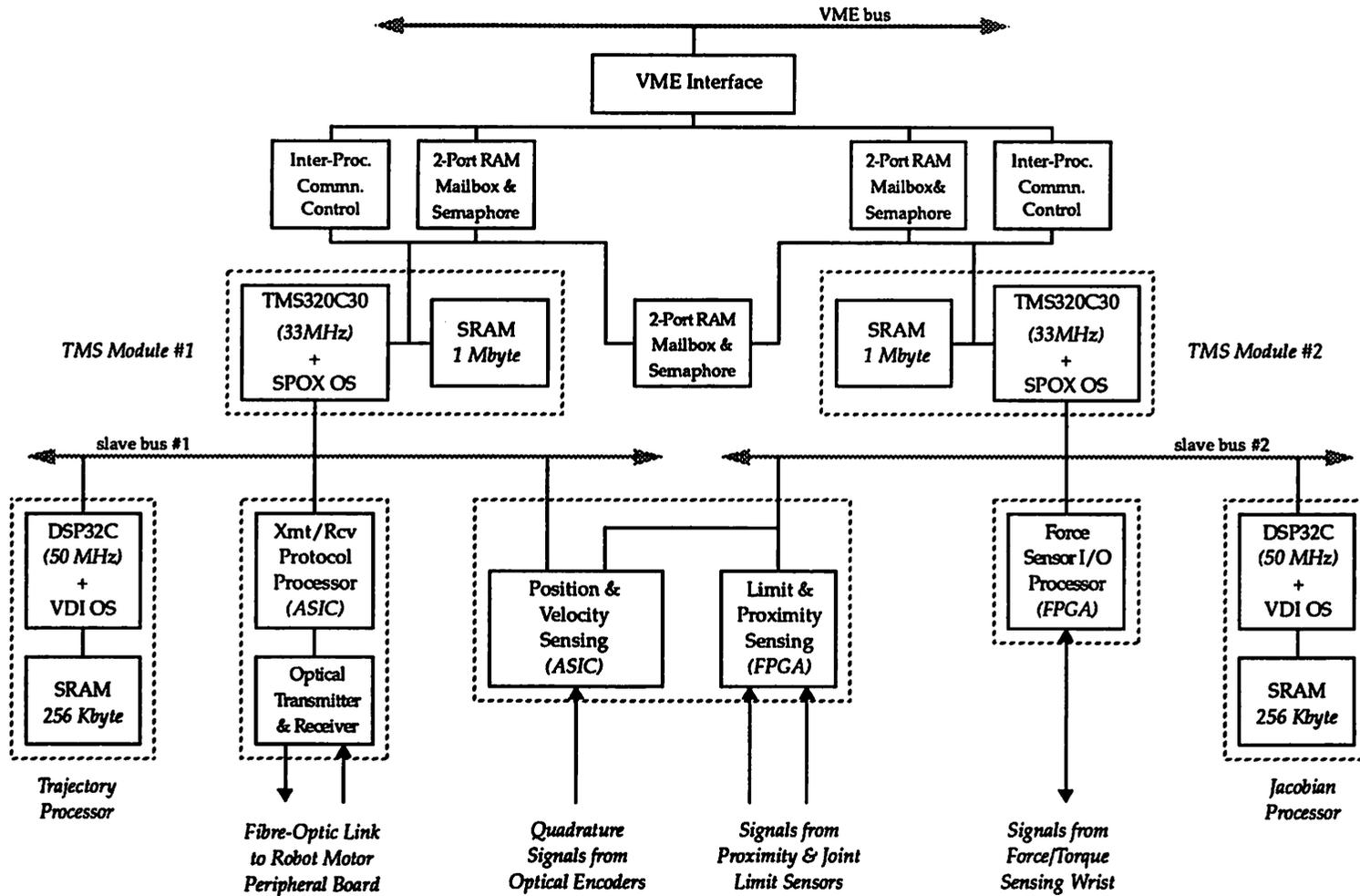


Figure 8-3 : Architecture of the Robot Controller Board

true dual-ported RAM together with hardware semaphores and mail-box interrupts to communicate with the layer 2 processor across the VME bus. In addition, the CPU chip itself provides on-chip resources like dual-bank SRAM, instruction cache, timers, serial ports, and DMA controller. The two processor modules communicate with each other using another instance of the dual-ported RAM with hardware semaphores and mail-box interrupts. In addition, an efficient single-instruction barrier synchronization facility is provided.

The two TMS processing modules are distinguished by the kind of slave modules that they have. These slave modules reflect the specific requirements of the robot control tasks.

TMS processor module #1 has three slaves. First is a powerful programmable processor module based around a 50 MHz DSP32C, and has 256Kbyte of zero wait state SRAM. It is a dedicated processor to which the trajectory calculation process is mapped. The second slave is a fiber-optic based communication link that utilizes a custom ASIC to implement the protocol processors. This link is connected to the robot peripheral board at the other end, and serves as an interface to the robot motors. It is used to apply specific voltages or torques to the motor, to sense the motor currents, and to apply brakes to the robot. A sophisticated custom protocol is used that utilizes caching to make the optical-link transparent to the TMS processor - the A/D, D/A and other resources on the remote peripheral board appear to be local. The ASIC that implements the protocol processing employs an asynchronous design methodology, and was synthesized using the interface synthesis tools. The third slave attached to TMS processor module #1 is a position-velocity sensing module. It is based around a pair of custom ASICs that take in signals from the position sensors in each of the robot arm joints, does noise filtering, and then calculates the relative and absolute positions and instantaneous velocity of each joint. Hardware support for the initial origining and calibration of the robot arm is also provided. The ASICs for this module were automatically generated from a parameterized structural description.

The second processor module, TMS processor module #2, has three slaves. The first slave is a

DSP32C based processor module, identical to the one attached to TMS processor module #1. This, however, is dedicated to the process calculating the Forward Kinematics and the Jacobian of the robot arm. The second slave is the force sensor module that interacts with a strain-gauge based force-torque sensing wrist. The third module is the position-velocity sensing module, which is in fact shared with the TMS processor module #1. However, besides the position and velocity information, the slave module also provides data from the joint limit sensors and the proximity sensor to this processor module.

The architecture of the board demonstrates the ease with which two different types of processor modules - TMS320C30 and DSP32 - can be integrated. This heterogeneity is a result of choosing the processor module most appropriate for the task. Although TMS320C30 is a faster processor with a higher bus I/O bandwidth than DSP32C, the latter has a flexible host-interface that makes it possible to interface it to a master processor with little glue logic, and therefore less board area. Therefore the TMS320C30 processor module was used in layer 3, but the DSP32C module was considered a better choice as a layer 4 slave module. The compute power of the DSP32C processor module (25 MFLOPS peak) is sufficient to do the desired calculations on the controller board at a fast throughput - for example, the jacobian and inverse kinematics calculations for the six joint PanaRobo robot arm can be done at better than a 1 KHz sample rate, which is more than sufficient for this system.

The hierarchy of busses provided by the layered architecture template results in a much improved I/O throughput than would be possible in the case of a single bus architecture, such as the one used by Vulcan-II system mentioned in Chapter 1. This is because the total available bandwidth, assuming good partitioning of system functionality, is the sum of the bandwidths of the individual busses. In the controller board each of the two layer-4 slave busses can support up to 16.5 million 32-bit transactions per second for a peak I/O throughput of 66 Mbytes/sec. Of course, the sustained rate depends on the I/O bandwidth of the individual slaves as well as the computation being done by the layer 3 processor module between transactions. Similarly, the point-to-point link

between the two layer 3 TMS320C30 modules can operate at a peak rate of about 11 million 32-bit transactions per second for a peak I/O throughput of 44 Mbytes/sec. The actual rate that is obtained also depends on the synchronization overhead and the computation being done - in practice rates as high as 10 Mbytes/sec have been obtained. The VME slave interface is capable of I/O rates in excess of 24 Mbytes/second. However, in practice the rate is severely limited by the layer 2 processor that is the VME master. For example, with the Heurikon HKV30 processor as the layer 2 processor, peak I/O rates do not exceed 10 Mbytes/sec. When the synchronization overhead is taken into account, the actual achievable rate through the channels between layers 2 and 3 is less than 2-3 Mbytes/second. The total system I/O bandwidth, which is the sum of these individual bandwidths, is much higher than what would be possible with a single bus as in the Vulcan-II model. This fact, together with the high computation capabilities of the processors (33 MFLOPS for each TMS320C30, 25 MFLOPS for each DSP32C - for a total of 116 MFLOPS), results in a dedicated multi-processor system with high compute power and a balanced I/O system.

The board was fabricated as a 9U VME slave board, and is fully functional. Figure 8-4 shows a photograph of the board. Primarily as a result of the module-generators available, and the sub-system level library, this complex 500+ component 12" x 14" board had a design cycle of less than two months. Further, since it follows the architecture template, the system software was configured for it in very little time. Table 8-1 summarizes the salient features of the board:

Dimensions	14" x 14" 9U-VME Board
Number of Layers	12
Number of Components	310 parts + 330 bypass capacitors
Design Time	2 man-months, including ASICs & some library parts
Amount of SDL Code	1375 lines of SDL code

Table 8-1 : Main Features of the Robot Controller Board

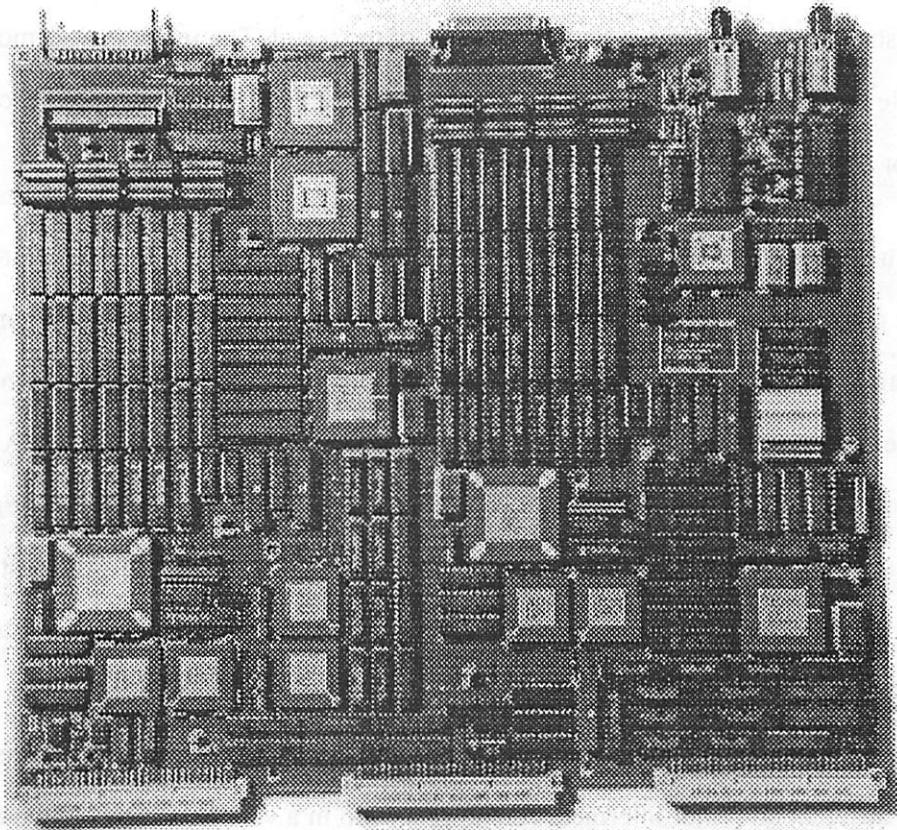


Figure 8-4 : Photograph of the Robot Controller Board

8.4.2 Peripheral Board

This board is really part of the robot in that it provides interface to the robot joint motors and brakes. Its task is to receive voltage or current values from the controller board and apply them to the robot motors, to sense the motor current, and to apply the brakes in case of stalls or when commanded. It is a mixed analog-digital board. For reasons of electrical noise isolation, and to avoid mechanical problems with thick cables for carrying signals to the controller, a duplex optical fiber with a custom communication protocol is used as the system-level interconnect between this board and the controller board. The optical fibers carry the data serially and the high bandwidth (125 Mbits/sec) allows protocols with low latency which is important in this case because the

optical fibers are part of the controller feedback loop.

Figure 8-5 shows a block diagram and photograph of the board. The intent is to demonstrate that the board-level hardware module generation tools provided by our framework can be used fruitfully for a custom board that is not part of the layered architecture template.

The board uses A/D, D/A, and optical communication modules from the sub-system module library. The protocol processors for implementing the custom packet communication protocol over the fiber-optic links are synthesized using the ALOHA tool and implemented using two Actel 10x0 FPGAs. The six-channel digital pulse-width modulators are also implemented using Actel FPGAs from a mixed structural and combinational behavioral description. Only a small analog portion of the board (opamp based filters) had to be custom designed for this board - the rest was either automatically generated or instantiated from the reusable parameterized sub-system module library. As a result of this level of automation the entire development cycle from input description to the working board was less than two months. Similar boards for robots with different numbers of joints or different amplifier interfaces can be generated in a very short time. Table 8-2 lists the salient features of the board:

Dimensions	14" x 10"
Number of Layers	8 (4 signal + 4 split-planes)
Number of Components	464 parts + 110 bypass capacitors (Note: the board has many discrete parts for its analog, A/D, and D/A functions.)
Design Time	3 man-months, including ACTEL FPGAs etc.
Amount of SDL Code	2000 lines approximately

Table 8-2 : Main Features of the Robot Peripheral Board

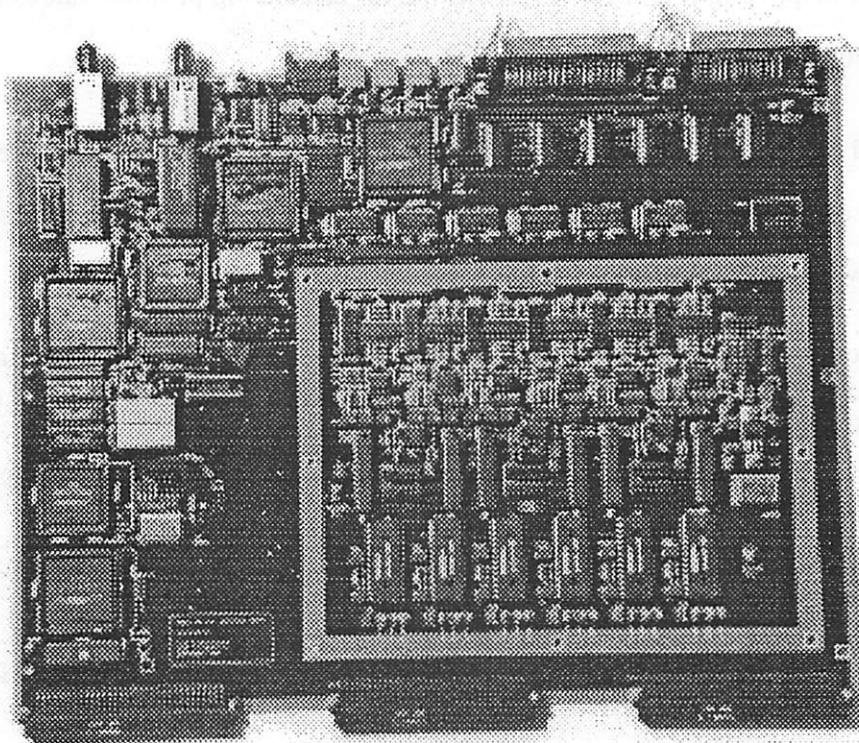
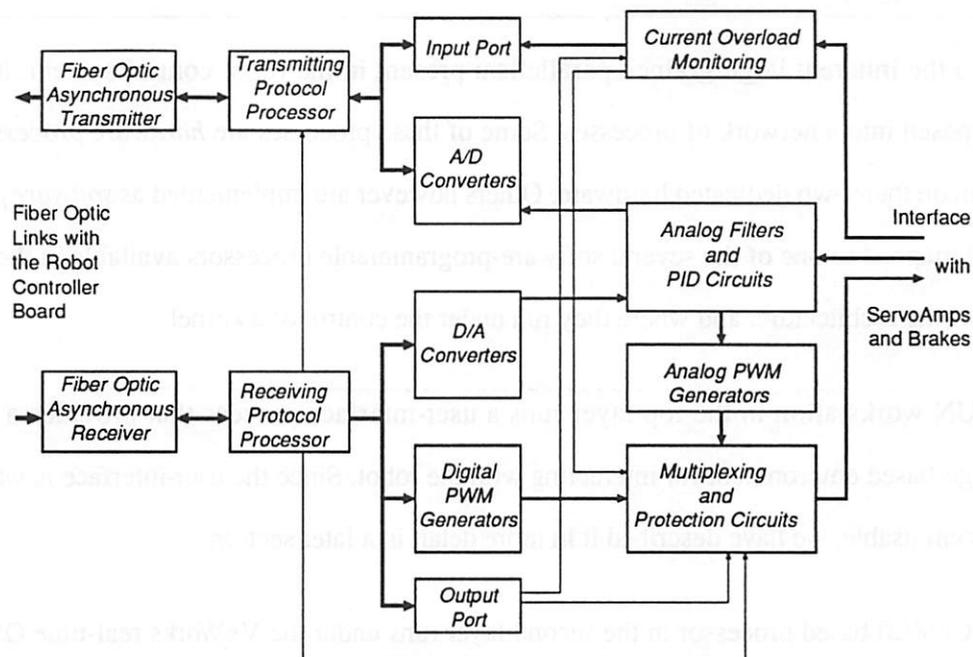


Figure 8-5 : Block Diagram and Photograph of the Custom Robot Peripheral Board

8.5 Software Organization

Due to the inherent large-grained parallelism present in the robot control system, it is easily decomposed into a network of processes. Some of these processes are *hardware processes* in that they run on their own dedicated hardware. Others however are implemented as *software processes*, and get mapped to one of the several software-programmable processors available in the top three layers of the architecture, and where they run under the control of a kernel.

The SUN workstation in the top layer runs a user-interface process that provides a powerful language-based environment for interacting with the robot. Since the user-interface is what makes the system usable, we have described it in more detail in a later section.

The MC68020 based processor in the second layer runs under the VxWorks real-time OS, and the only process mapped to it is the one responsible for calculating position values from the data obtained from the vision board.

As described earlier, there are two layer 3 processing modules on the robot controller board. Both of these processor modules run under the SPOX real-time kernel. The TMS processor module #1 has the following processes mapped to it:

- a. server process that interprets commands from the user-interface process on layer 1. These include sophisticated trajectory control commands that automatically change the goal-frame or commanded joint rotations incrementally over time etc.
- b. process for computing the desired end-effector force to apply, by comparing the tool frame of the robot arm with the goal frame ($F = K\Delta X$).
- c. process adjusting the amount of voltage applied to the robot motors using the current sensor data obtained from the robot peripheral board.

The DSP32C slave module attached to this processor runs the trajectory calculation process.

The TMS processor module #2 has the following processes mapped to it:

- a. process to find the current tool-frame through forward kinematics.
 - b. process to compute the error in applied force/torque using the data from the sensor.
-

-
- c. process to compute the desired joint torque using the force/torque error information, the inertial, Coriolis, and gravitational terms, as well as the Jacobian transform.

The DSP32C slave module attached to this processor runs the Forward Kinematics and the Jacobian calculation process.

In addition to the user processes mentioned above, all the processing modules also run system processes in the background to provide run-time I/O services, and to handle data-routing.

8.6 User's View of the Robot System

The user interacts with a user-interface process that is mapped to a SUN workstation on layer 1. This communicates with the processes running on the controller boards using channels with the single-board computer on layer 2 acting as a gateway. This user-interface process provides two services. First it provides a console for processor modules on the custom boards, allowing programs to do C standard I/O calls. This is meant primarily for debugging purposes. The second and more important functionality is to provide an interactive robot programming environment.

The interactive environment is based on top of a C interpreter together with a library of robot specific C routines to provide an interface to the lower-levels of the control system. These routines can either be called from the interpreter command line, or can be embedded in other code making full use of the C language. These routines use the communication channels to make the lower layers of the robot control system appear as a *robot motion server*. These routines allow the user to update control law parameters, switch between impedance control and position control modes, calibrate the robot vis-a-vis its environment, read sensor data, update the goal state of the robot (position, force, and gripper status), specify motion trajectory etc.

This user interface has been used to perform several robot tasks, including testing algorithms for doing peg-in-hole insertion [Whitney82].

8.7 Summary

Although the robot is still far from being able to “play ping-pong” as my advisor wanted, it nevertheless provides a set-up for robotics experiments in force-position control algorithms while operating in real-time. More importantly though this system provided a very nice driver application for the CAD framework described earlier in this thesis. Although the system proved too complex to design completely automatically, or even to simulate in its entirety, it nevertheless helped in identifying the problems and defining the goals for this research.

CHAPTER 9

GRAMMAR SUBSYSTEM FOR SPEECH RECOGNITION

This second example is on the design of part of a speech recognition system, specifically the front-end and grammar processing parts of a Hidden-Markov Model based speech recognition system. This example is somewhat unusual because the overall system architecture had already been decided, and the hardware as well as software for the rest of the system implemented when it was decided to re-implement the front-end and grammar processing sub-systems as custom boards following the architecture template paradigm in order to improve the overall system performance. As a result there were many implementation constraints, particularly on the hardware implementation. Due to these constraints the architecture template was not used in its entirety - only layers 3 and 4 of the template were used in this sub-system.

9.1 Global Functionality of the Speech System

A detailed description of the speech recognition system can be found in [Stolzle91], here only a brief description is being presented. Figure 9-1 shown a block diagram depicting the system functionality. The system is targeted at real-time recognition of a large vocabulary (60,000 words),

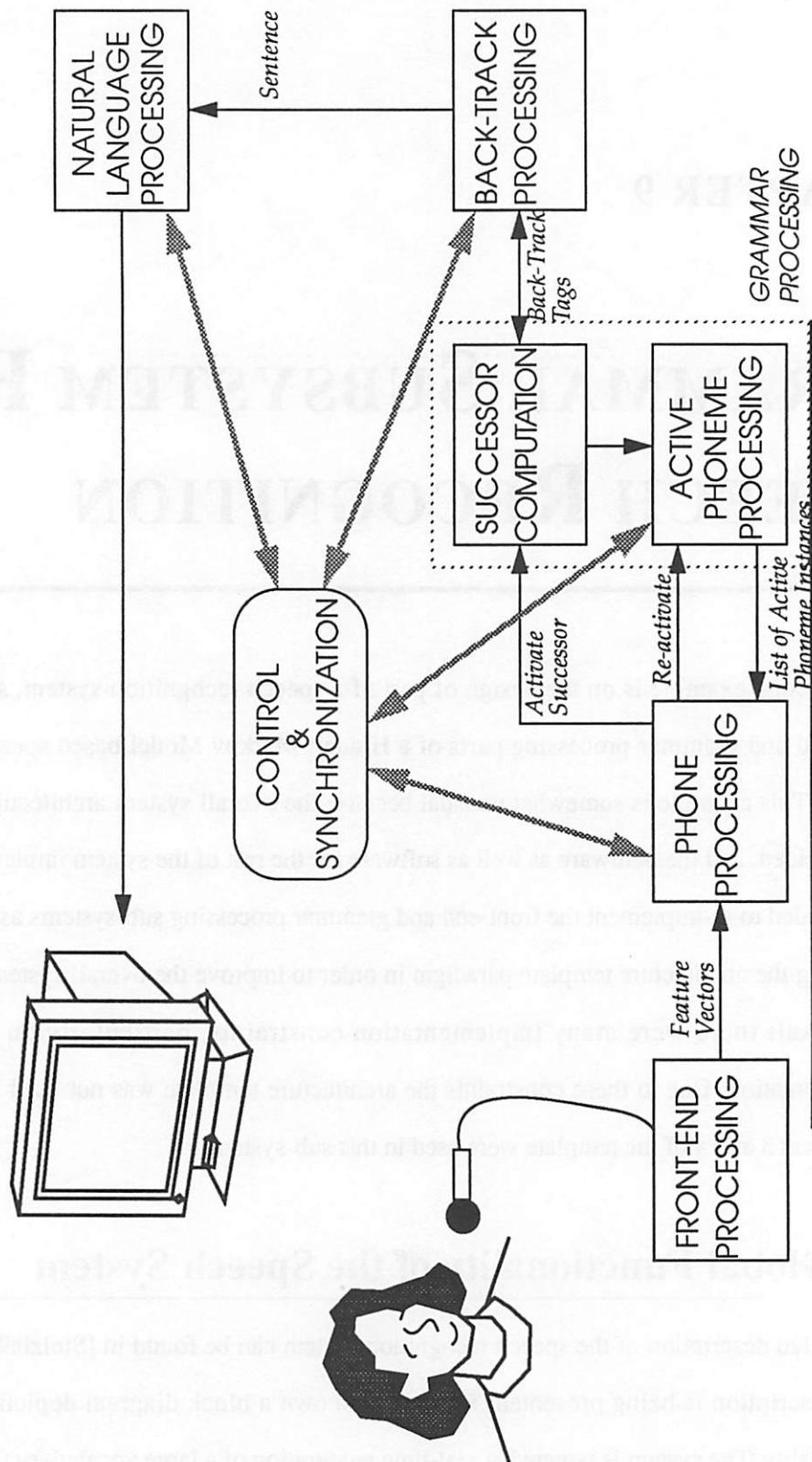


Figure 9-1 : Functional Decomposition of the Speech Recognition System

continuous time, speaker independent speech input using the popular Hidden Markov Model (HMM) approach. The basic idea behind the HMM approach is that the speech is modelled to be generated by a probabilistic function of a discrete time Markov random process. A state transition takes place in the Markov process every *frame* (typically about 10-20 ms), based on what is being spoken, at which time a speech segment (e.g., part of a phoneme), encoded as a feature vector, is output using a probabilistic output function associated with the state (or the state transition). The recognition problem is the inverse problem where given the speech in the form of a sequence of feature vectors at the frame rate, the task is to determine the most likely sequence of state transitions in the underlying (hidden) Markov process that would have caused the particular sequence of feature vectors to have been emitted. Knowing this sequence of states one can determine what is being spoken. An efficient way of organizing this search is by using a dynamic programming algorithm called the Viterbi algorithm. The knowledge about the language vocabulary is encoded in the topology of the state graph associated with Markovian random process, transition probabilities, and the probabilistic output functions.

As shown in the block diagram the task of speech recognition is decomposed into several sub-tasks. First, the incoming speech signal from a microphone is converted into a digital format, and then undergoes some signal processing to produce a vector of features at the frame rate. This task is called the *Front-End Processing*. The task of determining the most likely sequence of states itself has been divided into two by a hierarchical decomposition of the Markov model into *phoneme* and *grammar* level models. This decomposition allowed reduction in hardware as well as introduced a coarse-level parallelism. The phoneme level models describe the Markov models for individual phonemes, of which only a limited number are required in any given language even when variations in pronunciations are taken into account. These phoneme models form the basis set, or the building blocks, of the language. The grammar level model describes the language by composing instances of these phonemes to form words, and composing the words to form sequences of words. Transition probabilities are associated with transitions between phoneme

instances within a word, and also with transitions between words. As a result of this two-level decomposition the search for the most likely state sequence using the Viterbi algorithm also needs to be done simultaneously at two levels. The *Phone Processing* block carries out this search within a phoneme instance only considering the local transitions between states inside the Markov model associated with the phoneme. The result of these computations, together with transition probabilities between phoneme instances, are used by the *Grammar Processing* block to compute the probabilities of paths to the successor phoneme instances.

A list of phoneme instances to be processed by the *Phone Processing* block is maintained, and is called the list of *active* phoneme instances. During a frame the *Phone Processing* block works its way through the current list, while the *Grammar Processing* block builds up the list of active phonemes for the next frame using the information generated by the *Phone Processing* block. A phoneme instance gets inserted into the active phoneme list for the next frame if it was either on the current active list and has at least one state whose probability exceeds a certain threshold, or if it is a successor to a phoneme instance that has a high enough probability for starting. The *Grammar Processing* block can therefore in turn be decomposed into two sub-blocks. The first sub-block is the *Successor Computation* block which finds the successor phoneme instances for those phoneme instances that have a high probability of having reached their end, and also calculates the starting probabilities of the successor phoneme instances using the Viterbi algorithm. The second sub-block is the *Active Phoneme Processing* block is responsible for building the list of active phoneme instances to be processed in the next frame. It adds or updates phoneme instances as requested by the *Phone Processing* block or by the *Successor Computation* block.

The *Back-Track Processing* block is fired whenever there is a long enough pause in the incoming speech, thus indicating the end of a sentence. It then traverses the Markov model graph in reverse using back-track pointers that are set during the Viterbi search conducted by the *Grammar Processing* block. The output of this block is the most likely sequence of words spoken since the

last time the back-tracking was done.

The final block in the functional decomposition of the speech recognition system is the *Natural Language Processing* block which takes the sequence of words produced by the *Back-Track Processing* block, and uses contextual information and semantic models to understand what is being said by the speaker.

9.2 Old Implementation of the Speech System

Before the board described later in this chapter was made, a version of the speech system was implemented as shown in Figure 9-2. During that initial implementation it was rationalized that the *Phone Processing* block was the most time-critical, being in the inner-loop of the computation. Every state in every active phoneme instance needs to be processed within a frame of 10 ms, whereas at the grammar level only transitions between phoneme instances need to be processed. In addition, the Viterbi search is simpler for phoneme level models as the topology of the associated Markov model is very regular - it is left-to-right with a few local transitions. In contrast, the Viterbi search is not so simple at the grammar level because the topology of the Markov model, while simple left-to-right inside a word, is quite irregular and dense between words - at word boundaries a transition can occur between any two phoneme instances.

This combination of factors: high throughput and regular computation inside the *Phone Processing* block, and lower throughput and irregular computation inside the *Grammar Processing* block motivated the use of dedicated hardware for the *Phone Processing* block. In addition, the *Active Phoneme Processing* sub-block of the *Grammar Processing* block was also considered to have computational requirements deserving a dedicated hardware implementation. The result, as shown in the figure, was that the *Phone Processing* block, and the *Active Phoneme Processing* sub-block were implemented using ASICs on a custom VME board, called the *HMM Board*. A custom memory board, called the *Distribution Board* which uses the VME bus, was also made to store the

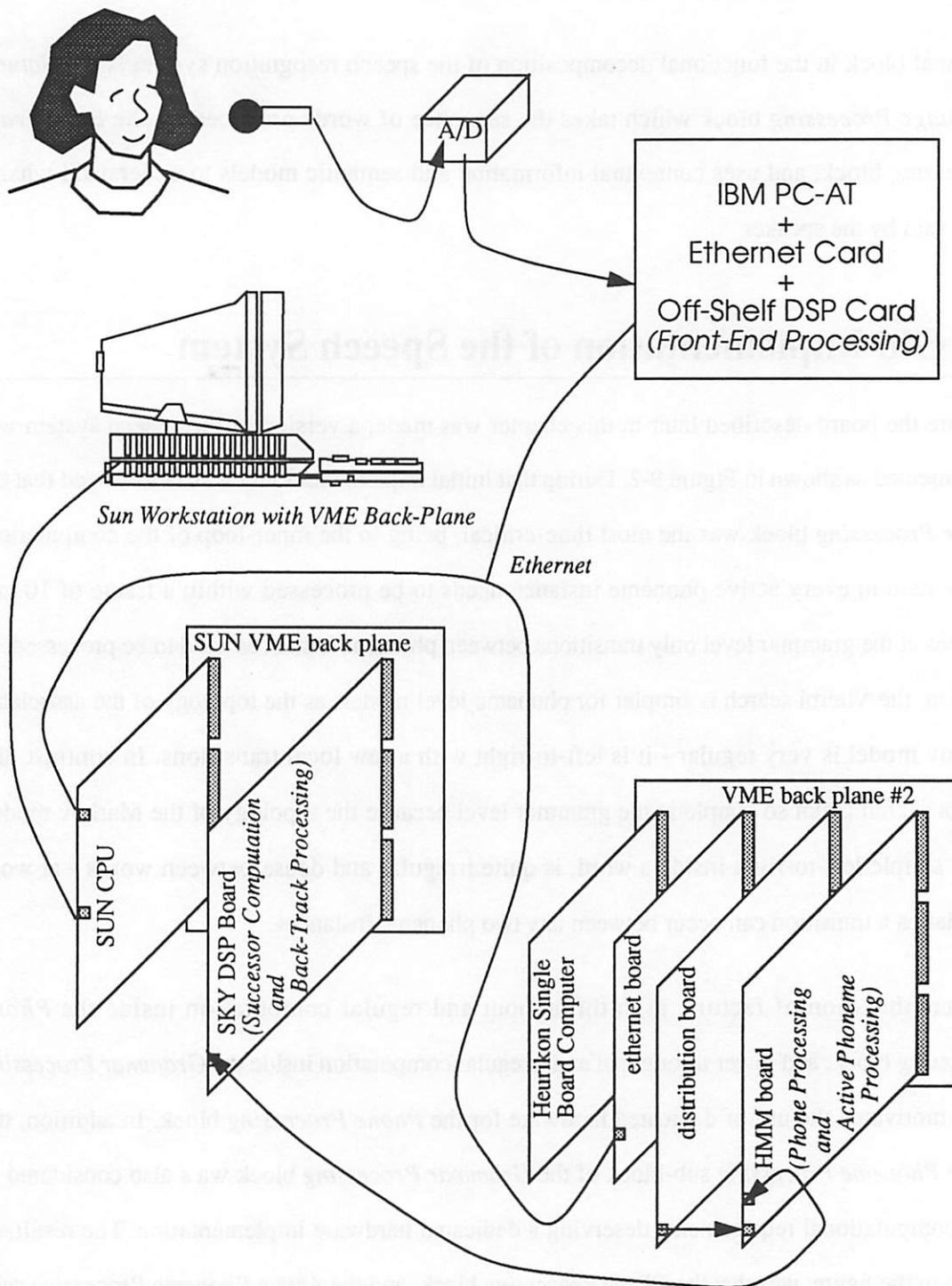


Figure 9-2 : Old Implementation of the Speech Recognition System

phoneme topologies, transition probabilities, output probability functions, and other information needed for phoneme level processing.

The rest of the system, including the *Front-End Processing* block, the *Successor Computation* block, the *Back-Track Processing* block, and the *Natural Language Processing* block were considered suitable for off-shelf boards and computers. The result, unfortunately, is an inelegant and inefficient system solution as well demonstrated by the figure. The system spans three backplanes - a VME bus controlled by a Heurikon HKV2F single-board computer on which the two custom boards are plugged, another VME bus associated with a SUN 3 system on which a commercial DSP board (from SKY computers) with two TMS320C30 processors is plugged, and an IBM PC-AT bus in which another commercial DSP board based on TMS3202x processor is plugged.

Not only was this implementation mechanically inelegant and unwieldy, its loosely coupled nature - a result of the reliance on off-shelf boards - resulted in a tremendous penalty on I/O performance. For example, the *Successor Computation* block, and the *Back-Track Processing* block are implemented on the SKY board. In addition to being computationally inadequate, the I/O interface between the SKY board and the HMM board proved unreliable, complicated to use, and to have inadequate bandwidth. One reason for this is that the data being transferred between the two boards is inherently 96-bit wide, but had to be packetized and depacketized using a special interface chip just because the SKY board had only a 32-bit connector even though it has two 32-bit processors. This resulted in a wastage of precious cycles on the DSP processors on the SKY board. In a similar vein, an entire IBM PC was needed just as to be able to implement the *Front-End Processing* block.

The net result of this implementation was that the system was unreliable, and because of the I/O bottleneck between the custom HMM board, and the off-shelf SKY board, was able to perform in real-time only for a 1000 word vocabulary even though the custom HMM board is capable of

supporting a 60,000 word vocabulary.

9.3 Re-Implementation of Front-End Processing, Successor Computation, and Back-Track Processing

In order to alleviate the problems associated with the implementation of the speech system, as outlined in the previous section, it was decided to re-implement parts of the system, originally relegated to off-shelf boards, by a custom VME board built using the tools and ideas outlined in this thesis. This single custom VME board had to implement the *Front-End Processing* block, the *Back-Track Processing* block, and the *Successor Computation* sub-block of the *Grammar Processing* block. In addition to providing sufficient computation power, the board also had to provide a high throughput interface to the HMM board while meeting the signalling, electrical, and mechanical constraints already imposed by the HMM board.

Such a custom VME board was successfully implemented and also served as a driver for the work in this thesis. Figure 9-3 shows how the new speech system, based around this custom board. As is immediately obvious the new system is much more compact as the two backplanes are made redundant. In addition, the increased I/O bandwidth between the custom board and the HMM board, together with increased computation power, makes it possible to achieve more than 5000-6000 word vocabulary with one custom board. Since the architecture is linearly scalable, higher vocabularies can be achieved by using multiple copies of the board. In addition, use of surface mount technology for the board would also increase the computation power that can be packed on each board.

9.4 Board Requirements and Constraints

This section describes the computational and I/O requirements on the board, and the implementation constraints on hardware and software imposed by the fact that it needs to be plugged in an existing system.

9.4.1 Front-End Processing

The *Front-End Processing* block is required to take the incoming speech from a microphone, and to produce a feature vector every frame of 10 ms. Figure 9-4 shows a block diagram of the processing that needs to be done by this block. The approach used in this system belongs to the

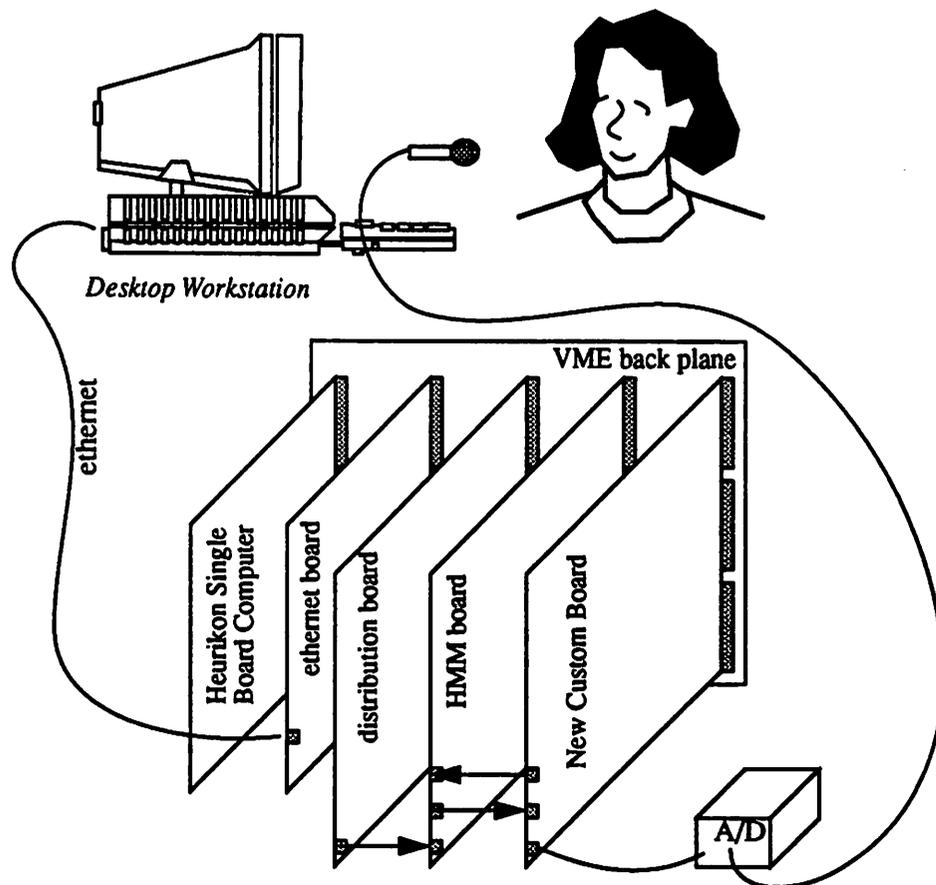


Figure 9-3 : The Speech System Using the New Custom Board for *Front-End Processing, Back-Track Processing, and Successor Computation*

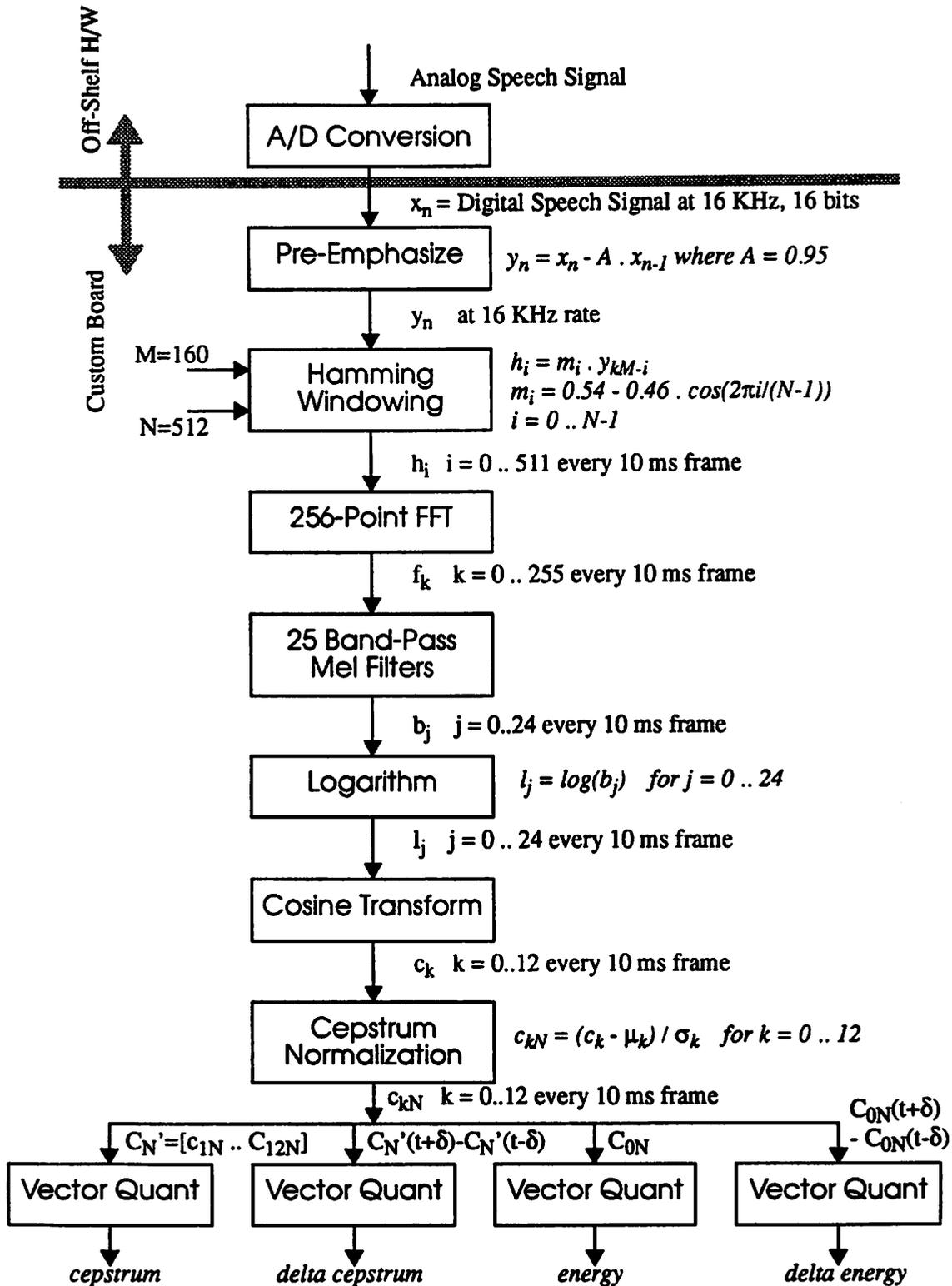


Figure 9-4 : Front-End Processing Block

class of non-parametric algorithms for speech analysis as it directly measures certain features of the speech.

In the current implementation it was decided that the analog speech signal from the microphone be sampled and converted into a digital format using a commercial A/D converter box. The commercial box is configured at run-time to anti-alias filter the analog signal and sample it signal at 16 KHz, linearly quantize it to 16 bits, and then transmit it using a serial communication protocol that is standard in the TMS320xx signal processing chips produced by Texas Instruments. In addition certain control signals are also required by the converter box, and are used to configure it at start-up.

The part of the functionality of this block that needs to be implemented on the custom board is also shown in the figure. The input is the speech sampled at 16 KHz and linearly quantized at 16 bits, being transmitted using Texas Instruments' standard serial protocol. This incoming speech is first pre-emphasized, and then blocked into frames of $N=512$ samples such that the frames are spaced $M=160$ samples (10 ms) apart. In effect consecutive frames have an overlap of 352 samples. Each frame is then smoothed by a Hamming Window. A 256-point FFT is then calculated from each windowed frame. The FFT spectrum is then integrated through a bank of $L=25$ bandpass Mel filters which are filters that are linearly spaced on the Mel frequency scale - a scale modelled on human auditory system such that the scale is linear below 1 KHz and logarithmic above 1 KHz. The energy of each of the bandpass filter output is then converted into the logarithm domain. Using cosine transform a vector of 13 cepstral coefficients is then calculated from the logarithmic energy values, and then the cepstral coefficients are normalized using mean and variance data obtained from a speech database. Using the normalized cepstral coefficients four features are calculated by vector quantization of the cepstral coefficient vector (*cepstrum*), the difference of cepstral coefficient vectors (*delta cepstrum*), the signal energy (*energy*), and the difference energy (*delta energy*). The output is a vector of four 8-bit codes, obtained by vector quantization, every 10 ms frame.

Since the input speech signal is available in Texas Instrument's serial protocol, it is natural for the front-end processing to be done on a signal processor from Texas Instruments, thus simplifying the hardware tremendously. This makes the TMS320C30 processor module described in Chapter 3 a natural choice. Software in C that implements the front-end signal processing outlined above is still under development [Lu92] for the TMS320C30 processor module running under the SPOX kernel.

The existing implementation of the system requires that the output feature vector produced every frame should somehow be transmitted to a control process running on the single-board computer that is the VME bus-master. The control process ships this data over to the *Distribution Board* when needed. In addition it also performs silence detection using the feature vectors.

9.4.2 Successor Computation

The *Successor Computation* sub-block is part of the *Grammar Processing* block. It receives input from the *Phone Processing* block implemented on the HMM board. The input consists of requests to activate successors of a phoneme instance. Each request is a structure that consists of three fields:

- a. D : the 20-bit *phoneme instance id* of the phoneme whose successors need to be activated in the next frame.
- b. $P(D)$: the 16-bit *probability* that the most probable path at the current frame terminates at the end of the phoneme instance.
- c. $TAG(D)$: the 20-bit *back-track tag* associated with the phoneme instance.

The requests are transmitted by the HMM board over a parallel bus using an asynchronous handshake protocol. In addition to a 56-bit data bus needed for the above three fields, there are control signals. Two signals *Req* and *Ack* are used to implement a two-wire four-phase handshake for transmission of the 56-bit data. A control signal *Active* is sent by the HMM board, and its high-to-low transition indicates that the *Phone Processing* block has generated all the requests for this frame. Another signal *Stall* can be sent to the HMM board in order to temporarily stop new

requests from being generated. This is required to avoid buffer overflows if the *Phone Processing* block on the HMM board produces requests at too fast a rate.

For each such request the *Successor Computation* sub-block has to do the following task using the grammar model. The grammar model is stored locally and defines the phoneme instances by referring to their unique phoneme id (the id of the master phoneme) and topology address, and it specifies the successors of a phoneme instance and the transition probabilities $A(i,j)$ to these successors.

```

For every phoneme instance S that is successor to phoneme instance D
(
  P(S) = P(D) * A(D,S);
  Store the structure (D, TAG(D)) in a Back-Track Memory
  and let TAG(S) be the 20-bit address at which it gets stored
  if (P(S)>threshold) {
    update threshold;
    send the following to the HMM Board:
      PI (20-bit) : phoneme instance id of S
      UniqueAdd (16-bit) : the unique phoneme id of S
      TopoAdd (4-bit) : address of the topology associated with S
      P(S) (16-bit) : probability of the source node of S
      TAG(S) (20-bit) : back-track tag to be associated with S
  }
)

```

The data is sent to the HMM board using another parallel data bus with two-wire four-phase handshake. In addition, a special signal *EndFlag* to the HMM board needs to be asserted by the *Successor Computation* block once it has finished processing all the requests.

The above task can be extremely computationally demanding if the average number of successors for each phoneme instance is large - such grammar models are said to have high perplexity. The grammar model is usually quite irregular and a phoneme instance can have an arbitrary number of successors. This makes a hardwired implementation inefficient. Also, it is often desirable to dynamically update the grammar model using input from a natural language. Both these facts suggest that a software implementation is desirable. Further, no single programmable processor can meet the computational demands of high perplexity grammars, multiple software programmable processors will be needed and the computation needs to be somehow partitioned

among them.

9.4.3 Back-Track Processing

This is probably the simplest block in the entire system. The control process running on the VME master single-board computer generates a message for the *Back-Track Processing* block whenever a pause is detected, typically indicating the end of a sentence. The message content is the back-track tag of the phoneme instance that has the highest probability in the last frame at the end of the Viterbi search (which indicates that the most likely sequence of phoneme instances ends at that phoneme instance).

A careful look at the *Successor Computation* task described in the previous sub-section shows that the process of calculating the back-track tag for the successor phoneme instance results in multiple linked lists being built in the back-track memory. Each list corresponds to potential paths through the Markov model. The back-track tag sent by the control process to the *Back-Track Processing* block is an address for the back-track memory, and indicates the start of one of the linked lists. All the *Back-Track Processing* block has to do is to traverse this list by following the pointers, and generate a sequence of phoneme instances, or work, and send it to the control process running on the single-board computer.

As is obvious this task is not computationally intensive, and can easily be done in software.

9.5 Hardware Architecture and Implementation

From the discussion in the previous section one can summarize the following characteristics for the three tasks that need to be carried out:

- a. The *Front-End Processing* block needs a Texas Instrument's digital signal processor for ease of hardware interfacing. Computationally it is not demanding enough to require dedicated hardware, and can easily be implemented in software on a digital signal processor.
-

-
- b. The *Successor Computation* block can be computationally very demanding for high perplexity grammars. However its irregular nature of computation suggests that a software implementation using multiple processors is desirable. It also requires a very specialized and high-bandwidth I/O interface to the HMM board.
 - c. The *Back-Track Processing* block is not computationally demanding and can easily be implemented in software. Further, it shares the Back-Track memory with the *Successor Computation* block in a tightly coupled fashion.

In light of the above an extensible board architecture based around multiple TMS320C30 processor modules was chosen, together with a high-performance ASIC based I/O interface to the HMM board. A global view of the board architecture is shown in Figure 9-5. As is readily apparent this architecture follows the architecture template outlined in Chapter 6. The board contains three identical TMS320C30 processor modules at layer 3 in the architecture template. Each module supports 8 Mbytes of 1 wait state SRAM. Each module also uses the dual-port RAM based interface to layer 2 that was described in Chapter 6. The interface is configured for 32 Kilobyte of true dual-ported shared memory, 16 hardware semaphores, and interrupts in either directions.

Each of the three layer 3 processor module is also connected to the other two, again using the dual-port RAM based interface. Effectively the three processors are fully connected on a ring.

One of the processors has a simple layer 4 slave attached to it in order to control the external A/D converter connected to the microphone. The serial data line from the converter is connected directly to the serial port of the processor.

There is a second layer 4 slave that has three almost identical ports - one connected to each of the layer 3 processor modules. This slave implements a high bandwidth interface to the HMM board that is flexible enough to allow the partitioning of the *Successor Computation* task on to the multiple processors in a variety of fashion and simplifies load balancing. The implementation is done using ASICs and PLDS. The slave module uses token passing mechanism to distribute the requests coming from the HMM board to the processors, and to gather the replies generated by the processors and send them to the HMM board. This input distribution and output gathering can be

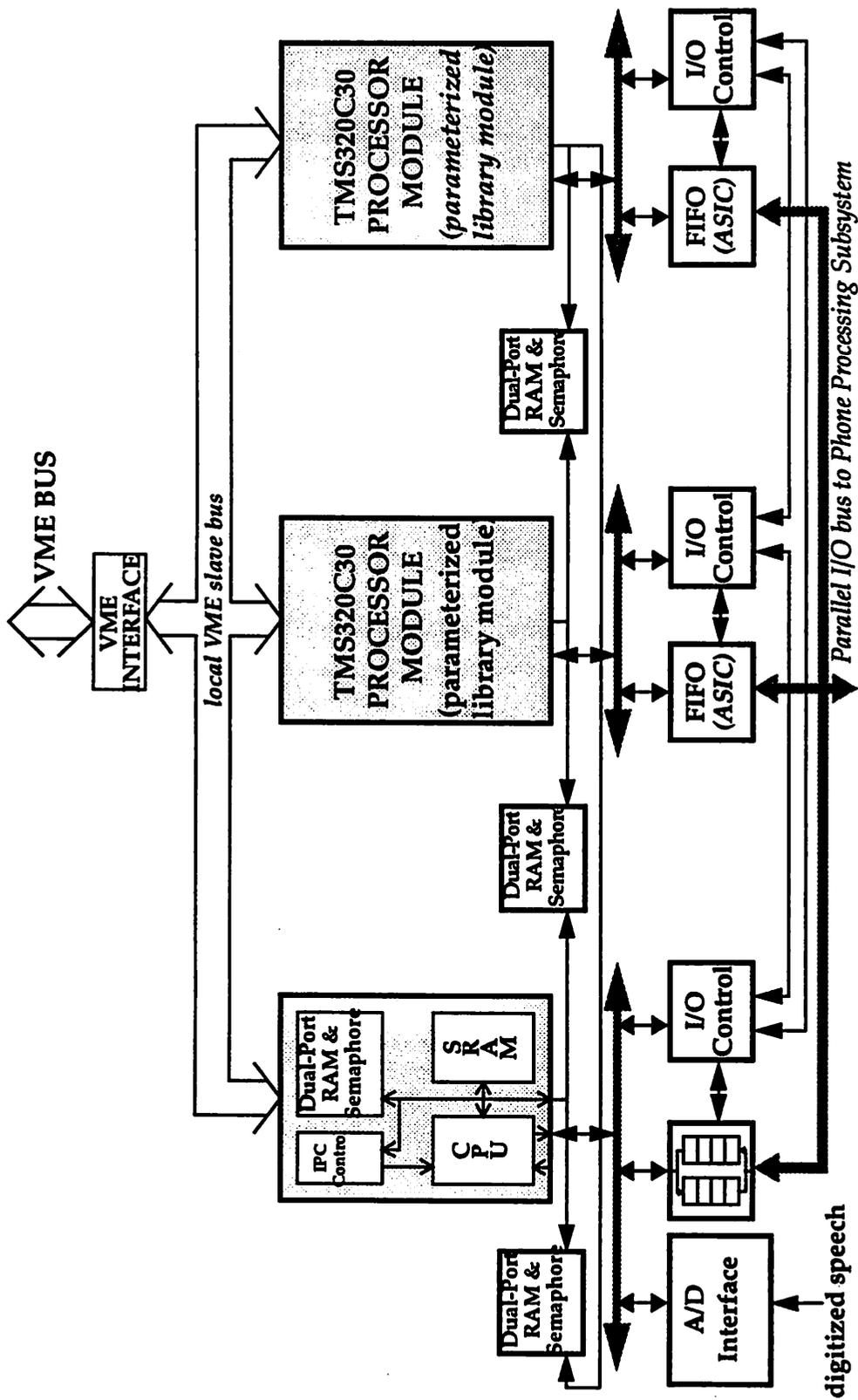


Figure 9-5 : Architecture of the Custom Board for Front-End Processing, Back-Track Processing, and Successor Computation

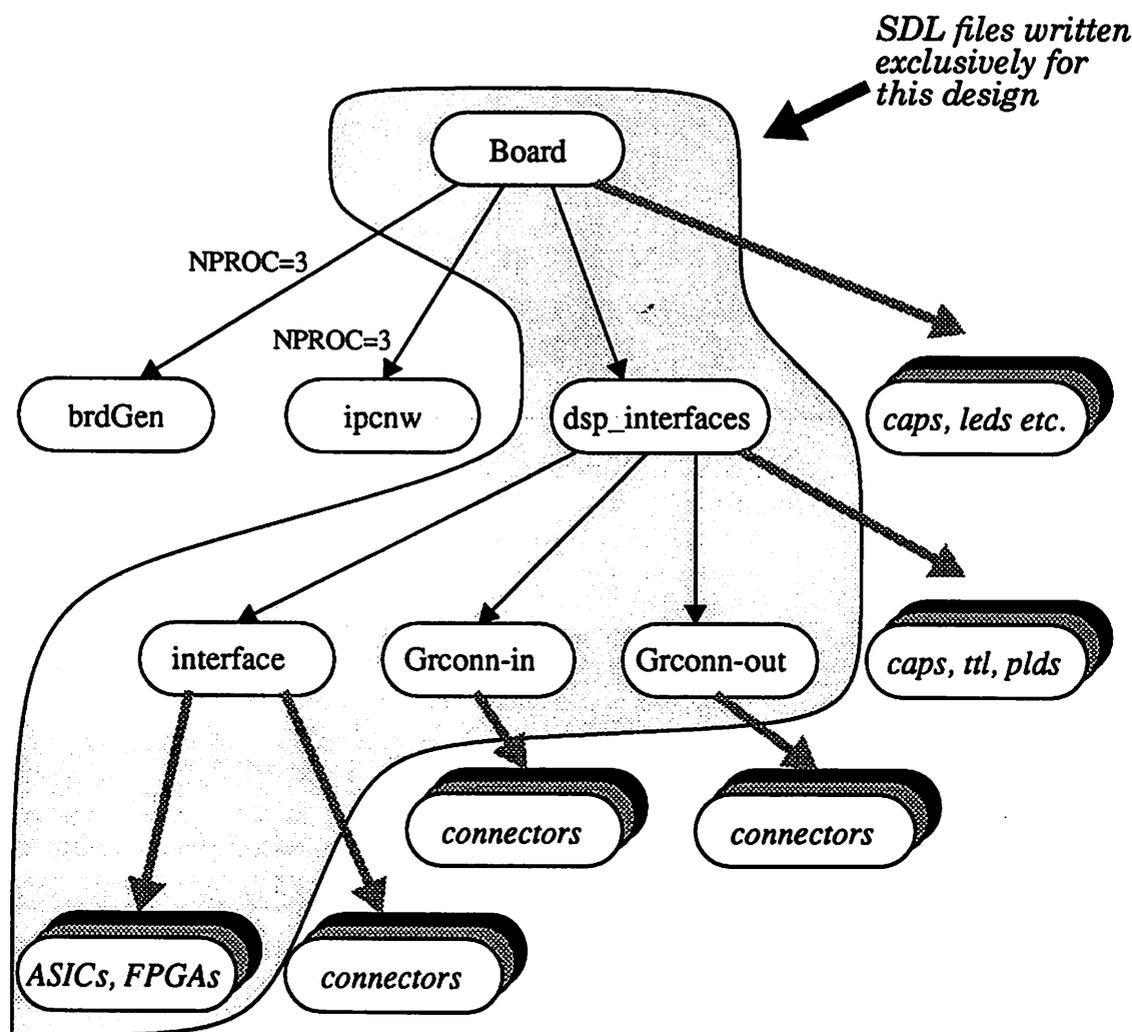
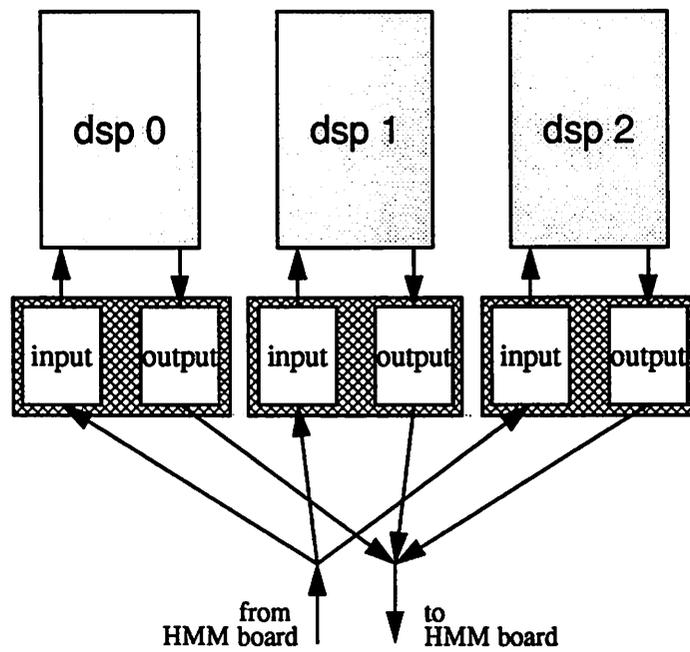


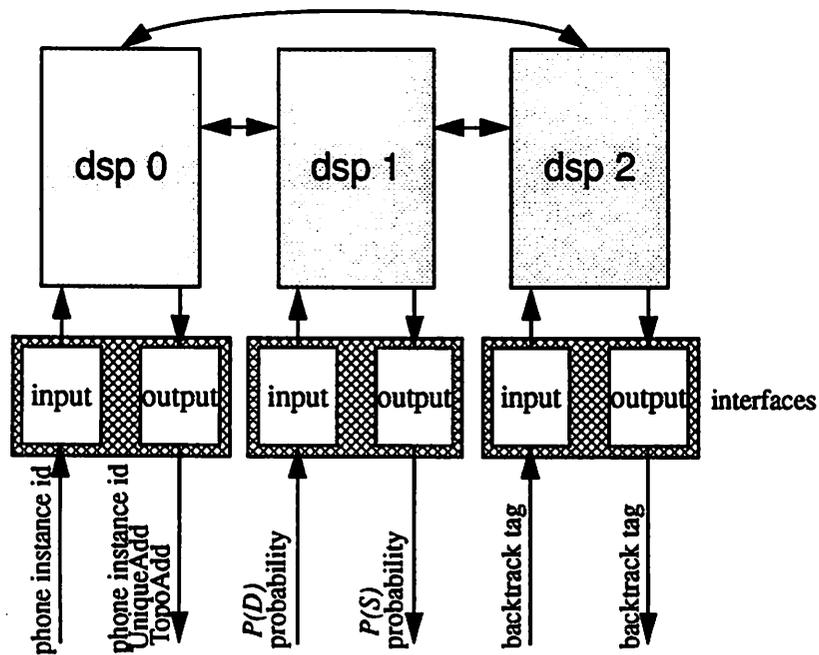
Figure 9-7 : SDL Hierarchy for the Custom Board

done in a *parallel* mode, or in a *bit-slice* mode as shown in Figure 9-6.

The reader is referred to Chapter 8 of [Stölzle91] for details of the hardware implementation, particularly the custom interface to the HMM board. Figure 9-7 shows the SDL file hierarchy in terms of modules as seen by the designer - in other words, reusable library modules are considered primitives. As one can see most of the complex parts of the board were taken care of by parameterized library modules thus tremendously simplifying the design task. Table 9-1 lists the



(i) Layer 3 Processors Operating in Parallel



(ii) Layer 3 Processors Operating in Bit-Sliced Fashion

Figure 9-6 : Alternate Ways of Structuring *Successor Computation* Using the Flexible Custom Interface to the HMM Board

Dimensions	14" x 16" 9U-VME Board
Number of Layers	12
Number of Components	350 parts + 400 bypass capacitors
Design Time	2.5 man-months, including time for reusable library parts
Amount of SDL Code	1800 lines excluding re-usable library parts and ASICs

Table 9-1 : Salient Features of the Custom Board

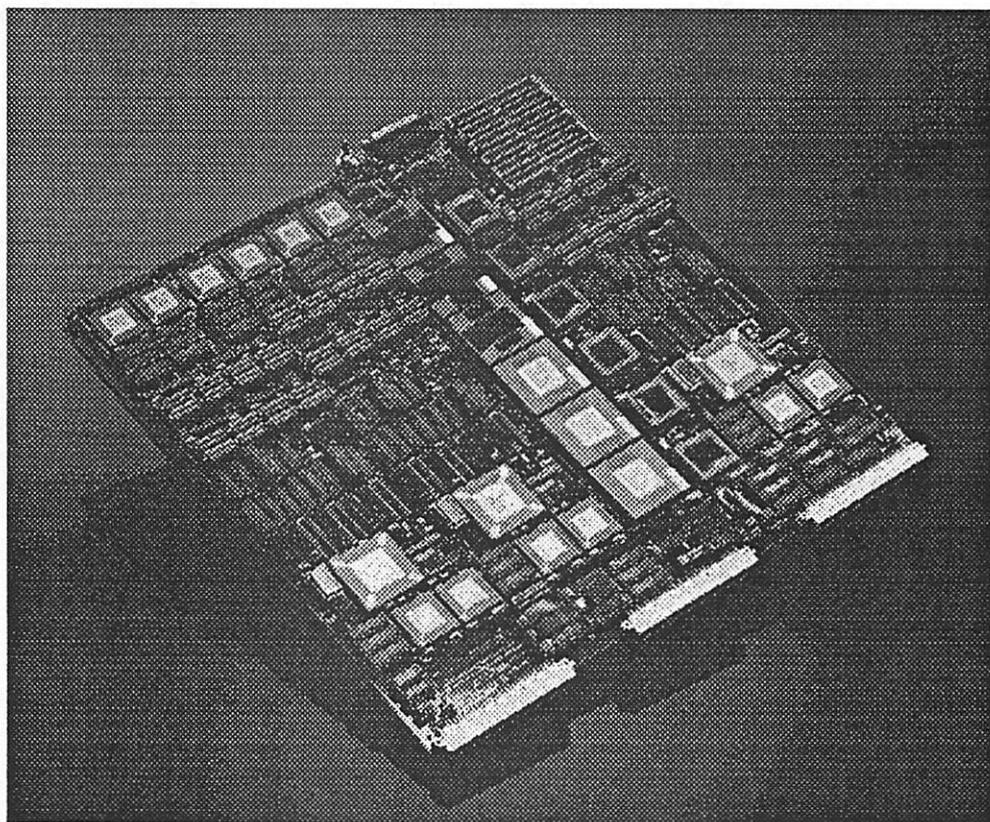


Figure 9-8 : Photograph of the Custom Board

salient features of the board, and Figure 9-8 shows a photograph of the board.

Variations of the board, for example with different processor configurations or even with different number of processors can be generated in a matter of hours, most of which are spent on routing. The parameterized module libraries were a critical element in making possible such a short

iteration time.

9.6 Software Implementation

The application software modules are still under development by the researchers associated with the speech recognition project. However all the system software, including the kernels on the processor modules, the UNIX-like I/O libraries, and the initialization software is fully functional. Thus a powerful and scalable, but use-friendly, multi-processor tailored for the speech recognition system is functional with a multi-tasking kernel at each processing node.

9.7 Summary

The architecture template and module generation utilities have been applied to a sub-system of an existing speech recognition system. The custom board relies on a mix of software for computation and dedicated hardware for high-bandwidth flexible I/O, and demonstrates the utility of customizing the architecture at the board level.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

This thesis presented a computer-aided methodology and framework for the design of application-specific systems. The framework emphasizes higher level aspects of system design, of which chip level design is one component.

10.1 Conclusions from this Work

It is clear from the discussion and examples presented earlier in this thesis that rapid-prototyping of multi-board application-specific real-time reactive systems presents many new problems that are not present at the chip level. This work took a systems view of the design process in order to identify the real problems that need to be addressed. The primary problem in the design of these systems was found to be one of concurrent development of the hardware and software aspects of a system. This *Hardware-Software Co-design* problem is present at all stages of the design of a system - specification, simulation, architecture generation, module generation, and physical implementation. Based on this the thesis addressed the problems of generation of board-level hardware modules and software modules, high level specification of the system behavior that may

be implemented in a mix of hardware and software, the simulation of the high level specification, and generation of appropriate system level architectures composed of both hardware and software.

Techniques from CAD for chips and software continue to be applicable to some aspects of these problems. For example, the notion of parameterized libraries and module generators proved successful in board level hardware module generation as well.

On the other hand some of the system level problems required new solutions. Narrowing the scope of the design problem by developing a target architecture that is not so restrictive as to not be useful for many applications was one such problem, as was the specification of the system in a fashion that does not discriminate between the hardware and the software components. The notion of *process* was used to provide the needed unifying thread between hardware and software. Its symmetric treatment of hardware and software facilitates migration of functionality across software and dedicated hardware. The concept of a layered architecture model was useful in providing a target architecture on which the system functionality is mapped. Processes with different computation and I/O requirements are mapped to layers with different characteristics.

Communication and synchronization between the hardware as well as software modules was identified to be another key problem. The solution to this problem is intimately linked to the layered architecture model being used. The concept of layering is also useful in implementing the communication and synchronization. Higher level communication and synchronization primitives - like the *channel* object from Chapter 5 - are implemented on the top of lower-level primitives.

At the lowest level, the problem of communication and synchronization between hardware or software processes that are mapped to different physical hardware modules manifests itself as the problem of generating the interconnect logic between two hardware modules. This problem is the primary subject of a co-researchers thesis [Sun92a].

10.2 Open and Unsolved Problems

Although a complete, vertically integrated, design methodology was adopted and presented in this thesis, there are several problems which were left unsolved and for which manual techniques were adopted. These problems, which are listed below, can serve as a map for additional near-term research.

- a. Restructuring of the process network description using behavior preserving transformations like process coalescing to achieve a more efficient implementation. Other transformations include process splitting, changing the channel protocol, sizing the channel buffer depth etc.
- b. A single or a set of languages for unified expression of the behavior of each of the processes in the process network.
- c. Automatic partitioning of the possibly restructured process network to an instance of the architecture template. The suitable instance of the architecture template also needs to be found in the process.
- d. Characterization of the computation and I/O performance of the software programmable processor modules. This is useful in performance estimation as well as in doing a good job at the automatic partitioning mentioned above.
- e. Automatic code generation for software modules from the high level specification.
- f. Extraction of performance characteristics from the physical output of board level hardware module generators, and simulation using these performance characteristics.

10.3 Future Directions in Systems Design

The long-term research in this nascent area of design methodology for complex systems will inevitably have to track the directions taken by the systems themselves. Over the past year or two several interesting trends have emerged in the types of computing systems being designed, and a look at these trends can provide a charter for future research efforts in the area of computer-aided methodologies for system design.

Much of the evolution of present day computing systems can be characterized as occurring along two main directions which include:

- a. Systems that are *devices* with which a user directly interacts. Examples of such systems are personal communication systems, multi-media terminals, smart appliances, home robots etc.
-

-
- b. Systems that provide the *infrastructure* for supporting the *devices*. Such systems include the communication networks that may connect the *devices*, compute and control servers, multi-media databases, signal-processing servers etc.

These two classes of systems - *devices* and *infrastructure* - have many unique requirements that need to be addressed during their design.

For one (*devices*) type of systems two very important problem areas are *design for portability* and *mechanical design*. For example low-power is a key aspect of design for portability, and requires power to be explicitly incorporated in the cost metrics at all levels of system design - architecture generation, module generation, hardware-software partitioning, physical design etc. Further, design for portability makes efficient packaging and interconnect important. This makes a proper mechanical design of the system extremely important and requires addressing problems of 3D mechanical analysis, interconnect and packaging. A new significant problem is that of the role played by mechanical packaging and interconnect constraints on the design and partitioning of the system at the level of software and electrical hardware.

Systems that are classified as *infrastructure* are distributed in nature, and thus issues of large-scale distributed computation and database management become important. Data storage and compression technologies are required as well as hardware for high-speed compute, control, and signal-processing. These infrastructure systems have speed as the primary performance metric as opposed to power and size in the case of *devices*.

These two classes of systems also share many common attributes, and this requires a set of enabling technologies that are useful for the design of both these classes of systems. Some of the important problems that need to be addressed are:

- a. portable *micro-kernels* for efficient software organization
 - b. meeting *real-time* constraints in the design of mixed hardware-software systems
 - c. interconnect protocols for a variety of physical medium
 - d. unified language for specifying and representing the software, hardware and mechanical behavior of systems
-

-
- e. re-usable sub-system libraries for software, hardware and (electro)mechanical domains
 - f. integrated simulation of software, hardware, and (electro)mechanical components of a system
 - g. partitioning system hardware and software taking into accounts constraints imposed by mechanical interconnect and packaging

The common theme that emerges from the above discussion is that systems have three types of components - software, hardware, and mechanical. All these components interacts, and therefore system design necessarily requires a concurrent design of these three aspects of a system. This suggests that research is needed so that the work described in this thesis evolves into a CAD framework for

“Software-Hardware-Mechanical Co-Design”



BIBLIOGRAPHY

1. [Albus89] J. S. Albus, H. G. McCain, and R. Lumia. *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*. NIST Technical Note 1235, 1989 Edition, National Bureau of Standards, April 1989.
2. [Andrew83] Gregory R. Andrews and Fred B. Schneider. *Concepts and Notations for Concurrent Programming*. Computing Surveys, vol. 15, no. 1, March 1983.
3. [Arya89] Manish Arya. *A Standard Software Platform for Shared Memory Multiprocessor Signal Processing Systems*. M. S. Report, EECS Department, U. C. Berkeley, 1989.
4. [Åström82] Karl Johan Åström. *A SIMNON Tutorial*. Technical Report, Lund Institute of Technology, 1982.
5. [Azim88a] S. K. Azim. Application of Silicon Compilation Techniques to a Robot Controller Design. Ph. D. Thesis, EECS Department, U. C. Berkeley, 1988 (UCB/ERL M88/35).
6. [Azim88b] S. K. Azim, C-S Shung and R. W. Brodersen. *Automatic Generation of a Custom Digital Signal Processor for an Adaptive Robot Arm Controller*. Proceedings of ICASSP, vol. 4, pp. 2021-2024, 1988.
7. [Azim88c] S. K. Azim and R. W. Brodersen. *A Custom DSP Chip to Implement a Robot Motion Controller*. Proceedings of CICC, May 1988.
8. [Baringer91] W. B. Baringer. *A Radon Transform Image Processing System*. Ph. D. Thesis, EECS Department, U. C. Berkeley, 1991.
9. [Ben-Ari90] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall International, 1990.
10. [Bennett88] Stuart Bennett. *Design of Real-Time Systems*. Chapter 5 of Real-Time Computer Control: An Introduction, Prentice-Hall, 1988.
11. [Bier89] J. C. Bier, and E. A. Lee. *Frigg: A Simulation Environment for Multiple-Processor DSP System Development*. Proceedings of International Conference on Computer Design, October 1989.
12. [Birmingham89] W. P. Birmingham, A. P. Gupta and D. P. Siewiorek. *The MICON System for Computer Design*. IEEE Micro, vol 9, no. 5, October 1989.
13. [Birmingham92] W. P. Birmingham, A. P. Gupta and D. P. Siewiorek. *Automating the Design of Computer Systems: The Micon Project*. Jones and Barlett Publishers, February 1992.

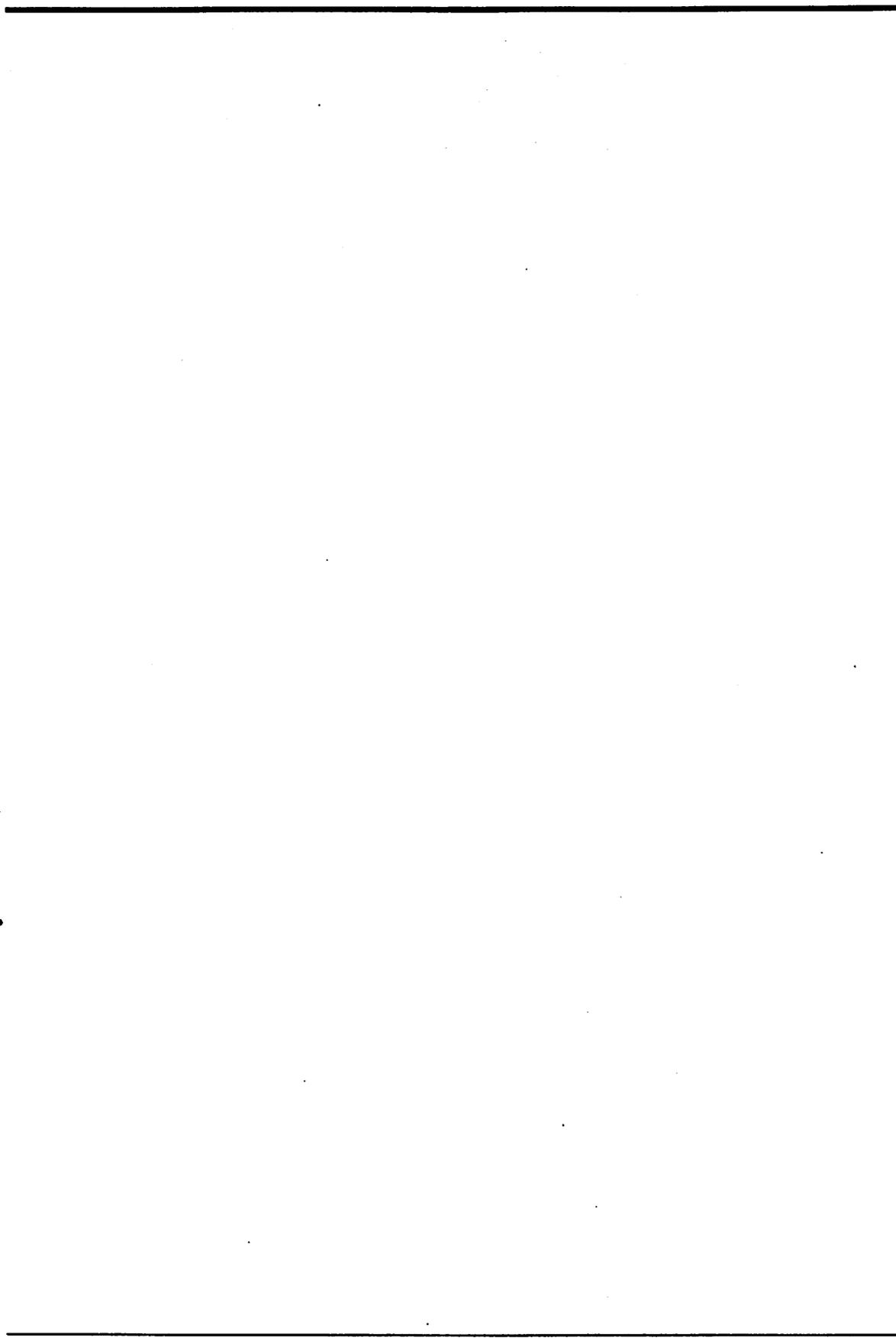
-
14. [Brady82] Michael Brady, John M. Hollerbach, Timothy L. Johnson, Tomas Lozano-Perez, and Matthew T. Mason. *Robot Motion, Planning and Control*. The MIT Press, 1982.
 15. [Brady89] Michael Brady (ed.). *The Problems of Robotics*. Chapter 1 of Robotics Science, The MIT Press, 1989.
 16. [Brodersen92] R. W. Brodersen (ed.). *Anatomy of a Silicon Compiler*. Kluwer Academic Publishers, 1992.
 17. [Chen86] J. B. Chen, et. al. *NYMPH: A Multiprocessor for Manipulator Applications*. IEEE International Conference on Robotics and Automation, April 1986.
 18. [Chu87] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. IEEE International Conference on Computer Design, 1987.
 19. [Chu89] C.. M. Chu, M. Potkonjak, M. Thaler, and J. Rabaey. *HYPER: An Interactive Synthesis Environment for Real Time Applications*. Proceedings of ICCD 1989, October 1989, pp. 432-435.
 20. [Craig79] J. Craig and M. Raibert. *A Systematic Method for Hybrid Position/Force Control of a Manipulator*. IEEE Computer Software Application Conference, November 1979.
 21. [Craig86] John J. Craig. *Introduction to Robotics*. Addison-Wesley, 1986.
 22. [Dennis74] J. B. Dennis. *First Version of a Data Flow Procedure Language*. Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, Springer-Verlag, 1974.
 23. [Doshi89] Gautam Doshi. *Design and Implementation of a Six-Axis Robot Controller*. M. S. Report, EECS Department, U. C. Berkeley, 1989.
 24. [EDC90] European Development Center. *EDC/DSP Station, SILAGE Language Reference Manual*. EDC Document Number DSP_SIL_2.0-C0A-00, March 1990.
 25. [Fu87] K. S.Fu, R. C. Gonzalez, and C. S. G. Lee. *Robotics Control, Sensing, Vision and Intellingence*. McGraw Hill, 1987.
 26. [Gomaa84] H. Gomaa. *A Software Design Method for Real-Time Systems*. Communications of the ACM, September 1984.
 27. [Gupta92a] Rajesh K. Gupta, and Giovanni De Micheli. *System-Level Synthesis Using Re-Programmable Components*. Proceedings of European Design Automation Conference, March 1992.
 28. [Gupta92b] Rajesh K. Gupta, Claudionor N. Coelho, and Giovanni De Micheli. *Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components*. Pre-print of paper to be presented at the Design Automation Conference, June 1992.
 29. [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 ICCAD, 1986, pp. 24-27.
 30. [Hoang92] P. Hoang, and J. Rabaey. *A Compiler for Multiprocessor DSP Implementations*. ICASSP, March 1992.
 31. [Hoare85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
 32. [Hogan] N. Hogan. *Impedance Control: An Approach to Manipulation: Parts I, II & III*. ASME Journal of Dynamic Systems, Measurement and Control, vol 107:1-24.
 33. [Hubbard90] P. Hubbard, and J. Torres. *Using VHDL for High-Level and Stochastic System Modelling*. Fall 1990 VHDL Users' Group Meeting.
 34. [Ish-Shalom88] J. Ish-Shalom, and P. Kazanzides. *SPARTA: Multiple Signal Processors for High-Performance Robotics Control*. IEEE International Conference on Robotics and Automation, April 1988.
 35. [Jain91] R. Jain, P. T. Yang, and T. Yoshino. *FIRGEN - A Computer-Aided Design System for High Performance FIR Filter Integrated Circuits*. IEEE Transactions on Signal Processing, July 1991.
 36. [Jassica85] J. R. Jassica, S. Noujaim, R. Hartley, and M. J. Hartman. *A Bit-Serial Silicon Compiler*. Proceedings of ICCD 85, October 1985.
 37. [Jones90] G. Jones. *ASYN Man Page*. EECS Department, U. C. Berkeley, 1990.
 38. [Kahn74] G. Kahn. *The Semantics of a simple language for Parallel Programming*. Information Processing, 74, North Holland, Amsterdam, 1974.
 39. [Kalavade92] Asawari Kalavade. *Hardware-Software Co-Design in Ptolemy*. DSP Seminar, EECS Department, University of California at Berkeley, March 1992.
 40. [Khatib87] O. Khatib. *A Unified Approach for Motion and Force Control of Robot Manipulators: The*
-

-
- Operational Space Formulation*. IEEE Journal of Robotics and Automation, vol RA-3, No. 1, 1987.
41. [Klafter89] Richard D. Klafter, Thomas A. Chmielewski, and Michael Negin. *Robotic Engineering: An Integrated Approach*. Prentice-Hall International, 1989.
 42. [Kornegay91] K. T. Kornegay and R. W. Brodersen. *A VME Based Test Controller Board*. Presented at the International Test Conference, October 1991.
 43. [Ku90] David Ku, and Giovanni De Micheli. *HardwareC - A Language for Hardware Design, Version 2.0*. Technical Report CSL-TR-90-419, Stanford University.
 44. [Lauwereins90] Rudy Lauwereins, Marc Engels, Jean Peperstratete, Eric Steegmans, and Johan Van Ginderdeuren. *GRAPE: A CASE Tool for Digital Signal Parallel Processing*. IEEE ASP Magazine, April 1990.
 45. [Lee87] Edward A. Lee and David G. Messerschmitt. *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing*. IEEE Transactions on Computers, January 1987.
 46. [Lee89] E. A. Lee, E. Goei, H. Heine, W. Ho, S. Bhattacharyya, J. Bier, and E. Guntvedt. *GABRIEL: A Design Environment for Programmable DSPs*. IEEE Design Automation Conference, Las Vegas, June 1989.
 47. [Lee90] E. A. Lee, and J. C. Bier. *Architectures for Statically Scheduled Dataflow*. Journal of Parallel and Distributed Computing, Academic Press Inc., vol 10, 1990.
 48. [Lee91] S. Lee. *Implementation of Process Domain in Ptolemy*. Private Communications, EECS Department, U. C. Berkeley.
 49. [Ling87] Y. L. C. Ling, K. W. Olson, P. Sadayappan, and D. E. Orin. *A Layered Restructurable VLSI Architecture for Robotic Control*. IEEE International Conference on Computer Design, 1987.
 50. [Lu92] Kathy Lu. *Software for Front-End Processing for Speech Recognition*. M.S. work in progress - private communications, EECS Department, U. C. Berkeley.
 51. [LWP] SUN Microsystems. *Light Weight Process Library Manual*.
 52. [LYMPH] Berkeley Robotics Group. *LYMPH Multiprocessor Architecture for Robot Control*. Private communication.
 53. [MacDougall75] M. H. MacDougall. *System Level Simulation*. Digital System Design Automation: Languages, Simulation, and Data Base, Computer Science Press, 1975.
 54. [MCC91] *VHDL System Release 3.2.2 Documentation*. Microelectronics and Computer Technology Corporation, 1990.
 55. [Messerschmitt84] David G. Messerschmitt. *A Tool for Structured Functional Simulation*. IEEE Journal on Selected Areas in Communications, January 1984.
 56. [Micheli90] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. *The Olympus Synthesis System*. IEEE Design & Test of Computers, October 1990.
 57. [Narasimhan88] S. Narasimhan, D. Siegel, and J. M. Hollerbach. *Condor: A Revised Architecture for Controlling the Utah-MIT Hand*. IEEE International Conference on Robotics and Automation, April 1988.
 58. [Press88] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing (chapter 15)*. Cambridge University Press, 1988.
 59. [Octtools91] Berkeley CAD Group. *OCTTOOLS Reference Manuals*. EECS Department, U. C. Berkeley, 1991.
 60. [Ptolemy91] *Almagest: Ptolemy User's Manual*. Electronics Research Laboratory, University of California, Berkeley.
 61. [Rabaey86] J. Rabaey, S. Pope, and R. W. Brodersen. *An Integrated Automatic Layout Generation System for DSP Circuits*. IEEE Transactions on CAD, July 1985.
 62. [Rabaey88] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, and F. Cathoor.. *Cathedrall: A Synthesis System for Multiprocessor DSP Systems*. Silicon Compilation, Addison Wesley, 1988.
 63. [Rabaey91] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. *Fast Prototyping of Datapath-Intensive Architectures*. IEEE Design & Test of Computers, June 1991.
 64. [Racal] Racal-Redac Inc. *VISULA-PLUS User's Guide*.
 65. [Richards] Brian Richards. *SDL Language Syntax*. LAGER-IV Manuals, U. C. Berkeley.
-

-
66. [Schwetman86] H. D. Schwetman. *CSIM: A C-based, Process-Oriented Simulation Language*. Proceedings of the 1986 Winter Simulation Conference.
 67. [Shung89] 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 of Integrated Circuits and Systems, April 1991.
 68. [Shung91] C. S. Shung, R. Jain, K. Rimey, E. Wang, M. B. Srivastava, B. C. Richards, E. Lettang, S. K. Azim, P. N. Hilfinger, J. M. Rabaey, and R. W. Brodersen. *An Integrated CAD System for Algorithm-Specific IC Design*. Proceedings of the 22nd Hawaii International Conference on System Science, January 1989.
 69. [Ruetz86] P. A. Ruetz, R. Jain, C-S Shung, J. M. Rabaey, G. M. Jacobs, and R. W. Brodersen. *Automatic Layout Generation of Real-Time Digital Image Processing Circuits*. Proceedings of CICC, May 1986.
 70. [SPOX] Spectron Microsystems. *SPOX User's Guide*.
 71. [Sriram92] S. Sriram. *Design of Multi-Processor Board for the Ordered-Memory Access Architecture*. Private Communications, EECS Department, U. C. Berkeley, 1992.
 72. [Srivastava91a] M. B. Srivastava, and R. W. Brodersen. *High-Level Mixed-Mode System Simulation in VHDL*. Proceedings of the VHDL User's Group Spring 1991 Conference, April 1991.
 73. [Srivastava91b] M. B. Srivastava, and R. W. Brodersen. *Rapid-Prototyping of Hardware and Software in a Unified Framework*. Proceedings of ICCAD, November 1991.
 74. [Srivastava92] M. B. Srivastava, T. I. Blumenau, and R. W. Brodersen. *Design and Implementation of a Robot Control System Using a Unified Hardware-Software Rapid-Prototyping Framework*. To be presented at ICCD, October 1992.
 75. [Stankovic88] J. A. Stankovic. *Real-Time Computing Systems: The Next Generation*. In "IEEE Tutorial on Hard Real-Time Systems", IEEE press, 1988.
 76. [Stölzle91] Anton Stölzle. *A Real Time Large Vocabulary Speech Recognition System*. Ph. D. Thesis, EECS Department, U. C. Berkeley, December 1991.
 77. [Stone91] Kevin Stone. *Mapping Combination Logic to PALs Using MIS-II*. EE199 Project, EECS Department, U. C. Berkeley, December 1991.
 78. [Strom91] R. E. Strom, D. F. Bacon, A. P. Goldberg, A. Lowry, D. M. Yellin, and S. A. Yemini. *HERMES: A Language for Distributed Computing*. Prentice-Hall, 1991.
 79. [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.
 80. [Sun92a] J. S. Sun and R. W. Brodersen. *System Module Interface Design in SIERA*. Under preparation.
 81. [Sun92b] J. S. Sun. *Event Graph Policy for ALOHA*. EECS Department, U. C. Berkeley.
 82. [Sutherland89] Ivan E. Sutherland. *Micropipelines*. Communications of the ACM, June 1989.
 83. [Thomas83] D. E. Thomas, and G. W. Leive. *Automating Technology Relative Logic Synthesis and Module Selection*. IEEE Transactions on CAD of Integrated Circuits and Systems, April 1983.
 84. [Thon89] Lars Thon, Ken Rimey, Brian Richards, and Lars Svensson. *From C to Silicon with LagerIV*. Presented at the IEEE/ACM Workshop on Silicon Compilation and Module Generation, Long Beach, May 1989.
 85. [Thon92] L. Thon and R. W. Brodersen. *From C to Silicon*. Custom Integrated Circuit Conference, May 1992.
 86. [THOR] THOR Simulator Reference Manual, Electrical Engineering Department, Stanford University.
 87. [TMS320C30] Texas Instruments. *Third Generation TMS320 User's Guide*.
 88. [VxWorks] Wind River Systems. *VxWorks Programmer's Guide*.
 89. [Ward85] Paul T. Ward, and Stephen J. Mellor. *Structured Development for Real-Time Systems*. Yourdon Press, 1985.
 90. [Whitcomb92] Gregg Whitcomb. *BLIS Reference Manual*. EECS Department, University of California, Berkeley, 1992.
 91. [Whitney82] D. E. Whitney. *Quasi-Static Assembly of Compliant Supported Rigid Parts*. ASME Journal of Dynamic Systems, Measurement and Control, vol 104:65-77, 1982.
 92. [Whitney] D. E. Whitney. *Historical Perspective and State of the Art in Robot Force Control*. Interna-
-

tional Journal of Robotics Research, 6(1):3-14.

93. [Yu91] Robert Yu. *PLDS: Prototyping in LAGER Using Decomposition and Synthesis*. M. S. Report, EECS Department, U. C. Berkeley, May 1991, ERL Memorandum No. UCB/ERL M91/53.
94. [Zhou91] W. Zhou, and H. Carter. *AnaVHDL: Mixed-Mode Simulation Using VHDL*. Proceedings of the VHDL Users' Group Spring 1991 Conference, April 1991.
-



APPENDIX A: VHDL PROCESS N/W PACKAGE

The MSGPACK_ *type* VHDL package template for the simulation of a system described according to the process network model of Chapter 5 is described here. The package header as well as the body are contained in a single file called msg_<type>.vhd. The package defines MSG_PORT_<type>, which is a resolved VHDL record data type. A process in the process network description is written in VHDL with ports of type MSG_PORT_<type>, where <type> is textually substituted by a VHDL native or user-defined data type. A port of this data type has six sub-fields: D is the actual data of type <type>, C is a record type for doing operations on a queue using two-wire handshake, S is a record type for making queue status visible to a process (useful for simulation output), and fields T, L0 and L1 are used by the resolution function *mrf*. Note that this structure does not imply anything about the actual implementation of the system - its only purpose is to emulate the port and channel based communication of the process network in VHDL.

A VHDL entity called MSG_PROC_<type> is defined to implement the FIFO channel - an instance of this component is made corresponding to every channel. The entity has an input port and an output port which are connected to the output port of the sender process and the input port

of the receiver process respectively. The entity also has two generics (VHDL parameters): `qtype` for the data type, and `qlength` for the buffer depth. The architecture of the `MSG_PROC_<type>` entity is based on a single process that asynchronously accepts requests at the two ports and updates a locally maintained data buffer. The requests specify a data sample to be put into the buffer, or obtained from the buffer, using one of the four possible protocols - blocking, overwrite/previous, ignore, and error - as explained in Chapter 5.

The resolution function `mrf` implements the actual handshaking needed to realize the channel based communication. It uses the `L0` and `L1` fields of the `MSG_PORT_<type>` signals to disambiguate the driver connected to the receiver or sender process from the driver connected to the queue entity. This is complicated by the fact that the connection may be through a series of port-signal associations across a hierarchy. The algorithm is based on maintaining the distances from the two ends in `L0` and `L1` respectively - thus `L0` indicates the distance from the sender end, and `L1` from the receiver end.

In addition to the data type `MSG_PORT_<type>`, the package also defines several access functions, such as `msgsend` and `msgrecv`, using which a VHDL process accesses the ports of type `MSG_PORT_<type>`. These access functions encapsulate the details of the underlying handshake protocol based on VHDL signals.

`msg_<type>.vhd`

```
--
-- This is a package to implement message queues (channels) of data
-- type <type> for my process network. Only single reader or writer are
-- allowed to allow general arbitration schemes. The queue is modelled
-- as a separate process with one input port and one output port.
-- To implement multiple readers or multiple writes, appropriate
-- multiplexor or demultiplexor processes with the desired
-- arbitration schemes need to be written.
--
-- User processes have the following procedures available to them:
--
--   msgsend(port, data, mode)
--       mode = MSG_BMODE      block on full
--             MSG_IMODE       ignore on full
--             MSG_EMODE       error on full
```

```

--          MSG_OMODE          overwrite on full
--  msgrecv(port,data,mode)
--          mode = MSG_BMODE    block on empty
--          MSG_IMODE          ignore on empty
--          MSG_EMODE          error on empty
--          MSG_LMODE          latest on empty
--
-- A queue process has parameters called
--  qtype = SYNC or ASYNC or INF_ASYNC
--          SYNC : synchronous (unbuffered) channel
--          ASYNC : asynchronous (buffered) channel with finite storage
--          INF_ASYNC : asynchronous (buffered) channel with infinite storage
--  qlength >=1 (makes sense only if qmode = ASYNC)
--
package MSG_<type> is
  -- port protocols
  type MSG_MODE is (MSG_BMODE, MSG_EMODE, MSG_IMODE, MSG_OMODE,
    MSG_LMODE);
  -- channel type
  type MSG_TYPE is (SYNC, ASYNC, INF_ASYNC);
  type MSG_PORT_TYPE is (MSG_NC, MSG_MOPORT, MSG_MIPORT, MSG_SIPORT,
    MSG_SOPORT);
  subtype MSG_DATA is <type>;
  type MSG_MODE_VECTOR is array(NATURAL range <>) of MSG_MODE;
  type MSG_DATA_VECTOR is array(NATURAL range <>) of MSG_DATA;
  type MSG_CONTROL is record
    mode: MSG_MODE;          -- queue operation mode
    req: BIT;                -- request an operation on the queue
    ack: BIT;                -- acknowledgement from the queue
  end record;
  type MSG_STATUS is record
    full: BIT;              -- is the queue full ?
    empty: BIT;             -- is the queue empty ?
    rendezvous: BIT;        -- is the other side ready ?
    count: NATURAL;         -- number of packets in the queue
    capacity: INTEGER;      -- capacity of the queue
  end record;
  type MSG_PORT_<type>_1 is record
    D : MSG_DATA;
    C : MSG_CONTROL;
    S : MSG_STATUS;
    T : MSG_PORT_TYPE;
    LO: NATURAL;
    LI: NATURAL;
  end record;
  type MSG_PORT_<type>_1_ARRAY is array (NATURAL range <>)
    of MSG_PORT_<type>_1;

  -- data type of the message ports
  function mrf(v:MSG_PORT_<type>_1_ARRAY) return MSG_PORT_<type>_1;
  subtype MSG_PORT_<type> is mrf MSG_PORT_<type>_1;

  -- functions to access the ports for channel operations
  procedure msgsend(d: in <type>; signal p: inout MSG_PORT_<type>;
    f: out BOOLEAN; m: in MSG_MODE := MSG_BMODE);

```

```

procedure msgrecv(d: out <type>; signal p: inout MSG_PORT_<type>;
  f: out BOOLEAN; m: in MSG_MODE := MSG_BMODE);
procedure msg_init_port(signal p: inout MSG_PORT_<type>;
  t: MSG_PORT_TYPE; qlength: INTEGER := -1);
function msgfull(signal p: in MSG_PORT_<type>) return BOOLEAN;
function msgempty(signal p: in MSG_PORT_<type>) return BOOLEAN;
function msgrendezvous(signal p: in MSG_PORT_<type>) return BOOLEAN;
function msgcount(signal p: in MSG_PORT_<type>) return NATURAL;
function msgcapacity(signal p: in MSG_PORT_<type>) return INTEGER;
procedure print(s: STRING; p: MSG_PORT_<type>);

-- component wrapper for entity to be instantiated for every channel
component MSG_PROC <type>
  generic(qtype: MSG_TYPE; constant qlength: NATURAL := 1);
  port(iport, oport: inout MSG_PORT_<type>);
end component;
end MSG_<type>;

package body MSG_<type> is

  -- resolution function for implementing the handshaking
  function mrf(v: MSG_PORT_<type>_1_ARRAY) return MSG_PORT_<type>_1 is
    variable result : MSG_PORT_<type>_1;
  begin
    -- a queue signal must have two ports connected to it
    -- the allowed combinations are (MSG_MOPORT, MSG_SIPORT) and
    -- (MSG_SOPORT, MSG_MIPORT)
    if v'LENGTH = 1 then
      --print("assert: v(0) =", v(0));
      result := v(0);
      if v(0).lo>0 then result.lo:=v(0).lo+1; end if;
      if v(0).li>0 then result.li:=v(0).li+1; end if;
    elsif v'LENGTH = 2 then
      --print("assert: v(0) =", v(0));
      --print("assert: v(1) =", v(1));
      assert (v(0).lo=0 and v(1).lo=0) or (v(0).li=0 and v(1).li=0)
        -- guard against a MSG_MOPORT->MSG_MIPORT combination
        report "bad queue port combination: MOPORT->MIPORT"
          severity FAILURE;
      if (v(0).lo/=0 or v(1).lo/=0) then
        -- a MO-->SI scenario
        if (v(0).lo=0) then
          -- v(1)->v(0)
          result.lo := v(1).lo+1;
          result.d := v(1).d;
          result.c := (v(1).c.mode, v(1).c.req, v(0).c.ack);
          result.s := v(0).s;
        elsif (v(1).lo=0) then
          -- v(0)->v(1)
          result.lo := v(0).lo+1;
          result.d := v(0).d;
          result.c := (v(0).c.mode, v(0).c.req, v(1).c.ack);
          result.s := v(1).s;
        elsif (v(0).lo-v(1).lo=2) then
          -- v(1)->v(0)
          result.lo := v(1).lo+1;
        end if;
      end if;
    end if;
  end mrf;

```

```

        result.d := v(1).d;
        result.c := (v(1).c.mode,v(1).c.req,v(0).c.ack);
        result.s := v(0).s;
    elsif (v(1).lo-v(0).lo=2) then
        -- v(0)->v(1)
        result.lo := v(0).lo+1;
        result.d := v(0).d;
        result.c := (v(0).c.mode,v(0).c.req,v(1).c.ack);
        result.s := v(1).s;
    else
        assert false
            report "internal error : bug 1 !"
            severity FAILURE;
    end if;
    elsif (v(0).li/=0 or v(1).li/=0) then
        -- a SO-->MI scenario
        if (v(0).li=0) or (v(0).li-v(1).li=2) then
            -- v(0)->v(1)
            result.li := v(1).li+1;
            result.d := v(0).d;
            result.c := (v(1).c.mode,v(1).c.req,v(0).c.ack);
            result.s := v(0).s;
        elsif (v(1).li=0) or (v(1).li-v(0).li=2) then
            -- v(1)->v(0)
            result.li := v(0).li+1;
            result.d := v(1).d;
            result.c := (v(0).c.mode,v(0).c.req,v(1).c.ack);
            result.s := v(1).s;
        else
            assert false
                report "internal error : bug 2 !"
                severity FAILURE;
        end if;
    else
        -- v(0).lo=0 and v(1).lo=0 and v(0).li=0 and v(1).li=0
        assert v(0).T=MSG_NC and v(1).T=MSG_NC and NOW=0 ns
            -- guard against a MSG_SOPORT->MSG_SIPORT combination
            report "bad queue port combination: SOPORT->SIPORT"
            severity FAILURE;
    end if;
else
    assert false
        report "bad number of ports on a queue : exactly 2 allowed"
        severity FAILURE;
end if;
--print("assert: result=",result);
return result;
end mrf;

-- send a message on an ouput port
procedure msgsend(d: in <type>; signal p: inout MSG_PORT_<type>;
    f: out BOOLEAN; m: in MSG_MODE := MSG_BMODE) is
    variable b : BIT;
begin
    b := not p.c.req;
    case m is

```

```

when MSG_BMODE | MSG_OMODE =>
  p.c.req <= b;
  p.d <= d;
  p.c.mode <= m;
  wait until p.c.ack=b;
  f := true;
when MSG_IMODE =>
  if (p.s.full = '0') then
    assert false
    report "queue is full"
    severity NOTE;
    f := false;
  else
    p.c.req <= b;
    p.d <= d;
    p.c.mode <= MSG_BMODE;
    wait until p.c.ack=b;
    f := true;
  end if;
when MSG_EMODE =>
  assert (p.s.full = '0')
  report "queue is full"
  severity ERROR;
  p.c.req <= b;
  p.d <= d;
  p.c.mode <= MSG_BMODE;
  wait until p.c.ack=b;
  f := true;
when MSG_LMODE =>
  assert false
  report "bad msgsend mode MSG_LMODE"
  severity ERROR;
end case;
end msgsend;

-- receive a message on an input port
procedure msgrecv(d: out <type>; signal p: inout MSG_PORT_<type>;
  f: out BOOLEAN; m: in MSG_MODE := MSG_BMODE) is
  variable b : BIT;
begin
  b := not p.c.req;
  case m is
    when MSG_BMODE | MSG_LMODE =>
      p.c.req <= b;
      p.c.mode <= m;
      wait until p.c.ack=b;
      d := p.d;
      f := true;
    when MSG_IMODE =>
      if (p.s.empty = '1') then
        assert false
        report "queue is full"
        severity NOTE;
        f := false;
      else
        p.c.req <= b;

```

```

        p.c.mode <= MSG_BMODE;
        wait until p.c.ack=b;
        d := p.d;
        f := true;
    end if;
    when MSG_EMODE =>
        assert (p.s.empty = '1')
            report "queue is full"
            severity ERROR;
        p.c.req <= b;
        p.c.mode <= MSG_BMODE;
        wait until p.c.ack=b;
        d := p.d;
        f := true;
    when MSG_OMODE =>
        assert false
            report "bad msgrecv mode MSG_OMODE"
            severity ERROR;
    end case;
end msgrecv;

-- initialize a port on start-up
procedure msg_init_port(signal p: inout MSG_PORT_<type>;
    t: MSG_PORT_TYPE; qlength: INTEGER := -1) is
begin
    p.t <= t;
    case t is
        when MSG_MIPORT =>
            p.c.req <= '0';
            p.li <= 1;
        when MSG_MOPORT =>
            p.c.req <= '0';
            p.lo <= 1;
        when MSG_SIPORT | MSG_SOPORT =>
            assert qlength >= 0
                report "queue depth must be >= 0"
                severity error;
            p.c.ack <= '0';
            if qlength=0 then
                -- full as well as empty
                p.s <= ('1', '1', '0', 0, qlength);
            else
                -- empty and not full
                p.s <= ('0', '1', '0', 0, qlength);
            end if;
    end case;
end msg_init_port;

-- find if the channel is full
function msgfull(signal p: in MSG_PORT_<type>) return BOOLEAN is
begin
    return (p.s.full='1');
end msgfull;

-- find if the channel is empty
function msgempty(signal p: in MSG_PORT_<type>) return BOOLEAN is

```

```

begin
  return (p.s.empty='1');
end msgempty;

-- find if the process on the other side is ready for communication
function msgrendezvous(signal p: in MSG_PORT_<type>) return BOOLEAN is
begin
  return (p.s.rendezvous='1');
end msgrendezvous;

-- find the number of data items in the channel
function msgcount(signal p: in MSG_PORT_<type>) return NATURAL is
begin
  return p.s.count;
end msgcount;

-- find the capacity of the channel
function msgcapacity(signal p: in MSG_PORT_<type>) return INTEGER is
begin
  return p.s.capacity;
end msgcapacity;

-- debugging function to print a signal of type MSG_PORT_<type>
procedure print(s:STRING;p:MSG_PORT_<type>) is
  use std.textio.all;
  variable l: LINE;
  type MSG_MODE_TABLE is array(MSG_MODE) of STRING(1 to 9);
  constant msg_mode_names : MSG_MODE_TABLE := (
    "MSG_BMODE", "MSG_EMODE", "MSG_IMODE", "MSG_OMODE", "MSG_LMODE");
  type MSG_PORT_TYPE_TABLE is array(MSG_PORT_TYPE) of STRING(1 to 10);
  constant msg_port_type_names : MSG_PORT_TYPE_TABLE := ("MSG_NC",
    "MSG_MOPORT", "MSG_MIPORT", "MSG_SIPORT", "MSG_SOPORT");
begin
  write(l, s);
  write(l, " (");
  write(l, <type>(p.d));
  write(l, ", (");
  write(l, msg_mode_names(p.c.mode));
  write(l, ","); write(l, p.c.req);
  write(l, ","); write(l, p.c.ack);
  write(l, "), ("); write(l, p.s.full);
  write(l, ","); write(l, p.s.empty);
  write(l, ","); write(l, p.s.rendezvous);
  write(l, ","); write(l, p.s.count);
  write(l, ","); write(l, p.s.capacity);
  write(l, "),"); write(l, msg_port_type_names(p.t));
  write(l, ","); write(l, p.lo);
  write(l, ","); write(l, p.li);
  write(l, ") @ "); write(l, NOW);
  writeline(output, l);
end print;
end MSG_<type>;

library work; use work.MSG_<type>.all;

```

```

-- the entity to be instantiated for every channel
entity MSG_PROC_<type> is
  generic(qtype: MSG_TYPE; constant qlength: NATURAL := 1);
  port(iport, oport: inout MSG_PORT_<type>);
  subtype QINDEX is INTEGER range 0 to qlength;
  type QARRAY is array (QINDEX) of MSG_DATA;
  constant qlength1 : POSITIVE := qlength + 1;
begin
end MSG_PROC_<type>;

architecture BEHAVIOR of MSG_PROC_<type> is
begin
  process
    type MSG_DATA_CELL;
    type pMSG_DATA_CELL is access MSG_DATA_CELL;
    type MSG_DATA_CELL is record
      D : MSG_DATA;
      N : pMSG_DATA_CELL;
    end record;
    variable q : QARRAY;
    variable qh, qt : QINDEX := 0;
    variable qt1 : QINDEX := 1 mod qlength1;
    variable icount, ocount, qcount : NATURAL := 0;
    variable qhp, qtp, tmp : pMSG_DATA_CELL;
    variable b1, b2 : BIT;
    variable eff_qlength : INTEGER;
  begin
    case qtype is
      when ASYNC => eff_qlength := qlength;
      when SYNC => eff_qlength := 0;
      when INF_ASYNC => eff_qlength := -1;
    end case;
    msg_init_port(iport,MSG_SIPORT,eff_qlength);
    msg_init_port(oport,MSG_SOPORT,eff_qlength);
    qhp := new MSG_DATA_CELL;
    qhp.n := null;
    qtp := qhp;
    while true loop
      b1 := not iport.c.req;
      b2 := not oport.c.req;

      -- wait for a request to come in from the sender or the receiver
      wait until iport.c.req=b1 or oport.c.req=b2;
      if iport.c.req=b1 then
        assert icount=0
          report "protocol violation at queue input"
            severity error;
        icount := icount+1;
      end if;
      if oport.c.req=b2 then
        assert ocount=0
          report "protocol violation at queue output"
            severity error;
        ocount := ocount+1;
      end if;
      case qtype is

```

```
-- infinite asynchronous channel protocol (Kahn's model)
when INF_ASYNC =>
  if icount=1 and ocount=1 then
    -- request on both iport and oport
    -- we need not worry about full, empty and count
    iport.s.rendezvous <= '0';
    oport.s.rendezvous <= '0';
    icount := 0;
    ocount := 0;
    iport.c.ack <= not iport.c.ack;
    oport.c.ack <= not oport.c.ack;
    if qhp=qtp then
      -- queue is empty
      oport.d <= iport.d;
    else
      tmp := qhp.n;
      qhp := tmp.n;
      oport.d <= qhp.d;
      qtp.n := tmp;
      qtp := tmp;
      qtp.d := iport.d;
    end if;
  elsif icount=1 then
    -- request on iport but not on oport
    icount := 0;
    iport.c.ack <= not iport.c.ack;
    tmp := qtp;
    qtp := new MSG_DATA_CELL' (iport.d, null);
    tmp.n := qtp;
    qcount := qcount+1;
  else
    -- request on oport but not on iport
    if qcount=0 then
      -- queue is empty
      iport.s.rendezvous <= '1';
    else
      ocount := 0;
      oport.c.ack <= not oport.c.ack;
      tmp := qhp;
      qhp := qhp.n;
      deallocate(tmp);
      oport.d <= qhp.d;
      qcount := qcount-1;
    end if;
  end if;

-- finite asynchronous channel protocol (real-life)
when ASYNC =>
  if icount=1 and ocount=1 then
    -- request on both iport and oport
    -- we need not worry about full, empty and count
    iport.s.rendezvous <= '0';
    oport.s.rendezvous <= '0';
    icount := 0;
    ocount := 0;
```

```
iport.c.ack <= not iport.c.ack;
oport.c.ack <= not oport.c.ack;
if qh=qt then
  -- queue is empty
  -- could have done
  --   oport.d <= iport.d;
  -- but then qh and qt would have remained stationary
  -- which would be a hassle to keep track during debugging
  -- first push
  qt := (qt+1) mod qlength1;
  q(qt) := iport.d;
  -- then pop
  qh := (qh+1) mod qlength1;
  oport.d <= q(qh);
else
  -- the queue may be full, therefore
  -- first pop to the output
  qh := (qh+1) mod qlength1;
  oport.d <= q(qh);
  -- and then push from the input
  qt := (qt+1) mod qlength1;
  q(qt) := iport.d;
end if;
elsif icount=1 then
  -- request on iport but not on oport
  qt1 := (qt+1) mod qlength1;
  if qh=qt1 and iport.c.mode /= MSG_OMODE then
    -- queue is full and mode is blocking
    -- block
    oport.s.rendezvous <= '1';
  else
    -- queue is not full or the mode is overwrite
    if qh/=qt1 then
      -- queue is not full
      qt := qt1;
      qcount := qcount+1;
    end if;
    icount := 0;
    iport.c.ack <= not iport.c.ack;
    q(qt) := iport.d;
  end if;
else
  -- request on oport but not on iport
  if qh=qt and oport.c.mode /= MSG_LMODE then
    -- queue is empty and mode is blocking
    -- block
    iport.s.rendezvous <= '1';
  else
    -- queue is not empty or the mode is latest
    if qh/=qt then
      -- queue is not empty
      qh := (qh+1) mod qlength1;
      qcount := qcount-1;
    end if;
    ocount := 0;
    oport.c.ack <= not oport.c.ack;
```

```
        oport.d <= q(qh);
    end if;
end if;

-- synchronous channel protocol (rendezvous like ADA/OCCAM)
when SYNC =>
    if icount=1 and ocount=1 then
        -- we have a rendezvous !
        icount := 0;
        ocount := 0;
        oport.d <= iport.d;
        iport.c.ack <= not iport.c.ack;
        oport.c.ack <= not oport.c.ack;
        iport.s.rendezvous <= '0';
        oport.s.rendezvous <= '0';
    elsif icount=1 then
        -- producer is ready first
        oport.s.rendezvous <= '1';
    else
        -- consumer is ready first
        iport.s.rendezvous <= '1';
    end if;
end case;
-- update the status fields
if (qcount=0) then
    -- queue is empty
    if iport.s.empty/= '1' then iport.s.empty <= '1'; end if;
    if oport.s.empty/= '1' then oport.s.empty <= '1'; end if;
else
    -- queue is not empty
    if iport.s.empty/= '0' then iport.s.empty <= '0'; end if;
    if oport.s.empty/= '0' then oport.s.empty <= '0'; end if;
end if;
if (qcount=qlength) then
    -- queue is full
    if iport.s.full/= '1' then iport.s.full <= '1'; end if;
    if oport.s.full/= '1' then oport.s.full <= '1'; end if;
else
    -- queue is not full
    if iport.s.full/= '0' then iport.s.full <= '0'; end if;
    if oport.s.full/= '0' then oport.s.full <= '0'; end if;
end if;
    if iport.s.count/=qcount then iport.s.count <= qcount; end if;
    if oport.s.count/=qcount then oport.s.count <= qcount; end if;
end loop;
end process;
end BEHAVIOR;
```

APPENDIX B: THE ASSYS SOFTWARE UTILITIES

Assys is a set of libraries and programs that together provide the run-time software environment for systems designed in SIERA using the design methodology described here. This appendix describes the file organization of *assys*, and also presents an example of using it.

The root directory of *assys*, currently located in `~siera/shared/ass`, is organized as following:

- a. `cofflib`
A low-level library to manipulate COFF object files (used by DSP32C and TMS320C30)
- b. `coffutil`
Wrapper functions for UNIX and VxWorks to load a COFF file section by section.
- c. `include`
Header files common to all processor modules. The two important header files are *assCommon.h* and *assArch.h* which define basic macros and data structures for the system architecture.
- d. subdirectories for individual processor modules: `L3_c30_spoX`, `L2_m68k_vw`
There is one subdirectory for every processor module. The naming convention is `<layer number>_<processor name>_<kernel name>`. Thus `L3_c30_spoX` refers to the TMS320C30 based layer 3 processor module running the SPOX kernel. Inside each of these directories, subdirectories `src`, `include`, `lib`, `obj`, and `bin` contain the sources, headers, library files, object files, and executable binaries respectively.

-
- e. frontend
This subdirectory contain various frontend utilities that run on the workstation. This includes RPC servers, and client stubs, for providing terminal I/O and UNIX-like services.
 - f. vwutil
This contains extensions to the VxWorks kernel.

Using Assys

Following are the steps involved in using *assys* with a system made using SIERA. The following description assumes a MC68020 running VxWorks as the layer 2 processor, and a SPARC based workstation.

- a. Put `~siera/sun4/bin` in the UNIX path.
 - b. Run the following under X on the workstation:


```
unixEmulationWindow &  
ioConsoleServer &
```
 - c. Rlogin to the layer 2 VxWorks processor (say vw):


```
rlogin vw
```
 - d. Assuming that the boot parameters are set correctly, the current directory would be `~vwboot`. Type the following:


```
< vwsiera  
iam "your_login_name"  
nfsAuthUnixSet("zion", your_user_id, your_group_id, 0)  
unixEmulation_init("name of workstation in step b")  
ioConsole_init("name of workstation in step b")  
cd "your application directory"
```
 - e. Now the system is initialized by loading the SAIL file (described in Chapters 6 and 7).


```
assSysInit("SAIL file name")
```
 - f. Then the object files are loader into each processor module in layer 3 using the following:


```
assL3ProcAppLoadAndRun("processor name", "object file name")
```
 - g. To restart:


```
assShutdown(the_system)  
assInitialize(the_system)
```


and then the same as step f.
-

APPENDIX C: SIERA SOFTWARE ORGANIZATION

Most of the tools, libraries, example designs, and software mentioned in this thesis are available on-line on the *BroderSuns* workstation cluster in the EECS Department. This appendix is a guide to all this SIERA related material and describes what is available and where is it located. Before we delve into the actual details there is a general disclaimer that applies to all the following description: the SIERA system, and particularly its organization is in a state of continuous evolution. Therefore whatever is described here is really a snapshot as of writing this thesis - things will inevitably continue to change.

Following is a run-down of what is available. In addition, some tools and libraries are also needed from *~lager* and *~octtools* - these include *DMoct* and *MIS-II*.

I. *~siera*

This directory will be the ultimate repository of all SIERA related packages. At present it is still in a nascent state. The files and directories in it that are relevant to the users of *siera* are enumerated below:

-
- a. `siera.cshrc`
File to be sourced by users of SIERA - it sets up useful environment variables.
 - b. `sun4/lib/siera`
File containing paths for *DMoct* and other tools using the *GetPath* mechanism.
 - c. `sun4/bin`
Contains the various tools needed by a user of SIERA on a SPARC-based machine.
 - d. `shared/cadtools/oct2rinf`
Source code for the *oct2rinf* layout-generator - this is the version that should be used and not the one available in the LAGER release.
 - e. `shared/cadtools/pcb_tools`
Source code for *pfp*, *psg* and some other structure-processors/layout-generators for PCB generation. *pfp* and *psg* were described in Chapter 3.
 - f. `shared/cellib/hardware/packages`
PCB level package library.
 - g. `shared/cellib/hardware/packages-racal`
PCB level package library specific to Racal
 - h. `shared/cellib/hardware/leafcells`
Library containing SDL and THOR files for individual chips.
 - i. `shared/cellib/hardware/modules`
Library containing SDL files for board-level reusable modules.
 - j. `shared/cellib/hardware/boards`
Contains files for four example boards mentioned in the thesis - the robot controller board, the robot peripheral board, the speech grammar and front-end processing board, and the MC96002 multi-processor board. The directories contain all the board-specific SDL files as well as files for PLDS and the final board design database.
 - k. `ass`
Contains run-time software libraries and modules for the various processors supported in SIERA.
 - l. `ass/include`
Header files describing the basic data structures used in SIERA.
 - m. `ass/L2_m68k_vw`
Various packages and software modules for running on a MC68K processor running VxWorks.
 - n. `ass/L3_c30_spox`
Packages and software modules for running on a TMS320C30 processor running SPOX.
 - o. `ass/cofflib`
Package for manipulating COFF object files.
 - p. `ass/coffutil`
Package for loading COFF format files.
 - q. `ass/frontend`
Various packages and utilities for providing front-end support to the SIERA software.
-

r. `ass/c30util`
TMS320C30 dis-assembler.

II. `/usr/tools/commercial/vxworks/vw`

This directory contains the latest version of the VxWorks kernel that is needed by the software in `~siera`.

III. `/usr/tools/commercial/vxworks/vwboot`

This directory contains the start-up file that needs to be loaded in to VxWorks for running SIERA software. The file is called `vwsiera` and an example of using it is in the file `vwmb.s`.

IV. `/usr/tools/commercial/vxworks/vw.vdi`

The VDI package for DSP32C module.

V. `/usr/tools/commercial/spox`

This directory contains the installation of the SPOX kernel for the TMS320C30 modules. The important directories are *include* (contains the header files) and *lib*.

VI. `/usr/tools/dsp/{dsp32,tms3x4x}`

These directories contain the compiler and simulators for use with the DSP32C and TMS320C30 processors respectively.

VII. `/usr/tools/commercial/mcc-vhdl/vhdl-3.2.2-sun4`

The VHDL simulator from MCC

VIII. `/usr/tools/commercial/mcc-vhdl/vhdl-lib`

Various VHDL packages mentioned in this thesis, and other useful locally developed packages for work with MCC VHDL.

