AMD

MACH™ Casebook

1991

MACH™ Casebook

1991

Advanced
Micro
Devices

MACH™ DESIGN CASEBOOK

A STUDY OF MACH IMPLEMENTATION IN SYSTEM DESIGN

901 Thompson Place
P.O. Box 3453
Sunnyvale, CA 94088

TEL: 408-732-2400
TWX: 910339-9280
TELEX: 34-6306
TOLL FREE: 800-538-8450

APPLICATIONS HOTLINE: 800-222-9323

Advanced Micro Devices reserves the right to make changes in specifications at any time and without notice. The information furnished by Advanced Micro Devices is believed to be accurate and reliable. However, no responsibility is assumed by Advanced Micro Devices for its use, nor for any infringements of patents or other rights of third parties resulting from its use. No license is granted under any patents or patent rights of Advanced Micro Devices.

MACH™ is a trademark and PAL® and PALASM® are registered trademarks of Advanced Micro Devices, Inc.

OrCAD/SDT™ is a trademark of OrCAD Incorporated

IBM® is a registered trademark of International Business Machines Corporation

TABLE OF CONTENTS

Preface

PREFACE

This casebook uses a complete logic state analyzer (LSA) design to illustrate a bit-slice design methodology. The focus is on several design segments that illustrate the considerations and tradeoffs associated with MACH-device designs and demonstrate methods you use repeatedly to implement the complete LSA.

Important: Designs that require up to 70% of MACH-device resources can be achieved with very little effort. The design in this case study shows that MACH-device utilizations of greater than 70% can be achieved using various combinations of language syntax and software fitting options. The degree of fit varies from design to design.

Also: The design in this study is a paper design that has been implemented using the PALASM® 4 software; however, it has not yet been implemented in hardware.

Most abbreviations in this casebook are those defined as standard by the IEEE. Abbreviations unique to PALASM 4 and this design are defined at first use.

The reader is assumed to have a working knowledge of programmable logic device (PLD) design, including state-machine and microprocessor design. It is also assumed you are familiar with a logic state analyzer.

Note: This case study does **not** provide steps to follow at the workstation. However, the design files are included on PALASM 4 installation diskettes, in the \PALASM\EXAMPLES\CB directory, as discussed in Appendix A.

Acknowledgements

For their help with the publication of this casebook, Advanced Micro Devices (AMD) would like to thank the following major contributors. AMD also extends thanks to the many other people, too numerous to mention here, for their help with, and commitment to the quality of, this publication.

John Davis
Jana McNulty
Nick Schmitz
Vivian Kong
Gail Tiberi
Joe Walcek

Important: For answers to questions about any information in this casebook, contact a customer support representative at the AMD Applications Hotline: 800-222-9323. The people acknowledged above are not part of the customer support group and are not available to answer questions.

MACH DESIGN CASEBOOK

This casebook uses a complete logic state analyzer (LSA) design, implemented using both MACH 110 and MACH 210 devices, to illustrate the interplay between device attributes, design requirements, and architectural considerations during a typical design cycle. As you read about this design and its implementation, you'll pick up tips and insights regarding

- the MACH devices and supporting design tools,
- an algorithmic flow for system-level designs, and
- the decisions and tradeoffs for logic partitioning, device speed, and pin-out requirements for MACH-device designs.

Techniques presented in this study, and some of the circuits themselves, can be applied to other designs and design tasks. The methodology presented here has three major advantages.

- **Design flexibility**: you can easily assign system functions to minimize the number of chips.

- **Architectural flexibility**: you can realize system functions of varying word widths.

- **Optimization flexibility**: separate data and control domains enable you to independently optimize system features for speed or logic density.

The focus of each major discussion is identified below.[1]

- The design description, 1, provides an overview of the complete LSA design and includes details about its functionality, performance, and density.
- The topics under discussion 2 summarize MACH-device resources, introduce design-tool support, and explore a method to segment the design to evaluate its requirements.
- The architecture of this LSA design, 3, explains how to convert an idea into an overall system description that allows design elements to fall out naturally into appropriate devices.
- The data-flow discussion, 4, focuses on specific design segments to show you how to divide the data flow into subfeatures likely to fit in the selected device initially.
- The control-flow discussion, 5, focuses on specific design segments to show you how to divide control logic into subfunctions likely to fit in the selected chip initially.
- The integration of data and control flows, 6, explores how to connect control-flow and data-flow domains using MACH I/O constructs.
- The tuning summary, 7, introduces tuning tips for this design and summarizes tuning strategies you can use for other designs.

[1] This guide does not illustrate all segments of the LSA design; it focuses on segments that identify basic techniques and considerations for a large design. Design files for all segments of the LSA design are available on the PALASM 4 installation diskettes. Files are identified in the text where appropriate. Appendix A provides a complete list of all files. Refer to the *MACH Design Workbook* for design examples that provide details about resolving specific fitting problems.

- The complete LSA discussion, 8, presents the design in the context of an add-in card implementation.
- The review, 9, retraces information pertinent to this and other designs.
- The discussions in Appendix A describe each file, for this LSA design, included on the PALASM 4 installation diskettes.

1 DESIGN DESCRIPTION

The subject of this study is a 16-bit LSA design.[2] The next figure shows a sample LSA board; the shaded boxes show design segments implemented using MACH chips.

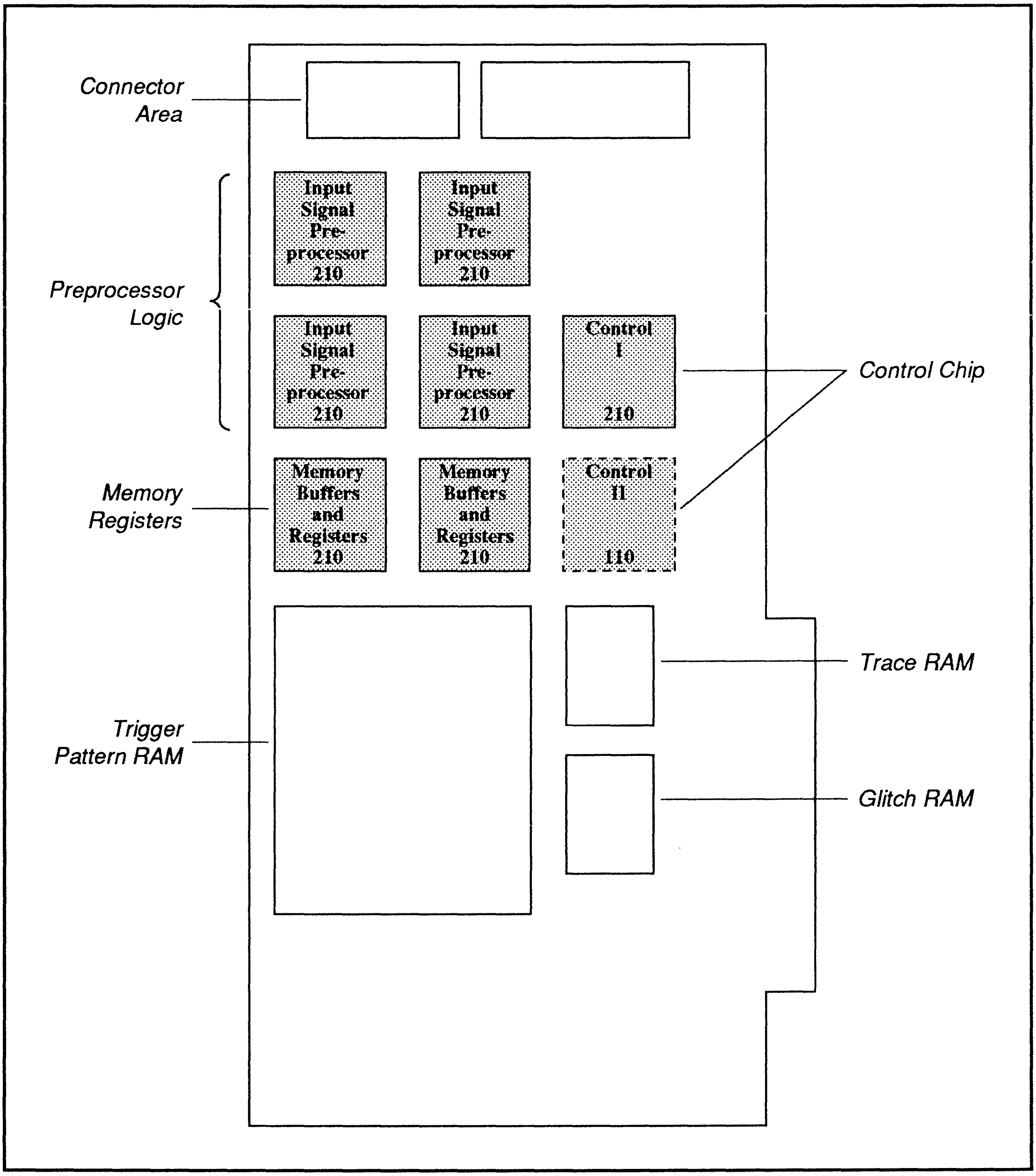
Excluding input buffers and RAM, the LSA functions implemented for this study consist entirely of MACH programmable logic.

- The preprocessor logic requires four MACH 210 chips.
- The memory registers are implemented in two MACH 210 chips.
- The LSA control is contained in a MACH 110 and a MACH 210.
- All optional functions, such as the host interface, keyboard support, and memory control, can be implemented on either a MACH 110 or a MACH 210 chip.

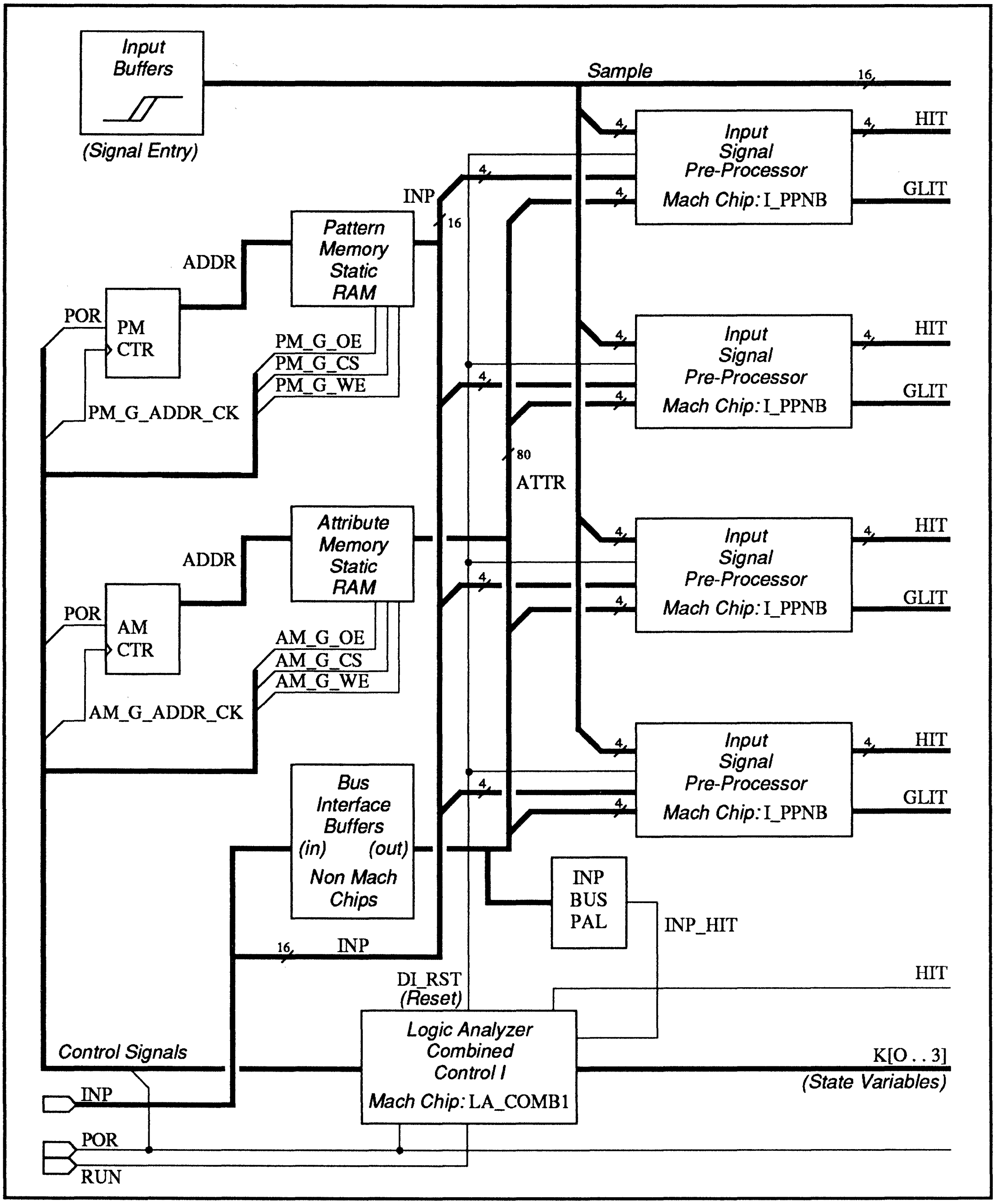
You can implement the input buffers, such as Schmitt triggers, using any external logic you choose. This design provides control lines for static RAM logic. However, you can customize this part of the control logic to use another type of RAM if you choose.

The two-page block diagram that follows the sample board layout presents the LSA in the context of other functional elements, such as RAM, signal input buffers, and input control. Each block is labeled with a function name and the name of the file where the logic is implemented; a description follows the block diagram.

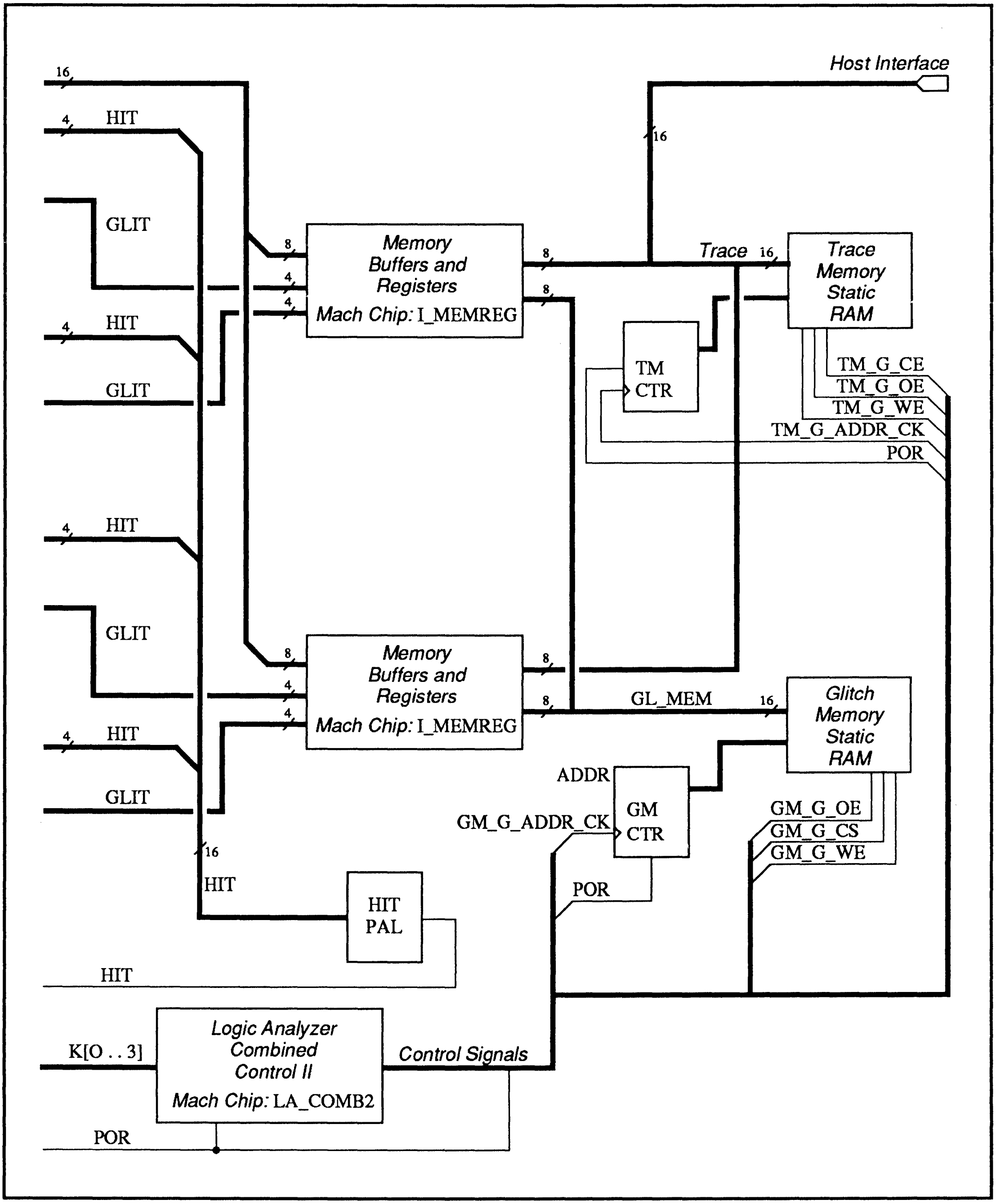
[2] This study presents a paper design that has been implemented using the PALASM 4 software. It has not yet been implemented in hardware.


Connector Area
Preprocessor Logic
Input Signal Pre-processor 210
Input Signal Pre-processor 210
Input Signal Pre-processor 210
Input Signal Pre-processor 210
Control I 210
Control Chip
Memory Registers
Memory Buffers and Registers 210
Memory Buffers and Registers 210
Control II 110
Trace RAM
Trigger Pattern RAM
Glitch RAM

Sample Logic State Analyzer Board


Input Buffers
(Signal Entry)
Sample
16
4
HIT
GLIT
Input Signal Pre-Processor Mach Chip: I_PPNB
INP
16
ADDR
Pattern Memory Static RAM
POR
PM CTR
PM_G_OE
PM_G_CS
PM_G_WE
PM_G_ADDR_CK
80
ATTR
Attribute Memory Static RAM
AM CTR
AM_G_OE
AM_G_CS
AM_G_WE
AM_G_ADDR_CK
Bus Interface Buffers (in) (out) Non Mach Chips
INP BUS PAL
INP_HIT
DI_RST (Reset)
Logic Analyzer Combined Control I Mach Chip: LA_COMB1
Control Signals
K[O..3] (State Variables)
RUN

Left Side: LSA Block Diagram



Right Side: LSA Block Diagram

The input preprocessing chips, which detect edges, glitches, and levels, process inputs in nibble-sized units. The nibbles are multiplexed into byte-sized increments by the memory-input register chips.

During operation, sample data enters the LSA via the input buffers and is checked for specified combinations of levels, edges, and glitches by the input-signal preprocessor chips. All types of signal attributes are checked. However, only attributes specified by a 1 in the attribute mask are included in a comparison for trigger events.

When a match occurs for a selected condition, it is further masked by a pattern from the pattern memory. The pattern allows only selected bits that match trigger conditions to leave the chip as a hit. Glitch data also leaves the chip and is stored in the glitch RAM. Hits are ORed to gate the trigger state machine.

Glitch data that leaves the input-signal preprocessing chips is directed to the glitch-memory static RAM by the memory buffers and register chips. At the same time, the input data on the Sample bus is directed to the trace-memory static RAM. At the end of the trace cycle, trace-memory data is read to the host RAM via the host interface. The memory buffers and register chips are then re-configured to allow the glitch data to be read to the host.

1.1 FUNCTIONALITY

The LSA implemented in this study has the following functions.

- Independent trigger conditions for each bit
- Active-high and active-low trigger options
- Rising-edge and falling-edge trigger options
- Glitch detection
- Eight levels of triggering before trace begins
- Detection logic that can be used in either parallel mode or serial mode

In addition to the usual practice of triggering on a pattern across the entire trigger word, the independent triggering conditions for each bit in this design allow you to trigger on the activity of any single bit. Triggering on the active level of a particular set of signals is fairly standard. Edge triggering and glitch detection can be found on most intermediate-level LSAs; the more sophisticated instruments often allow bit-level triggering for edges and glitches.

The MACH chips in which LSA functions were implemented can be used either in parallel, to increase sampling speed, or in serial, to implement more triggering levels.

1.2 PERFORMANCE

This LSA has a trigger rate of 20 MHz in serial mode. When the triggering modules are configured in parallel, the rate increases to 40 MHz. The limiting factor on the available trigger rate is the number of trigger types included in the design. The access time for fast RAM chips is on the order of 35 ns; the 15 ns propagation delay of the MACH device is not the most critical path.

1.3 DENSITY

This design requires a total of eight chips: seven to implement the heart of the 16-bit LSA and one to implement optional features, such as the keyboard and host interfaces. Additional chips can be added to customize the design for a particular platform. The sample board layout shows the chips placed on a PC add-in board. In this case, the user interface is handled by the host PC.

In this design, one MACH 210 device independently processes each bit of a 4-bit nibble of the input-data stream. Four such chips are used in this design; each has the following functions.

- Store internal triggers.
- Preprocess five attributes per bit: rising-edge, falling-edge, glitch, active-high, and active-low event trigger conditions.
- Detect the occurrence of a trigger event.

Two memory-register chips are used in this design. Each one, implemented in a MACH 210 chip, contains the path-routing and buffer registers used during triggering and tracing. The paths are also used to upload captured data.

The system-control chips contain the state machines that control the trigger and trace operations. They are implemented in both a MACH 110 and a MACH 210. For this particular design, a MACH 210 was chosen for one chip to allow for future growth.

2 DEVICE RESOURCES, DESIGN TOOLS AND PROCESS

Discussions below summarize MACH-device resources and introduce design-tool support. The design process used to evaluate and implement the entire LSA design is also introduced.

2.1 MACH-DEVICE RESOURCES

MACH devices provide high-density programmable logic combined with high pin-count modules to give you high-logic density in an appropriate package for your design. The MACH 110 provides resources in both 15 ns and 20 ns speeds; any or all flip-flops can be implemented as buried macrocells.

The electrical and physical characteristics of the MACH 210 are similar to those of the MACH 110. The MACH 210 provides twice the logic density in the same physical package. This conserves board space and avoids additional delays, which result when signals are driven off-chip across the printed-circuit board and back into another chip. In the MACH 210 device, half of the flip-flops are designated as buried macrocells. Both devices provide up to 38 inputs and up to 32 outputs.

The table below summarizes specifications for all MACH devices.

Device	Pins	Macrocells	Gate Equivalents	Max Inputs	Max Outputs	Max Flip-Flops	Speed (ns)
MACH 1 Family							
MACH 110	44	32	900	38	32	32	15, 20
MACH 120	68	48	1200	58	48	48	15, 20
MACH 130	84	64	1800	70	64	64	15, 20
MACH 2 Family							
MACH 210	44	64	1800	38	32	64	15, 20
MACH 220	68	96	2400	58	48	96	15, 20
MACH 230	84	128	3600	70	64	128	15, 20

2.2 MACH DESIGN TOOLS

Complete support for MACH-device designs is provided through the PALASM 4 software, which includes the tools for entry, compilation and fitting, simulation and documentation, and communication with a device-programmer so you can create and process the design and download JEDEC data to the chip.[3]

Several design-entry methods[4] are supported within the PALASM 4 environment.

- Text-based Boolean-equation descriptions
- Text-based state-machine language descriptions
- Schematic-based designs
- Mixed-mode designs

You begin a schematic-based design from the PALASM 4 software, which automatically invokes OrCAD/SDT™ III with the AMD-supplied MACH library.[5] Schematics are automatically converted to Boolean equations during the compilation process and the resulting PDS file is used to complete remaining processes.

The compilation process includes fitting the design to the MACH device and producing a machine-readable JEDEC file, which represents the fuses to be programmed.[6] Though MACH devices are reprogrammable, simulation is available to help you uncover problems with the design before you build the chip.

[3] Refer to the *PALASM 4 User's* Manual, Chapter 9, for complete details about commands, forms, and options.

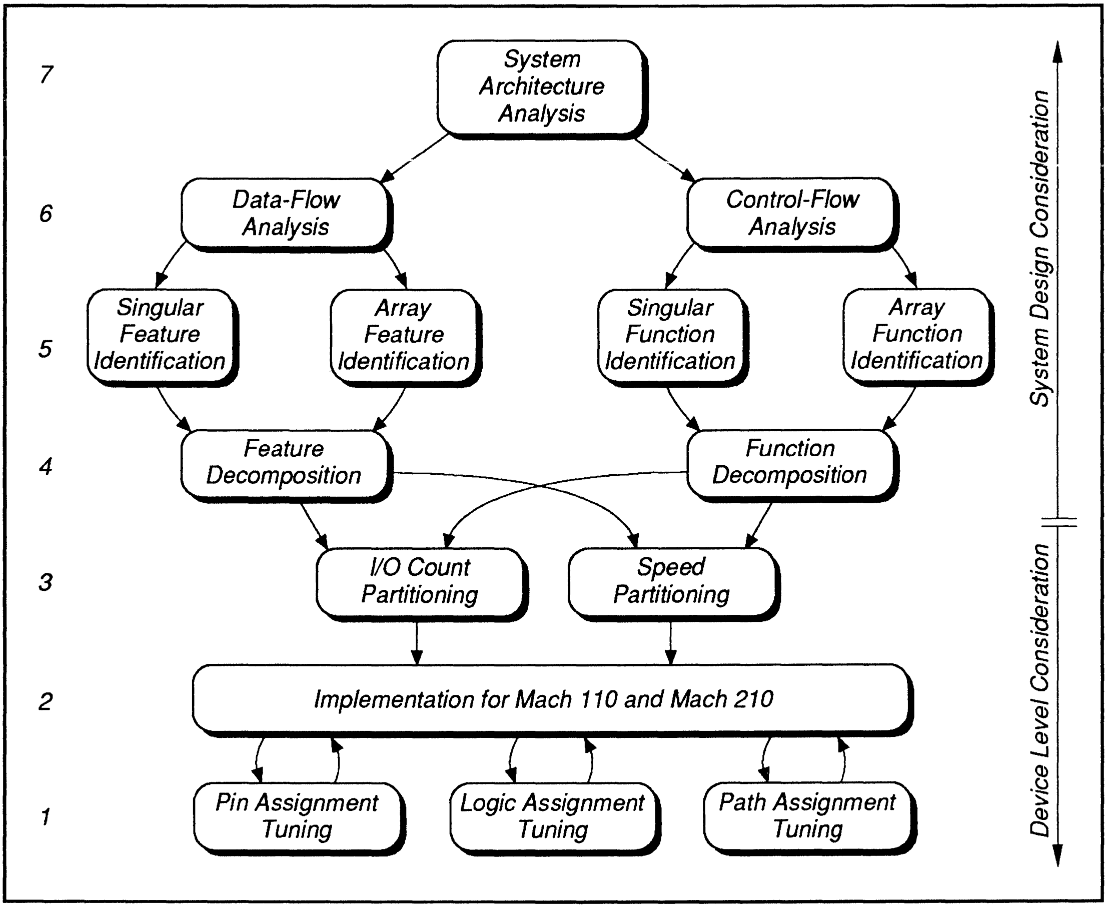
[4] Refer to the *PALASM 4 User's Manual*, Chapter 2, for step-by-step tutorials that guide you through design entry, Chapter 4, for entry strategies, and Chapter 10, for language syntax.

[5] The OrCAD/SDT III software and the PALASM 4 interface to it must be purchased explicitly.

[6] Refer to the *PALASM 4 User's Manual*, Chapter 1, for details about hardware requirements.

2.3 DESIGN PROCESS

The seven-level algorithmic process used for this design helps you reduce the high-level LSA system description into smaller pieces that fit into a single chip. The next figure shows the flow of this approach.


7
System Architecture Analysis
6
Data-Flow Analysis
Control-Flow Analysis
5
Singular Feature Identification
Array Feature Identification
Singular Function Identification
Array Function Identification
4
Feature Decomposition
Function Decomposition
3
I/O Count Partitioning
Speed Partitioning
2
Implementation for Mach 110 and Mach 210
1
Pin Assignment Tuning
Logic Assignment Tuning
Path Assignment Tuning
System Design Consideration
Device Level Consideration

Seven-Level Design Process

As you can see, there are seven vertical levels in the process flow. Boundaries between levels are not sharply defined because the design partitioning process varies with technology and with the system to be implemented.

You begin the process by evaluating the architecture for the entire system and dividing the design into its data-flow and control-flow domains. The left branch of the process flow addresses the data-flow domain; the right branch addresses the control-flow domain.

For each domain, you use splitting, decomposition, and partitioning techniques to identify features and functions that can fit into a single chip. MACH-device resource considerations come into play during the manual design-partitioning phase. Once the chip-sized and smaller functions are isolated, you begin implementation and end the process with tuning. The last three stages are the same for both design domains.

Note: If the design is small enough or simple enough to partition mentally, you start development with device-level considerations.

The tables below identify each level in the design process. A generic discussion of each phase follows. Details and specific LSA design considerations are discussed under topics 3 through 7.

SYSTEM DESIGN CONSIDERATIONS	
7	System Architecture Analysis
6	Data-Flow Analysis / Control-Flow Analysis
5	Feature / Function Identification: Singular & Arrays
4	Feature / Function Decomposition

DEVICE-LEVEL CONSIDERATIONS	
3	Partitioning
2	Implementation
1	Tuning

2.3.1 System Architecture Analysis

When converting ideas to designs, you can start at the conceptual stage and work down to the chip level on paper. The objective here is to identify the most basic elements you need to build the system, called architectural primitives. In general, there are two kinds of architectural primitives: data flow and control flow. Initially, you review and evaluate the list of functions specified for the entire design, then divide the design into data-flow and control-flow domains.

Once this division is complete, the process flow, discussed next, is similar for each domain.[7]

2.3.2 Data-Flow Analysis

The objective during this phase is to reduce the data flow to a set of bit slices, each of which will usually fit into a single MACH chip. Data flow is defined as the sum of all system input paths that move through storage and processing modules. Data-flow features are treated separately from control functions and other parts of the system.

There are two kinds of features in the data-flow domain, which you must identify and isolate.

- Array features
- Singular features

The process can either be informal, where you mark up a block diagram, or formal, where you list singular and array features separately.

[7] Refer to discussion 3 of this casebook for details about the system architecture of this LSA design.

Singular Feature Identification

Singular features are defined as all unique aspects of the data flow that affect a limited number of the data elements, and are usually characterized by a high degree of internal interconnectivity. Serial data-channel functions are an example of singular features in the data flow. A classic example is bus demultiplexing. On some microprocessors, the data bus carries both data and address information at different times. A special, singular data path must be created to route the bits carrying address information to the address registers.

Look for examples of singular functions in your data flow.

Array Feature Identification

The most outstanding trait of the data flow is the large number of arrays. An array feature can have multiple pieces; each of these can be further subdivided during decomposition.

- Each array has specific speed and I/O requirements, which you can match to a particular MACH device.

- Each array represents an element of the LSA you can design once as a bit slice and test for chip fit.

Storage registers and multiplexers are examples of array features in a data flow. The same array can appear repeatedly in a design; however, each instance must have a unique name. Other examples of array features for this LSA design are listed below.

- Rising-edge detector
- Glitch detector
- Multiple-triggering levels

Feature Decomposition

Once you have identified all array and singular features in the data-flow domain, each becomes the root of a tree with data types passing through the feature as leaves. If a particular leaf has special features not shared by all data types, the leaf is expanded as a subtree.

For example, a sign bit on a data bus has features not shared by the magnitude bits. The two modes of use might result in data path additions. The data path tree should be expanded to two leaves.

- One leaf with no sign considerations
- One leaf with sign considerations.

The leaves that result when all data-flow features are expanded are the data-flow architectural primitives.

Once you choose the array elements, it's a simple matter of multiplication to find how much of the array will fit into a single MACH chip. For example, the bit slice for the rising-edge detector is repeated 16 times to produce the detector for one word of the LSA. Singular-feature bit slices usually fit into a chip immediately during implementation because they don't require an entire chip's resources.

Choosing bit slices judiciously helps ensure a fit either as a singular function or as a MACH-based component you can chain. Most digital functions will fit into MACH devices. However, design functions, such as analog functions, not supported in MACH chips must be separated for implementation outside MACH.

2.3.3 Control-Flow Analysis

The control flow is defined as all signals, either input to the system or generated by the system, which direct the flow of data through the system or direct the transformation of data in the system. Data-transforming nodes are part of the control flow though they appear in the data-flow diagram.

The latter distinction is somewhat artificial yet reflects common design practice. The distinction is practical because data-transforming nodes generally have more of the bit-to-bit interconnectivity associated with control than the repeated, independent bit structure associated with data flow.

System functions meeting this control-flow definition should be separated from other system functions during the function-identification phase. Choosing control functions judiciously helps ensure a fit, either as a single function or as a MACH-based component you can chain. The control-flow domain contains two kinds of functions.

- Array functions
- Singular functions

Singular Function Identification

The most outstanding trait of the control flow is the predominance of singular functions, which dominate because control logic usually combines several inputs to create a single output. The resulting output controls either a single node in the data flow or a single path through the data flow. The nodes are unique so an array structure is needed only for parallel processing.

State machines are the primary examples of singular functions in control logic. Each system function becomes a state machine you can reduce to a set of component state machines. The complete list of state machines forms one list of singular functions ready made for implementation in a MACH device. Again, a function can have multiple pieces you further subdivide during decomposition.

Array Function Identification

A control-logic array occurs when data must be converted from one form to another. The conversion process generally involves creating new data, in an arithmetic-logic unit or a parity generator, for example. The key attribute of an array is different input and output data. The data differs either in value or in the number of bits of information contained. The logic that defines a single bit is the logic selected to represent the array.

Since the array functions in control logic generally come from data-flow conversions, they usually consist of both an array and a singular function. Thus, each control-flow array can be decomposed to a pair of control functions.

- The input array typical of data flow
- The output singular function that reduces the number of bits

Function Decomposition

During decomposition, you assign each control operation to the root of a tree and assign specific subfunctions to be performed as part of the operation to the leaves. When a function has subfunctions, the leaf is expanded as needed. In this way, final chip resources are used more efficiently; fitting many small pieces results in less unused space than fitting large pieces.

The final leaves define the control-flow architectural primitives needed to implement the system control. You use the leaves to define the basic elements needed to build the system's control.

2.3.4 I/O and Speed Partitioning

The two process paths become one in this phase, during which you evaluate MACH-device resources and assign basic elements of the design (the control- and data-flow architectural primitives) to MACH chips. Two tasks start this phase.

- Determine I/O count for each bit slice
- Determine speed of each bit slice

You use the I/O requirements to identify functions or features exceeding the device pin-out for further decomposition or assignment to multiple chips. If further decomposition is possible, final chip resources are used more efficiently.

The speed of a critical path can be achieved by redesigning functional properties in stages. Functions in which logic formerly overlapped are spread out into separate logic implementations using this pipelining technique.

During this phase, however, you need only identify areas you think might need reconsideration. Actual optimization occurs during the tuning phase, after the entire design has been implemented and you have a better idea of how much unused space remains.

2.3.5 Implementation

The following steps outline implementation activities.

- Enter each architectural primitive as either a text-based or schematic-based design.

- Compile and fit each architectural primitive and review the MACH report; re-engineer the design if needed.

- Simulate each architectural primitive.

- Merge into one design the architectural primitives you know can fit on a single chip; recompile the combined design.

Entry

You can enter designs as either text or schematic-based information. Initially, you enter each architectural primitive separately.

Tip: It's a good idea to build up hierarchical schematic files for each array feature.

After you compile each hierarchical schematic, and verify statistics in the MACH report, you can create a single schematic that references primary schematics.

For example, if a single architectural primitive requires one eighth of the chip's resources, you create a schematic that references the primitive either six or eight times. Then you compile the six-bit or eight-bit schematic, which allows you to quickly estimate the amount of chip resources actually needed for candidate word widths.

It is a good idea to compile and fit each architectural primitive to ensure it is complete and correct before you merge it with other designs for implementation on a single chip.

Compilation and Fitting

Isolating control from data flow and singular functions from array functions pays off here.

- Control functions, such as state machines, and singular functions often have many internal interconnections.

- Array functions often have few internal interconnections.

Initially, you specify the following mode on the Compilation Options form.

Run mode:	Auto

Then you choose the MACH-fitting option below for each design.

FITTING OPTIONS	
When compiling	Run all until first success

The software uses different fitting-option combinations until the first fit is achieved. Device-utilization statistics in the MACH report for each primitive show resource requirements for that design. This leads directly to the number of chips needed for an array or the number of singular functions you can fit on one chip. Initially, a **good rule** of thumb is to keep chip-resource assignments for storage elements and product terms at or below 70% of the device's available resources.[8] It won't take long to determine whether a given function will fit, and, if it won't, why. When the fit is close, you can often use manual techniques to achieve success.

Simulation

It is a good idea to simulate each architectural-primitive design to ensure it is functionally correct and complete. This is especially important before you merge designs.

Recommendation: Use an auxiliary simulation file instead of the PDS simulation segment. If you enter the design as a schematic, the PDS file is created with a blank simulation segment during compilation, so each time you compile, the segment is erased. If you merge designs, simulation commands are removed from the PDS simulation segment during the merge process.

[8] Designs that require up to 70% of MACH-device resources can be achieved with very little effort. The design in this study shows that MACH-device utilizations of greater than 70% can be achieved using various combinations of language syntax and software fitting options.

Merging Designs

Designs for architectural primitives must be combined to yield preliminary MACH chip implementations. You can merge any number of designs together using the Merge design files command on the PALASM File menu.[9]

Tip: You can merge schematic-based designs with text-based designs using the schematic-based PDS file created automatically during compilation.

During the merge operation, you are advised of any signal-name conflicts so you can change names immediately. This prevents problems caused when the same architectural primitive is being used repeatedly.

After merging two designs, you process the combined design.

- Compile the combined design
- Simulate the combined design

Again, the MACH report identifies resources and other fitting information so you can verify the design fits into the chip. Since singular functions are independent of one another, you can group unrelated functions together to fill a chip.

Suppose the device-utilization statistics in various MACH reports indicate function A requires 40% of a chip, function B requires 20%, and two other functions require 10% total. You can merge the four designs together and resolve signal contention interactively as you proceed. Then you compile the combined design.

[9] Refer to the *PALASM 4 User's Manual*: Chapter 4 provides guidelines for the entry process, Chapter 9 provides details about available commands, forms, and options.

2.3.6 Tuning

During tuning, you optimize the design to achieve the best overall fit using the fewest number of chips. This includes both system fitting and fitting a particular function in a specific chip. Several optimization techniques are provided.

- Use different MACH-fitting options to optimize results and to correct non-optimal path assignments.
- Enable gate splitting to correct non-optimal logic assignments.
- Group functions in a particular MACH block to correct non-optimal logic or pin assignments.

Product-term allocation techniques include product-term steering, where product terms are automatically allocated from adjacent macrocells in a single block, and gate splitting, where product terms can be allocated from non-adjacent macrocells, including those in different blocks. These are useful when a MACH device does not provide sufficient product term resources within a macrocell for the logic being fit. However, when the design fits, product-term allocation is not likely to produce a better fit because the same device resources are required.

Tip: When further decomposition is possible, final chip resources are used more efficiently; fitting many small pieces results in less unused space than fitting large pieces. When a function fits in a chip, gate splitting can actually upset the fit because it must use global wiring channels to make the second pass through the array. The rule in this case is if it fits, don't force it to be better.

Use Different MACH-Fitting Options

After merging designs, you can use different fitting options to squeeze as much logic onto each chip as possible. You can also correct non-optimal path assignments caused when logic-placement decisions block routing paths to functions that must communicate.

Note: It is standard to use the Run all until first success option in the MACH fitting options menu. This option prompts the compiler to try various meaningful combinations of the compilation flags until a first fit is achieved. The descriptions below illustrate how the flags can be chosen manually.

For example, the need for internal connections in control and singular functions is supported in this LSA design by spreading out product terms as they are placed on the chip. You can use the fitting options below to allow room for internal local connections.

```
FITTING OPTIONS
When compiling                    Select one combination
:
Expand small PT spacing?                    Y
```

Array functions can be packed more densely because they lack extensive interconnections. When you compile array functions, you can disable product-term expansion options using the options below.

```
FITTING OPTIONS
When compiling                    Select one combination
:
Expand small PT spacing?                    N
Expand all PT spacing?                      N
```

There is no permanent penalty if you disable expansion options when compiling singular and control functions or enable these options for array functions. You can compile functions both ways to see which provides the best fit. Using the initial design partition can usually maximize the possibility of a fit. The logic impacts which fitting options produce the best result.

Enable Gate Splitting

You can use gate splitting to improve non-optimal logic assignments that can occur when the automatic placement algorithm has inappropriate or inadequate information. This results in logic groups not being optimally placed; large blocks of logic may be placed in areas of the chip that lack sufficient resources. Logic in the corners of the chip can pose a problem.

Also, functions that require more product terms than a MACH device provides within a group of adjacent macrocells can be specified as a composite gate with product terms split between macrocells in non-adjacent locations.

To correct these kinds of problems, you enable gate-splitting on the Logic Synthesis Options form, as follows.

```
Use automatic gate splitting?          Y   Max = 4
```

Intermediate logical functions are created as sets of product terms with the maximum width specified, from 4, the default, to 16, then combined as a single output function. Combining intermediate functions in this way requires additional passes through the logic array so the outputs can be used as inputs. Each additional pass costs an additional propagation delay; wide functions are supported but will be slower.

Gate splitting can be useful in state machines with many conditions for next-state transitions. This condition arises when there are many next states or many variables in the state definition.

You can also use the Group command, discussed next, to improve logic and pin assignments.

Group Functions within a Specific MACH Block

Non-optimal pin assignments can occur when automatic resource allocation causes reduced logic capacity due to wiring congestion. As a result, desired pin-out paths, and paths to pins, may not be realizable.

To improve the logic assignments, you can use the Group command to assign functions to particular blocks in a MACH chip. You must use the appropriate reserved word, MACH_SEG_*block*, as a group name, as shown below.[10]

```
GROUP MACH_SEG_A T[0] T[1] T[2]
```

Certain pins are associated with certain MACH logic blocks. The objective is to place logic as close to the desired pin as possible. You can achieve the objective by moving logic to the block associated with the I/O pins you need.

Singular control-flow functions that appear as data-flow nodes are the most likely candidates for grouping in a specific MACH block. These functions have array attributes, which make them large, and control attributes, which cause lots of internal interconnections. The latter indicates wiring considerations you may need to improve.

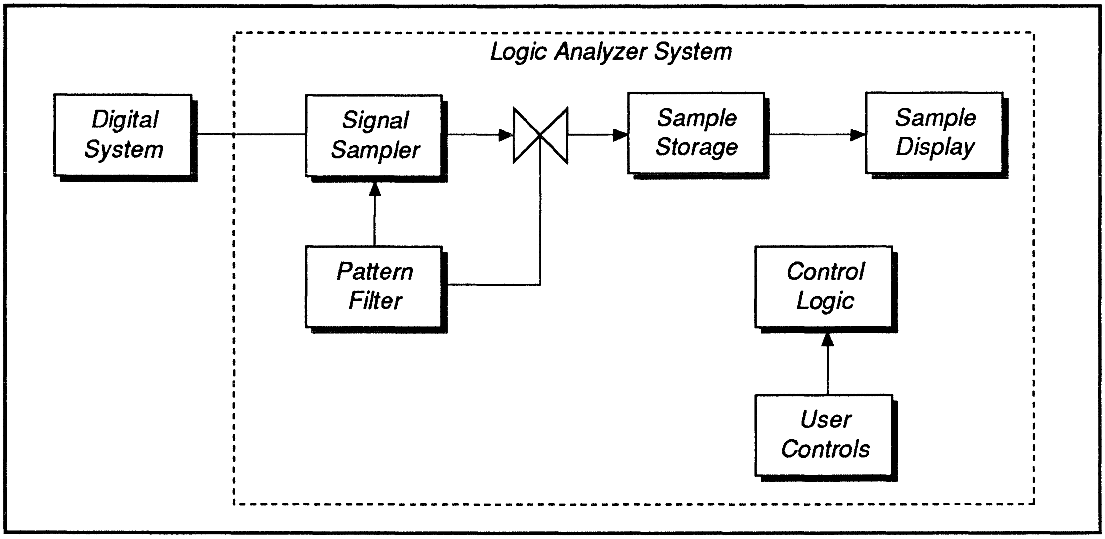
Forcing the function into a certain block allows for better wiring performance in other parts of the chip. This can be useful in the control domain. In this design, however, functions most likely to need this type of tuning were separated. For example, singular control functions were isolated from other control functions. This LSA design does not include Group commands.

[10] Refer to the *PALASM 4 User's Manual*, Chapter 10, for details about MACH_SEG_*block*.

3 LSA SYSTEM ARCHITECTURE

You start the design process with a preliminary system architecture for a logic state analyzer, like the one shown next. This architecture incorporates the minimum requisites of pattern detection and the user interface in the context of a system under test.

You mentally simulate its operation to ensure all the major functions are accounted for before you convert it to a data flow.



Block Diagram: Preliminary System Architecture

For example, the LSA user must specify, on a front panel, the operating mode and identify which patterns to look for. The LSA's run mode implies a supervisory state machine, which enables the two major state machines in the control-logic block: trigger checking and data collection. During trigger checking, the analyzer checks for signal patterns that indicate the start of the specified test phase. When the patterns occur and the Run button is pressed, data collection begins.

The sample board layout and the system block diagram, in discussion 1, show an optimized LSA implementation that results when you apply the seven-level design process, in discussion 2, to a system architecture description. When you start a design, you do not always know the final data-flow logic to convert to MACH chips. Even when you have a complete data flow, it may not be optimally configured for the best MACH implementation. The following discussion begins the LSA design by deriving the system architecture, which allows the seven-level process to proceed in a top-down fashion.

3.1 DERIVING THE LSA ARCHITECTURE

If you look at the functional requirements of the design shown in the previous figure, you begin to see how they determine the overall system architecture. Two major design requirements come to mind.

- High-speed sampling
- Pattern detection

The sample rate cannot appear on a block diagram, so you leave this aspect until the implementation phase. Pattern detection then becomes the most salient architectural feature. It's best to consider functional requirements in the context of normal use when describing the system's architecture.

You begin deriving the architecture by specifying the operating modes and the patterns to look for; these are defined via the front panel by the user. The user interface is the last piece of system architecture. For this, you need two blocks.

- A user input block
- A user feedback display block

3.1.1 LSA Functions and Flow

During data collection, input-sample data must be checked to isolate the occurrence of all trigger patterns. The LSA design in this study allows for multiple triggers rather than just one. The architecture of this design allows data to flow from the signal sampler to the pattern filter, which inhibits data storage until all triggers are detected. Data in the sample-storage area are passed to the sample display and presented to the user. However, this scenario accounts only for samples collected at regular intervals. Data changes between clock edges are called glitches. The following occurs between clock intervals.

- If the data changes and remains changed, the sample clock strobes changed data on the next pass.
- If the data changes and doesn't remain changed, the sample clock does not record a change.

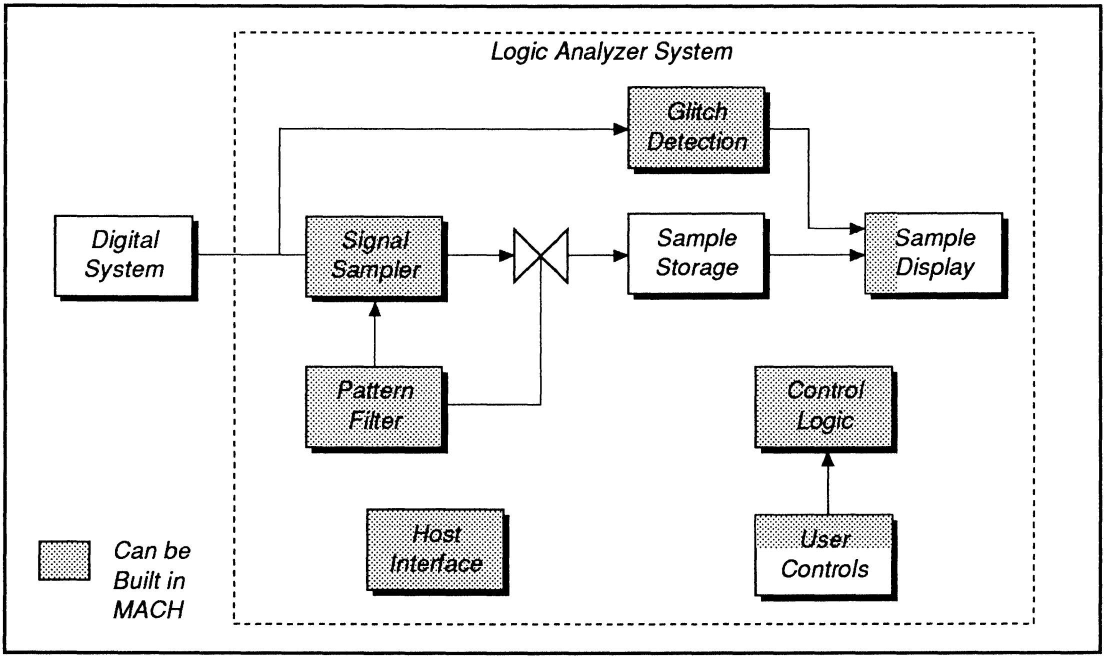
The second condition is a glitch. You provide a path for glitches either by adding a new output from the sample-storage box or by adding a new box to the system flow. The LSA design in this study includes a glitch-detection box to remind you the logic in the sample-storage box is only valid on the clock edges. The glitch-detection box must be active throughout the trace and trigger session.

Presenting data to the user is also a requirement determined by the mental simulation of data flow and operation. The most convenient way to accommodate the conversion from hardware data samples to human-readable data is to use some form of computer.

- Either a microprocessor that's built into the LSA
- Or a computer system with the LSA built in

In either case, the computer system is called the host system, since it embodies higher-level functions than the LSA. Adding this host system interface opens the possibility of LSA self-tests implemented as programs in the computer.

The mental simulation results in an improved high-level LSA architecture that includes both glitch detection and a host interface that enables testability, as shown next.



Improved LSA System Architecture Flow with Glitch Detection and Host Interface

3.1.2 MACH vs Non-MACH Devices

Any function comprised of combinatorial or sequential logic is a candidate for implementation in a MACH device. The shaded blocks in the previous figure indicate the relative amounts of function types you can implement in a MACH-device; complete shading indicates all functions of that type can be implemented in MACH. The only digital function you should not implement in a MACH device is sample storage.

Note: Regardless of the technology, programmable chips are not efficient for large memories, which is why RAM chips are used with all technologies.

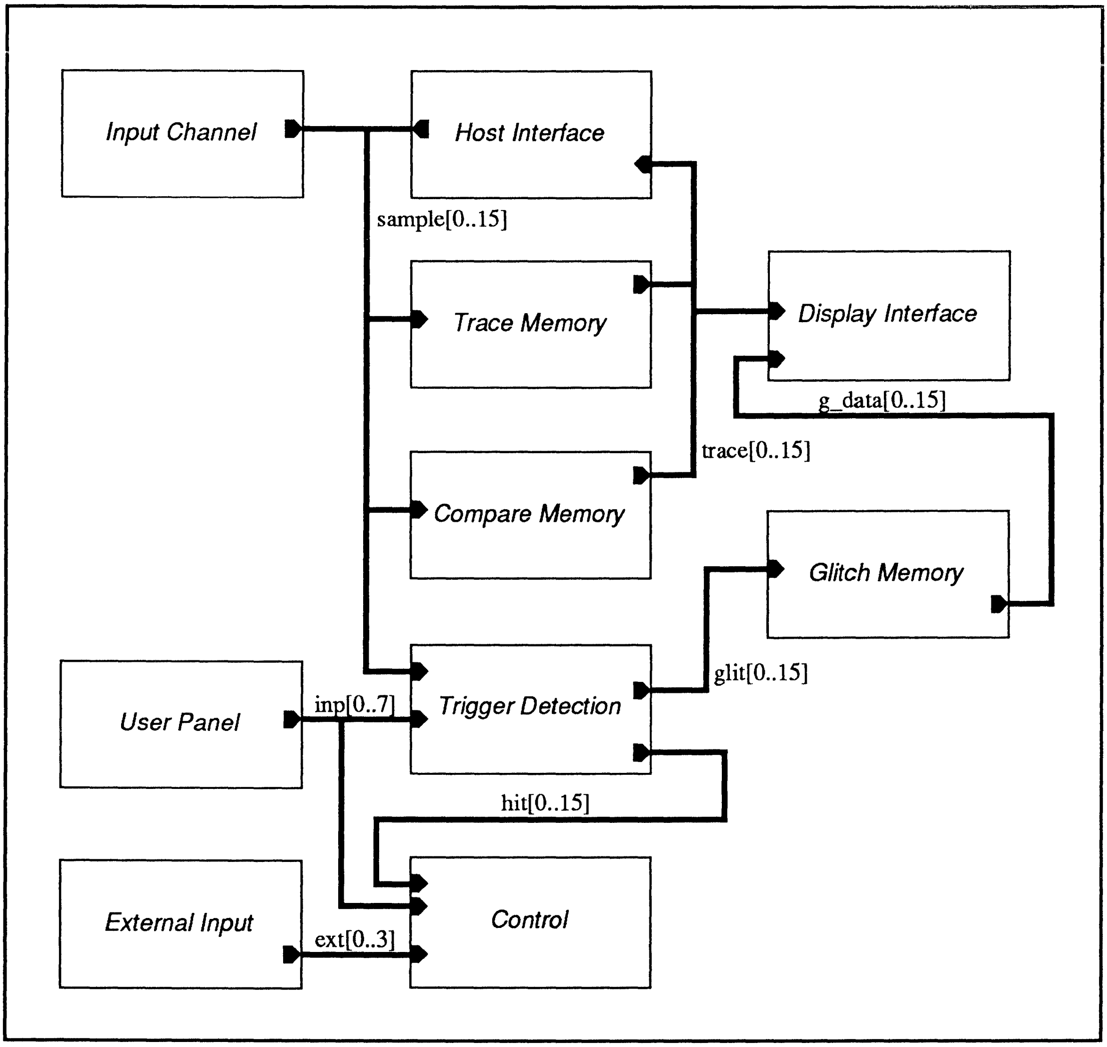
MACH devices are intended for digital subsystems; no analog functions, such as oscillators and so forth, can be realized. This design does not require any obvious analog functions; however, it does require several subtle analog functions, which are implemented using non-PLD devices.

- Single shots for strobing keyboard data are an example of a subtle analog function in this design.
- Schmitt triggers for input lines are examples of near analog functions in this design.

Speed may be considered a limiting factor when you implement timing-critical functions, such as triggering. However, at this point, it is not clear the control logic will be the limiting factor in this design. The fastest parts of the logic are the sample and storage cycles. Since fast RAM chips have access times on the order of 35 ns, the 15 ns propagation delay in the MACH device may not be the most critical path. Should control logic prove to be the limiting factor, a parallel-architecture technique can be used. The design must be refined more before the critical paths can be determined.

3.2 EXPANDING TO LSA DATA FLOW

You can convert the LSA's system-level architecture to a preliminary system data flow by increasing the detail for required functions. To do this, you create a functional block diagram to define the data-flow requirements of system functions without necessarily matching the final interconnection of logic elements, as shown next.



LSA System Data Flow

The LSA system flow not only expands the data flows from the previous block diagram, it includes functionality not readily apparent at the higher level. The most important new feature is built-in testability.

3.2.1 Testability

In keeping with good design practice, all major functions in this LSA are accessible from the host port. From a manufacturing standpoint, testability is designed in. The host port[11] allows writing data to specific parts of the system and reading the data back again. Any differences immediately indicate data-flow problems. This is especially true for the memories where a stuck bit would cause false readings for the system under test.

Note: The host interface also accesses the control logic, which means system manufacturing tests can be optimized to verify functional operation by loading specific states into the control registers. The benefit to the manufacturing process is shorter test suites for system validation and faster reduction of problem-cause sets to a single area of causality.

The consideration of manufacturing testability adds the option of increased product reliability. You could add an on-board microprocessor to interface with the host port for self-test compatibility. However, this LSA design does not include that particular microprocessor enhancement.

Observability from the host port is not universal; for example, the glitch-detection memory cannot be accessed from the host port. The LSA's functional architecture does not readily lend itself to embedding that particular data path in required data flows. How-

[11] This optional interface is not discussed here in detail.

ever, once the design is completed, observability won't be a problem. At this point in the design cycle, you merely note testability would be enhanced if the glitch-detection memory were accessible for electronic testing.

3.2.2 Trace, Trigger, and User-Interface Control

The LSA system flow shows a second trace memory that expands functional options to include the following.

- Comparison of traces taken at different times
- Comparison of live traces against stored traces, called signatures, which enter via the host port

External-timing signals consist of the clocks and trigger signals that enter the system via the external timing port.

The user panel includes a keyboard interface and an array of segmented alphanumerics to display feedback to the user. This design includes an optional keyboard scanner.[12]

3.2.3 Data Display and Data Processing

Some modes of LSA operation show samples as timing diagrams while others require the data be interpreted and shown as microprocessor instructions or hexadecimal digits. The conversion of data for presentation is not a real-time requirement. Data processing can be done very well by a microprocessor. The architecture of this design accounts for data processing through the host interface.

[12] See Appendix A for a list of all optional files provided for this LSA design.

3.3 DEFINING LSA ARCHITECTURAL PRIMITIVES

At this point, you begin to identify the functional primitives you need to create the individual bit slices of data flow and control. You'll use these repeatedly to create the overall LSA system. You can start with an analysis of the logic under test.

Designers create digital systems from combinations of logic levels and logic changes, and encode the required function in these combinations. This LSA must detect the sequences of combinations thereby decoding the logic functions. The architectural primitives needed for this LSA decode the following.

- Logic true signals with a value of 1
- Logic false signals with a value of 0
- Rising edges where 0 becomes 1
- Falling edges where 1 becomes 0
- Pulses, momentary changes in level

Once the primitives are identified you can implement each functional unit.

- In some cases, you'll create schematics using macros from the AMD-supplied MACH library.[13]

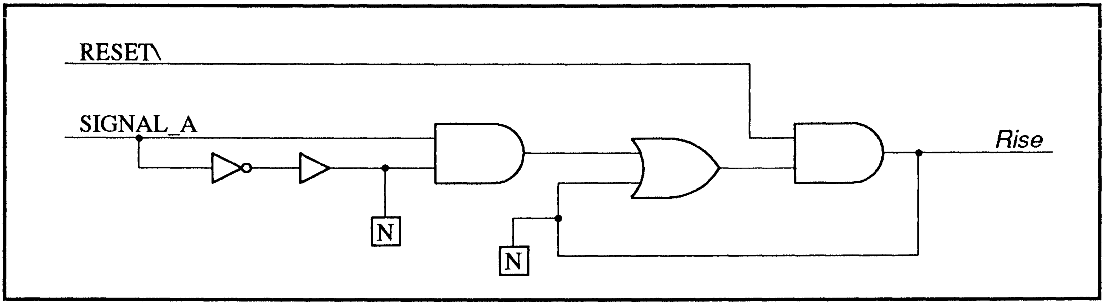
- In other cases, you'll use Boolean or state-machine syntax.

After you implement these primitives, you'll have an architecturally-unique set of integrated functions that provides the core you need to build the major blocks in the LSA's logic flow. These unique aspects of the design are as important to the efficient implementation of your architecture as the basic logic primitives, such as AND, OR, etc., in the MACH chip are to efficient logic implementation at the function level.

[13] Refer to the *PALASM 4 User's Manual*, Chapters 7 and 8, for details about the library.

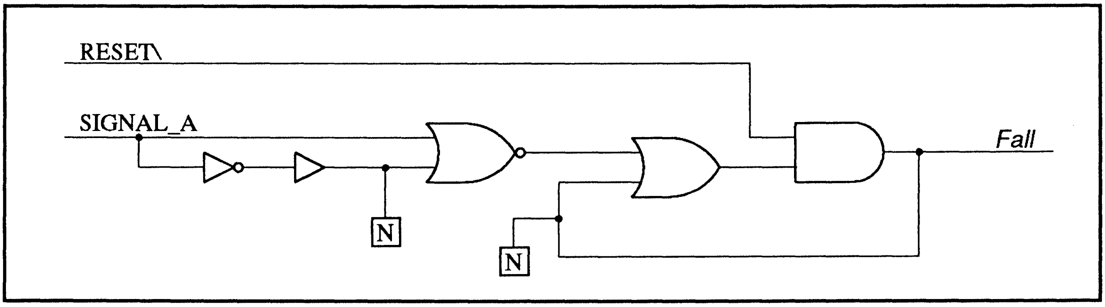
The logic-true and logic-false signals can be detected using schematic-based AND gates and inverters. The following figures show the other architectural primitives required for this LSA. To implement the control logic, you just combine these primitives.

The next figure shows one slice of a rising-edge detector.



Rising-Edge Detector

The figure below shows one slice of the falling-edge detector circuit.

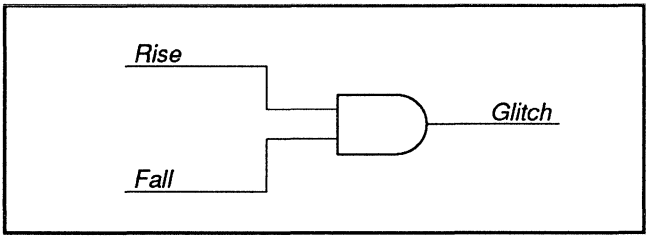


Falling-Edge Detector

The figures above illustrate a **unique aspect** of the **MACH architecture**. Signal A should arrive at the two-input gate slightly ahead of its inversion. In standard logic design, the presence of the inverters and the buffer would ensure signal A precedes its inversion. In MACH-device designs, all combinatorial logic is automatically converted to two-level logic during compilation; the standard implementation would not result in a relative delay between signal A and its inversion.

Important: To **ensure** the **appropriate delay**, you must add a **NODE macro** between signal A and the gate as shown on the falling-edge detector, block N, and discussed further under 5.1.2.

The architectural primitive for the glitch-detector circuit is shown next. A glitch is detected if a rising edge and a falling edge both occur between clock edges.



Glitch Detector

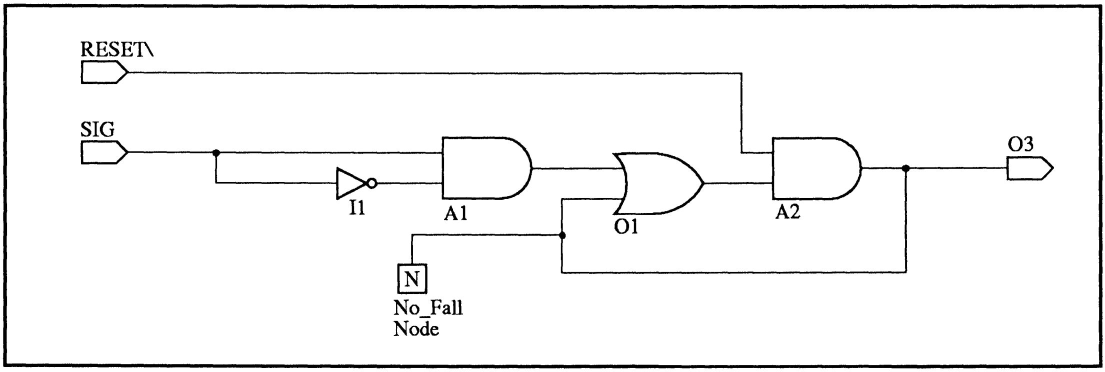
3.3.1 Performance Calculation

Each MACH report provides propagation-delay statistics for the specified chip, which identify the minimum and maximum delays for signal paths for both pure combinatorial paths and for latched paths. However, there may be times when you need to calculate timing for specific portions of the logic.

The following discussions explain how to make your own timing calculations. The key to understanding these calculations is to realize MACH-logic timing is specified in units of propagation delay through the combinatorial array; the array itself corresponds to two levels of logic.

Edge Performance

The following figure shows a portion of the edge-detection logic. Gates O1 and A2 form a latch that indicates the detection of an edge. The latch only stores data if the output of A1 remains true long enough for the true value to propagate through O1 and A2, then back to the lower input of O1. If A1 goes false afterward, the true value from A2 is maintained by the true value feeding back to O1. The time for the latch condition to become self-sustaining is called hold time in digital circuit specification sheets. In this design, it's t_LATCH.



Since I1 and A1 present the input to O1, this portion of logic must hold a true level for a period of time equal to t_LATCH. Any signal held for a lesser interval is not detected. The requirements for the minimum detectable signal can be expressed using the next equation.

t_LATCH = t_pd(O1) + t_pd(A2)

The hold-time requirement gives an additional equation, shown below.

t_h =tA1=t_pd(I1)

The minimum hold time is equal to the latch time expressed in the following equation.

t_h = t_LATCH
t_h = t_pd(O1) + t_pd(A2)

t_h consists of two levels of logic, which are equivalent to one MACH combinatorial delay, t_pd, shown next.

t_h = 1 t_pd

The calculated hold time could be a maximum of 15 ns to 20 ns, depending on the chip you select. However, in practice, the actual delay depends upon the characteristic delay of the chip you select. So you can expect to capture events much shorter than the maximum delay characteristic of MACH devices. The best you can expect is the minimum Tpd for the MACH device.

Glitch Performance

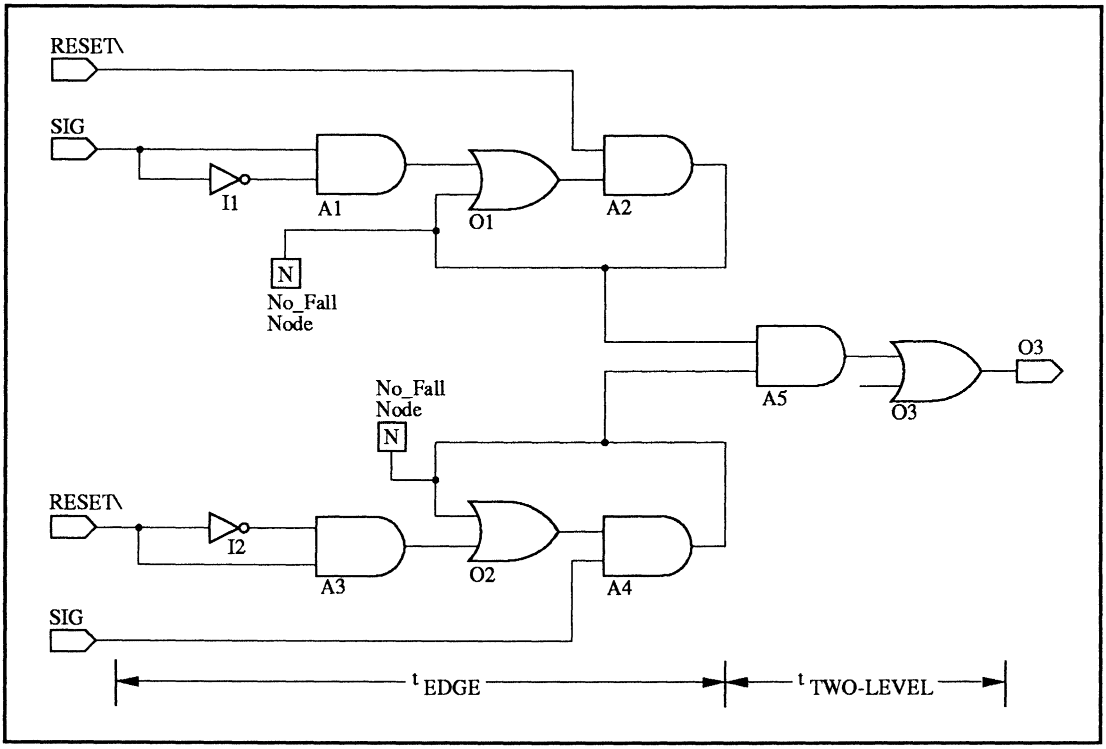
Calculating the time required to detect a glitch follows the same approach as calculating edge performance; the same principle of grouping levels of logic in pairs applies. However, the glitch-detection circuitry is a little more complex. The next figure shows glitch-detection logic.

A glitch is detected in any sample interval that includes at least one rising edge and one falling edge. Gate A5 implements the AND condition, which detects the coincidence of both edge types.

Counting from input to output, there are six levels of logic. This would lead you to suspect three propagation delays are required for glitch detection. However, a glitch event occurs when the last edge event becomes true. Edge events appear at the input to A5. The previous discussion shows t_EDGE = t_h =1 t_pd. The additional delay from the input of A5 to the output of O3 corresponds to two levels of logic, or 1 t_pd. Glitch detection time, t_ GLITCH, is the sum of these two expressed in equation form as follows.

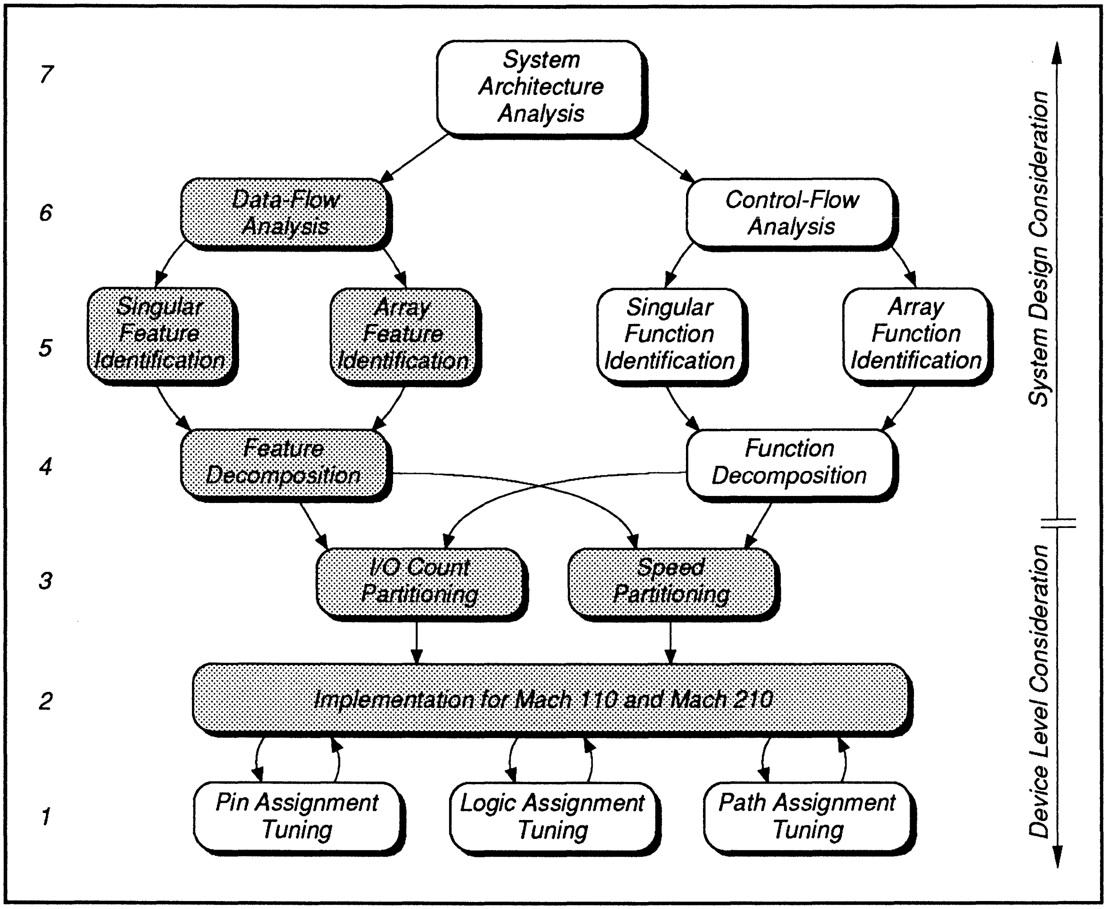
t_GLITCH = t_EDGE + t_pd(O3)
=t_two-level + t_two-level
=t_pd + t_pd
=2t_pd

As was the case for the former analysis, you can expect better performance than the maximum propagation delay for the selected MACH device. The glitch calculation just performed allows you to check performance for a specific portion of your logic. Since, in this case, it also corresponds to a complete path through the chip, its delay should be less than or equal to the delay you find in the MACH report. A review of the MACH report for the file named I_PPNB shows the maximum delay for the chip is 2 Tpd, which agrees with this calculation.



4 LSA SYSTEM DESIGN, DATA FLOW

You now divide the high-level data-flow arrays and singular features into pieces suitable for a MACH device. To ensure the pieces you select will fit in a single device, adhere to the left side of the process flow shown next. This discussion highlights the process for data flows by applying it to selected portions of the LSA.[14] You'll repeat these basic techniques to implement features in the data-flow domain.

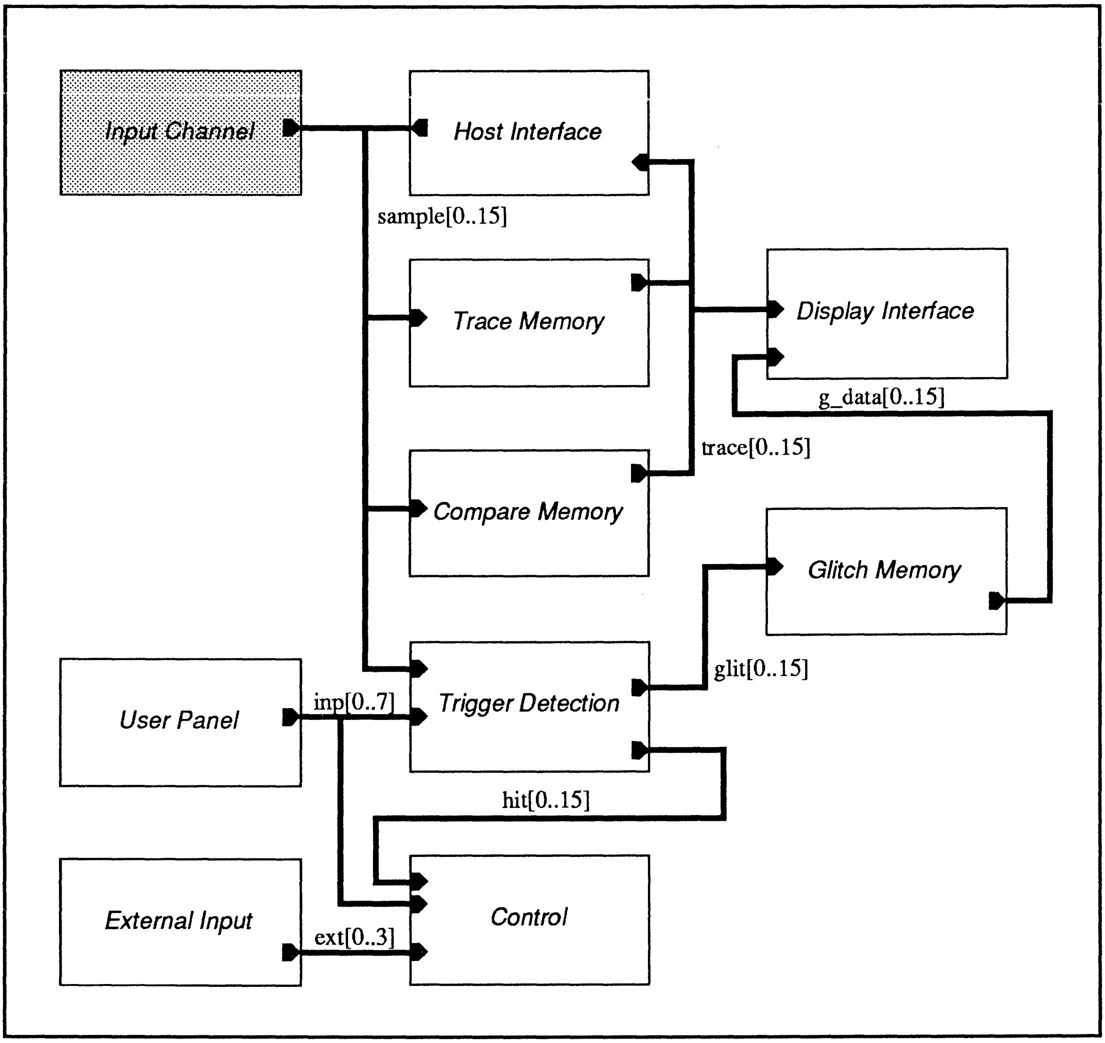

7
6
5
4
3
2
1
System Architecture Analysis
Data-Flow Analysis
Control-Flow Analysis
Singular Feature Identification
Array Feature Identification
Singular Function Identification
Array Function Identification
Feature Decomposition
Function Decomposition
I/O Count Partitioning
Speed Partitioning
Implementation for Mach 110 and Mach 210
Pin Assignment Tuning
Logic Assignment Tuning
Path Assignment Tuning
System Design Consideration
Device Level Consideration

Design Process: Data-Flow Domain

[14] Refer to Appendix A for a description of all text and schematic-based files for the complete LSA.

4.1 ARRAY FEATURES

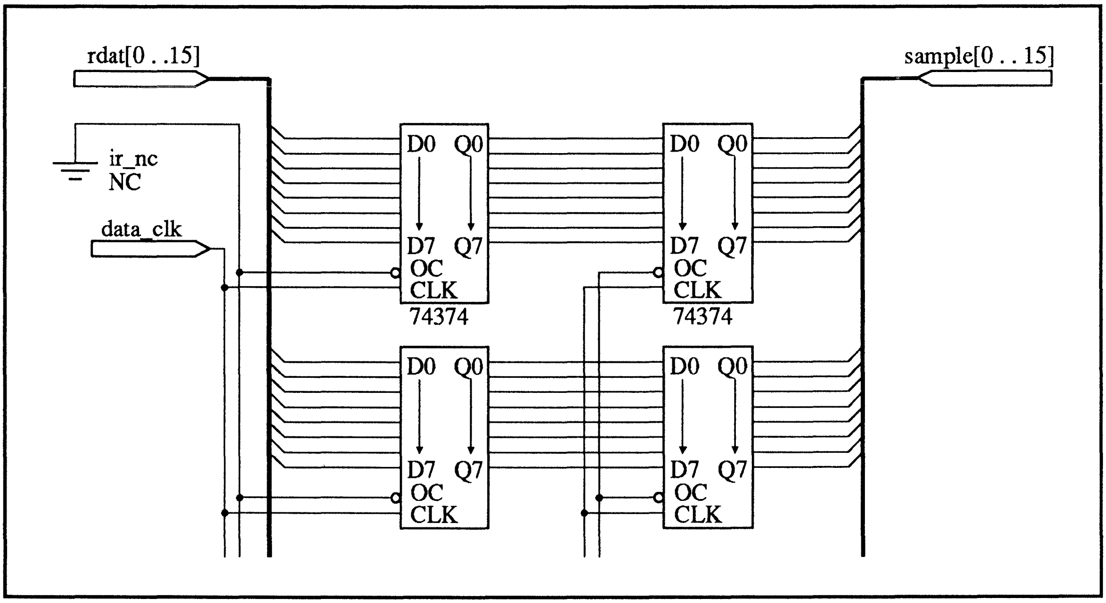
It's fairly easy to identify and isolate an example of an array structure in this LSA's data-flow domain. Just locate a part of the flow that either stores or transfers data **and** has more than one bit of information. One example is the input channel to the system, which is indicated by the shaded box in the following figure.



Input on LSA System Flow

The register in the input channel samples the input-data line whenever it's clocked. It's important to check setup and hold times. These parameters must be observed if the sample is to match the input.

However, what happens if the data changes as the clock collects a sample? You can't tell. The data may be collected correctly or it may not. Sometimes, the collected data bounces around before settling down to a final logic state. You live with this condition and give the signal time to settle down by using two registers in series, as shown next.



Schematic-Based Implementation of a Sample Register Array Feature

The input-channel implementation shown above includes two registers in series. The second register only loads stable signals to isolate the state machines from bouncing signals. You could implement these functions as a schematic-based design using the AMD-supplied TTL-type registers in the MACH 74xx library for OrCAD/SDT III.[15]

As you begin a schematic-based design, you must specify a device type in the schematic control file. Though you can easily change the specified device in the schematic control file, you must specify one initially.

At first, the MACH 210 looks like the best choice because it has more storage elements, which this register set uses in pairs. The advantage of the MACH 210 is its dedicated buried registers: the storage elements that do not use I/O pins. In this case, however, every input bit is paired with an output bit and you need the pins for the function. You don't gain anything by using buried registers in this pure data-flow application.

Upon further investigation, the MACH 110, at the fastest speed possible, becomes the best choice. Speed is a global constraint and is not determined by this single element.

The 74374 octal-register set is suitable for input and output requirements. The extra logic at the output of the first register could be cause for concern. Since both registers are on the same chip, there is no reason for the output buffers normally found on a TTL 74374.

Note: During compilation you can have small amounts of unused logic removed from a design automatically by tying the unused I/Os to a MACH NC macro.[16] This can ease concerns about an additional propagation delay and use of additional chip resources.

[15] Refer to the *PALASM 4 User's Manual* for details: Chapter 3 is a schematic design-entry tutorial, Chapters 7 and 8 describe the library, and Chapter 9 describes all commands, options, and forms.

[16] Refer to the *PALASM 4 User's Manual*, Chapter 7, for details about the NC macro.

An alternative to using the TTL 74374 would be to use the TTL 74273 macro, which does not have output buffers. This approach does not require the NC macro to disable the output-enable, because the 74273 does not have an output-enable terminal. The TTL 74273 has an active-low CLR terminal, which can be tied to ground if you select that option. At this stage, it is sufficient to know that the function can fit on the chip without a major shortage of pins.

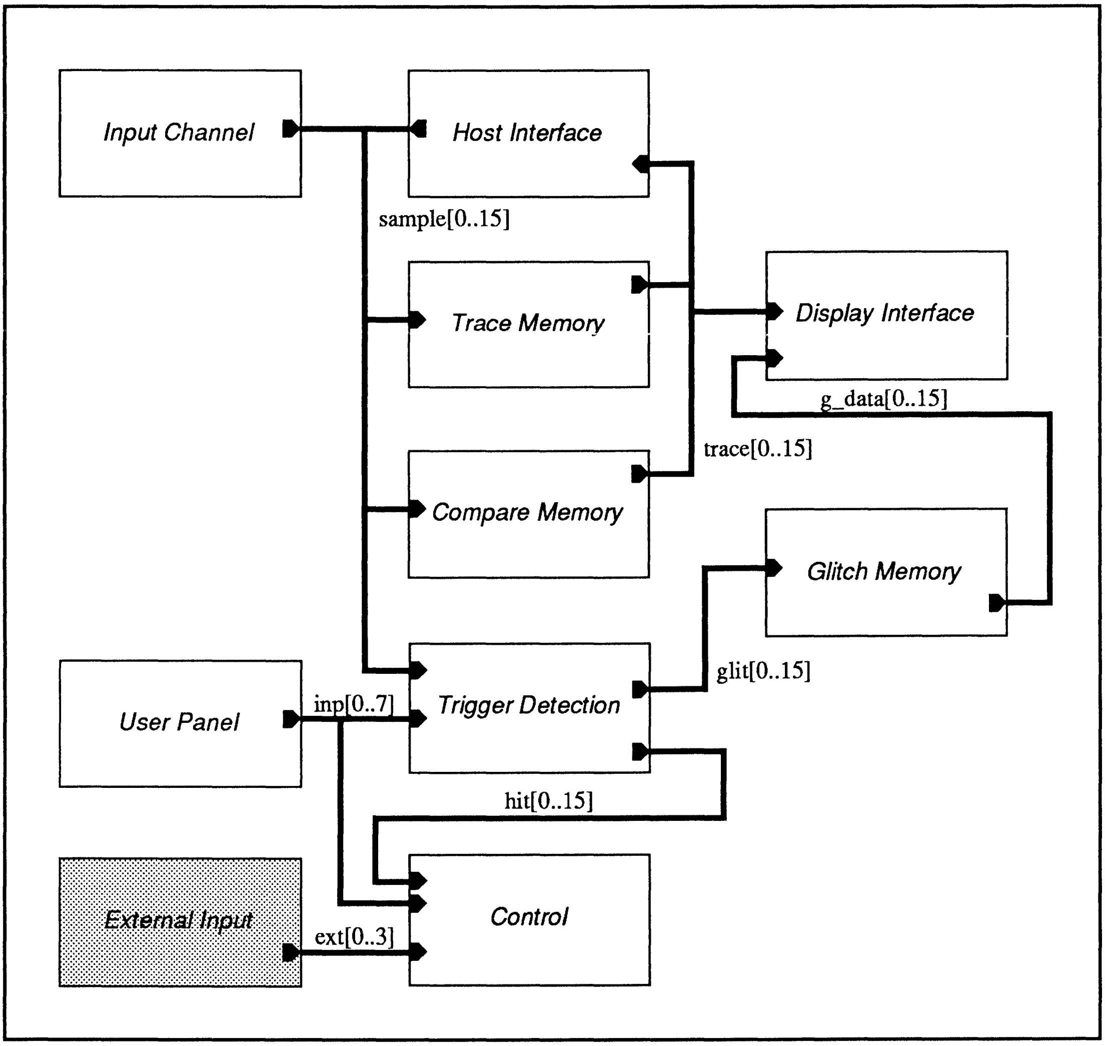
You can compile this function immediately and confirm the fit on a single MACH chip. At this stage, you use default logic-synthesis and MACH-fitting options to get a first-order estimate of required resources. The MACH report indicates the amount of device resources required for the bit slice so you can calculate how many bit slices will fit on a single chip.

Toward the end of the implementation phase, you'll place several bit slices together in one schematic design and compile them to see if the logic fits in a single MACH chip. At that time, specific fitting options can enhance the results. Using 32 of the 38 pin outs, you should be able to fit 16 bits of double registers in a single MACH 110 device. This leaves six I/Os for any isolated state-machine or data-flow bit slices.

4.2 SINGULAR FEATURES

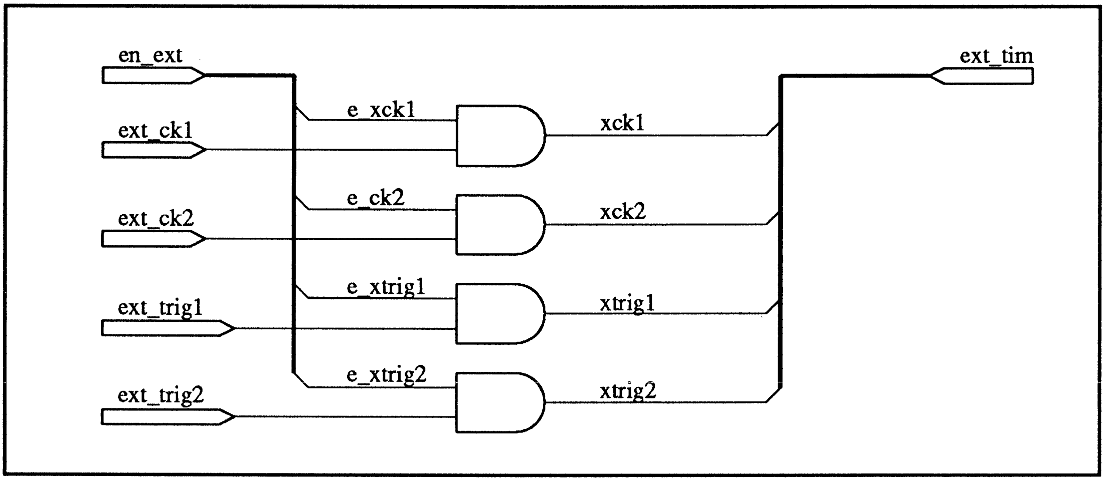
Next you look for singular data-flow features. These isolated pieces of data-flow logic can fit on array-based chips that do not require all device resources for the array. The external-timing signal logic, indicated by the shaded block in the next figure, is just such a case.

These signals enter the system and are not transformed as they traverse the data flow. This condition matches the definition of data flow exactly, although the signals are used in the control logic.



External Timing Block on LSA System Flow

As with most data-flow implementations, you begin this bit-slice design using a schematic-based input format because it's easier to place and name gates than to track equations for data-flow design elements. Another advantage of schematic entry for data-flow input is the ability to work on several pieces simultaneously, without losing track of where in the design you are or inadvertently leaving out signals. The implementation of a singular data-flow slice, the external-timing block, appears next.



Schematic-Based External-Timing Singular Data Flow

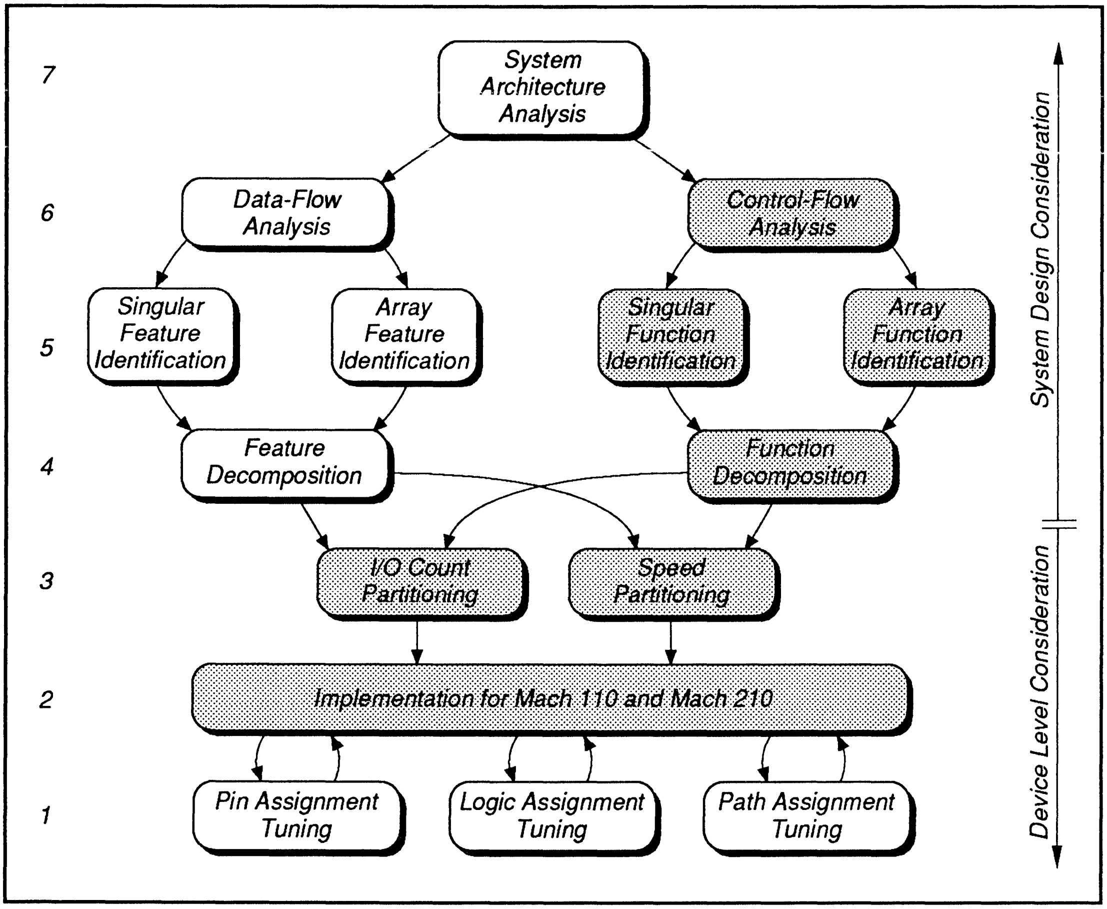
Again, you must decide which device to use. The MACH 110 is chosen because this small piece of singular logic does not require its own chip. After entry, you compile this singular feature using standard logic-synthesis and MACH-fitting options, then review the MACH report to determine the percentage of chip resources needed by this feature You then set this design aside until you find a chip with the appropriate percentage of space left. If you are using a MACH 210 device, you find a chip with about half the percentage of resources left.

The rest of the data-flow domain is implemented in the same manner.[17]

[17] Appendix A lists the schematic file names so you can print them for review or use them.

5 LSA SYSTEM DESIGN, CONTROL LOGIC

To develop the LSA's control logic, you separate the singular functions and arrays for that part of the design. This time, you use the right side of the design-process flow as a guide.


7
6
5
4
3
2
1
System Architecture Analysis
Data-Flow Analysis
Control-Flow Analysis
Singular Feature Identification
Array Feature Identification
Singular Function Identification
Array Function Identification
Feature Decomposition
Function Decomposition
I/O Count Partitioning
Speed Partitioning
Implementation for Mach 110 and Mach 210
Pin Assignment Tuning
Logic Assignment Tuning
Path Assignment Tuning
System Design Consideration
Device Level Consideration

Design Process, Control-Flow Domain

During the identification and decomposition processes, you develop a tree for each function; the leaves identify specific functions and subfunctions. The goal is to ensure each identified function can fit on a single MACH chip so you can make partitioning decisions quickly.

If the control-flow domain were implemented in a single PDS file, all states for the supervisory and subsidiary machines would be implemented as a single set of states. This may be a good strategy to minimize logic; however, an important design consideration is how to keep state definitions separate so you can observe the behavior of the logic during manufacturing or field-debugging.

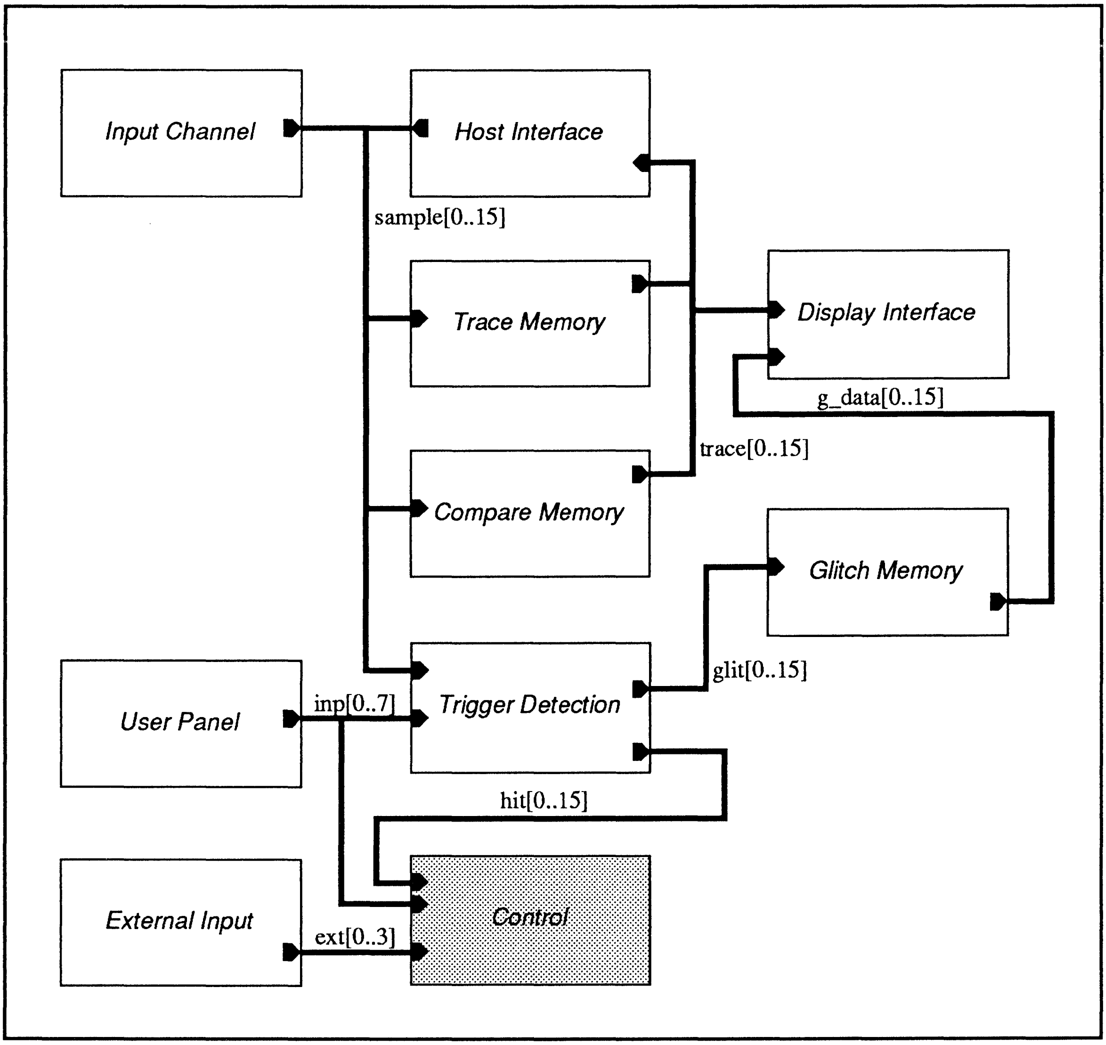
To keep state definitions separate, you must implement the supervisory and subsidiary state machines as individual PDS files. Then you use the Merge design files command on the PALASM File menu to combine several state-machine designs together for implementation on a single MACH chip without blending the states.

Details in following discussions cover only the LSA supervisory state machine and one subsidiary state machine;[18] you repeat the basic techniques to implement all blocks in the control-flow domain. Each block discussed next is initially implemented as a separate PDS file using PALASM state-machine syntax.

[18] Refer to Appendix A for details about all files required for the LSA implementation.

5.1 SINGULAR CONTROL STATE MACHINES

To identify the state machines required for this LSA, you consider the sequences of data flow that occur when the machine is operating. For example, the two primary activity sequences, triggering and tracing, help you identify the fundamental state machine. These functions are located in the shaded control block in the following LSA system-flow diagram.



Control Block, LSA System Flow

5.1.1 Trigger Detection and Trace Control

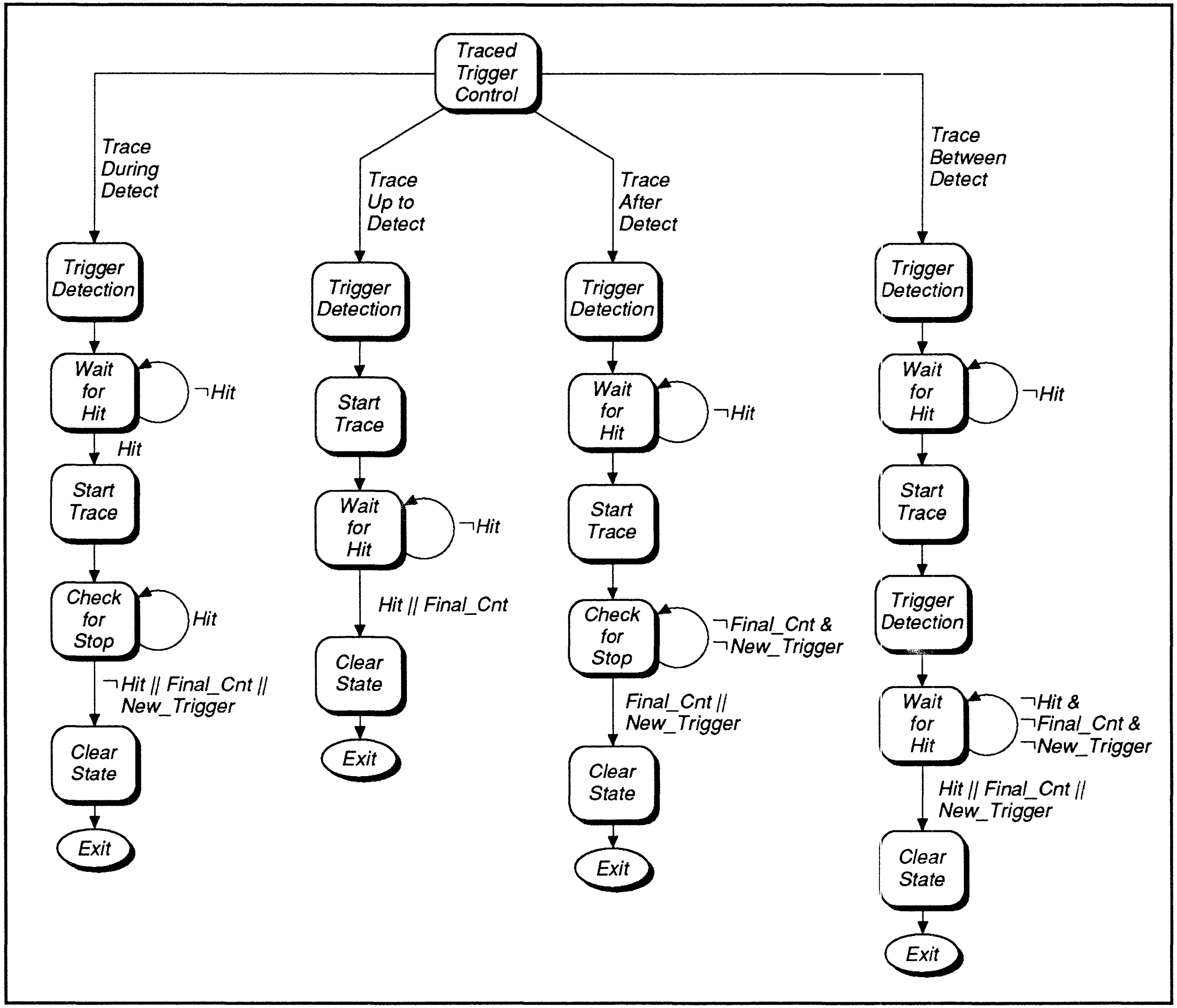
A generalized diagram of the types of actions the LSA control logic should manage is shown next as a state-machine flow diagram.

The top-level block, Traced Trigger Control, represents the **supervisory** state machine. Secondary blocks represent the beginning of the four operational modes shown as vertical columns.

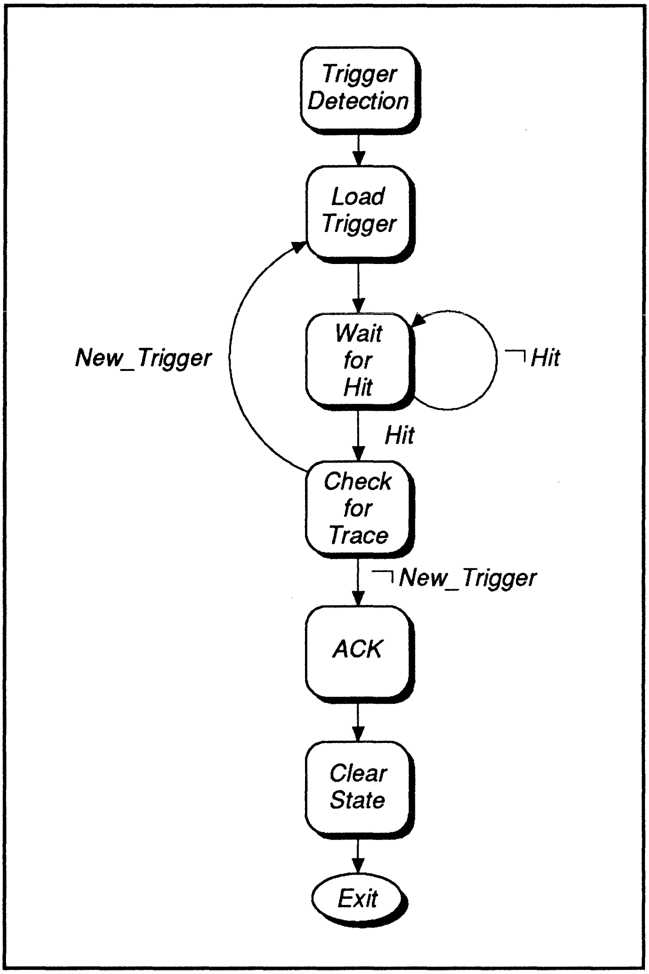
- Trace During Detect
- Trace Up to Detect
- Trace After Detect
- Trace Between Detect

For this design, every node of the supervisory state machine is a separate subsidiary state machine. The supervisory state machine is started by the user; subsidiary machines are started by the supervisory machine when they are needed.

Traced Trigger Control Flow



Each operational mode includes several subsidiary states. To confirm this, you can look at the first node in the third column of the previous figure, then look at the expanded view of one of the nodes in the following figure.



Trigger Detection Sequence

First, a trigger pattern is loaded, then a wait state is invoked during which one of two things happens.

- The wait continues if the patterns do not agree.
- A hit occurs and the next state is invoked if the patterns agree.

A new trigger is loaded after each hit and the load and wait sequence repeats. When all triggers have been matched and there are no new triggers, the acknowledgment signal is sent and the sample patterns are stored in the trace memory for display.

This machine does not control the trace; the supervisory machine does. The supervisory machine starts the trace state machine when acknowledgement is received from this machine. Trigger detection ends with the subsidiary machine's local Clear State.

During the trace-control process, successive input samples are loaded into the trace memory. This occurs until either the memory is full or some other terminal condition occurs, such as the final count for timing offset from the trigger, which also requires a state machine.

If you return to the entire traced-trigger control flow for the supervisory state machine, you'll see the subsidiary states represented by individual nodes on the figure are similar for each operation; many are repeated. For example, Trigger Detection initiates each operation, which explains why the Trigger Detection node appears at the beginning of each vertical column on the figure.

Each node consists of several states. This means you can decompose the supervisory state machine into submachines that influence special subfunctions only. This approach simplifies the LSA's control-logic synthesis and involves fewer states per machine. It also simplifies the manufacturing and debug process because specific failures can be associated with specific parts of the design.

5.1.2 State-Machine Assignment

Translating the supervisory flow diagram into actual state machines involves assigning binary codes to each node on the diagram and converting state assignments to PALASM language syntax. Particular state assignments are not critical when you use clocked logic, provided you use a clock pulse long enough to allow the next-state's decode logic to settle down.

If you decode the states to create output controls, the decodes are subject to the transient values assumed by the state variables.

Tip: When you do not assign adjacent states, ensure that the output variables, decoded from state variables, are also clocked. You will then avoid unwanted output results during state changes.

In some cases, fewer flip-flops are required if you do not make all next states adjacent. However, in other cases, designing sequential logic from gates rather than flip-flops can result in logic with a faster response time.

This LSA design uses the clocked flip-flop approach. The following discussions focus on only two state machines for the LSA because they embody the major control-flow functions.

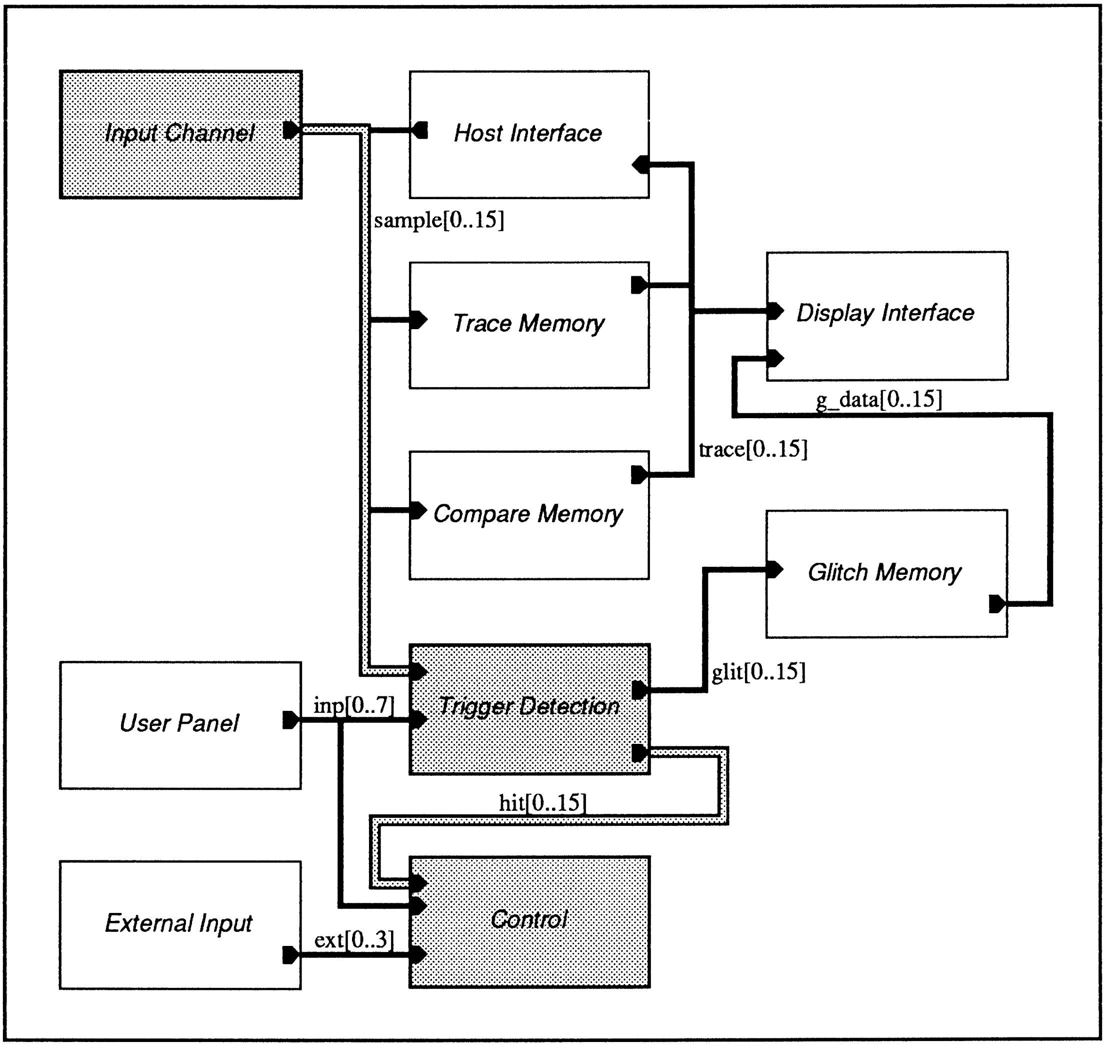
- Traced trigger control is the supervisory machine.

- Trigger Detection is the first subsidiary machine.

All other nodes can be implemented using the techniques discussed next.

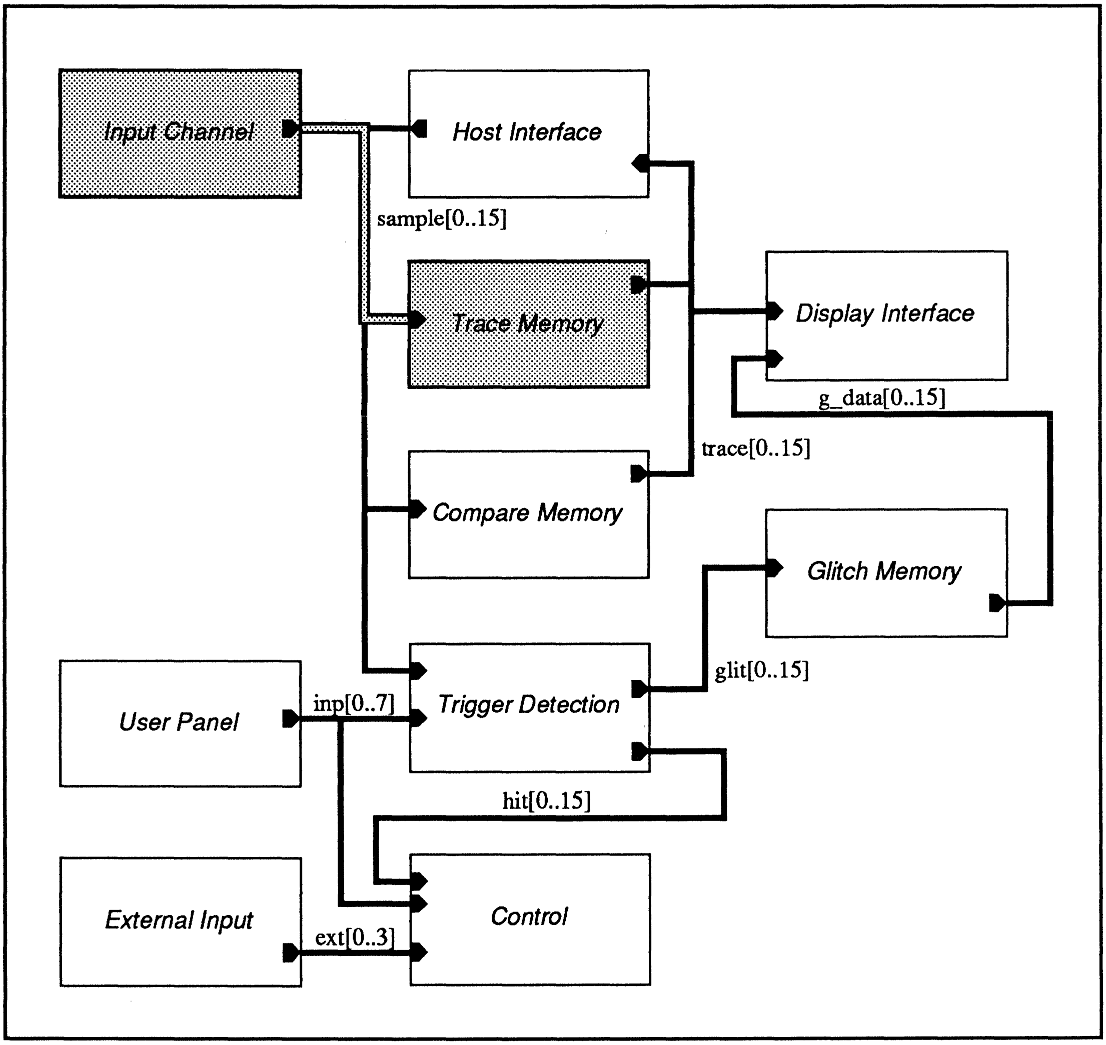
Partitioning and Implementation

The next two figures review the data flows the supervisory and first subsidiary state machines will control. Decodes on the states generate signals to control the paths. The path for the input trigger-detector control is highlighted in the following figure.



Trigger Data Path

The following figure highlights the path for the trace data.



Trace Data Path

MACH devices have many flip-flops; you don't have to use valuable combinatorial logic to implement them. In fact, if you use transition equations you don't even have to assign states; you just name each state and specify how to change from one state to the other. During compilation, the state values are assigned automatically and recorded in a table in the execution-log file so you can review them.

> **Note**: For this design, specific states were assigned to retain separate state machines inside the MACH 210 chip after merging multiple PDS files together.

As you begin partitioning, you note the maximum length of any path in the supervisory flow is six nodes. You could use three flip-flops for this number of states; however, four flip-flops allow for additional states you may need in other operations.

Relating Supervisory and Subsidiary State Machines

Since each node is a subsidiary state machine, you can decode the value of the supervisory state machine to activate the subsidiary machine. This strategy ties the supervisory machine to subsidiary machines. Two flip-flops are assigned to the subsidiary trigger-loading machine, which has three states.

A text-based state-machine design that describes the supervisory state machine is discussed, and shown in part, next. You can print the following file for review or use the file at the workstation.

```
PALASM\EXAMPLES\CB\SAMPLES\LA_KMAIN.PDS
```

The first thing you do in any sequential design is ensure it starts in the correct state. For this design, starting with all flip-flops cleared to zero is enough, which is facilitated using the statement below.

```
NODE 1  POR_INIT
```

Node 1 in a MACH device is a special buried node you use to initialize the storage elements. If you allocate Node 1 in the pin declarations, you can reset it in the equations segment to clear the entire chip.

```
:
CHIP     _LA_KMAIN        MACH 110
;------------------------------PIN Declarations------------------------------
PIN ? /POR                COMBINATORIAL          ; Power On Reset
NODE 1 POR_INIT
PIN 35 CLK1                                      ; Default Clock on pin 3
PIN ? MSW[0]              REGISTERED             ;
;
NODE ? K[0..3]            REGISTERED             ;
NODE ? K_C0[0..1]         REGISTERED             ;
NODE ? K_C1               REGISTERED             ;
:
;STRING DECLARATIONS.
STRING GL '(MWS[0])'
:
STRING S_K0  '/K[3]*/K[2]*/K[1]*/K[0]'           ;Main Control State Bits
:
STRING S_TDD '/TR1 * /TR0'
:
;------------------------------------Equations-------------------------
EQUATIONS
POR_INIT.RSTF = POR
STATE
M_K0 = /K[3]*/K[2]*/K[1]*/K[0]                   ;Main Control State Definition
:
MOORE_MACHINE                                    ;Main Trace Control State Machine
M_K0 := TDD -> M_K1
               + TTD -> M_K1
               + TAD -> M_K1
               + TBD -> M_K1
               + -> M_K0;
:
;------------------------------------Conditions------------------------
CONDITIONS
TDD = /TR1*/TR0*RUN*/POR                         ;Operational Mode Bits
TTD = /TR1* TR0*RUN*/POR
:
```

Supervisory State Machine, Partial Description

A partial listing of the first subsidiary machine is shown next. Again, you can print the following file for review or use it at the workstation.

```
PALASM\EXAMPLES\CB\SAMPLES\LA_C0.PDS
```

The line below appears in the state segments of both files to clear the entire chip.

```
POR_INIT.RSTF = POR
```

This LSA design uses POR, or Power On Reset, to reset the system when power is first applied. It could just as easily be a system reset or any legal name you choose.

States and Changes, Strings and State Definitions

The states in this design are defined twice: once in the string declarations and again in the state segment. You use the string definitions in logic or condition equations. State declarations are used by the transition equations in the state segment of the PDS file. Strings can only appear on the right side of an equation; state definitions can appear only on the left side of an equation.

You define actual changes from one state to another using transition equations in the state segment of the PDS file. This is where you list the states and conditions that cause changes to subsequent states.

- All state definitions in this LSA design are preceded by M_, which allows you to identify the use of the variable in the design.

  ```
  M_K0 = /K[3]*/K[2]*/K[1]*/K[0]   ;Main Control State
  ```

- The same names are used in the string definitions in this design, however, the prefix in this case is S_.

  ```
  STRING S_K0  '/K[3]*/K[2]*/K[1]*/K[0]' ;MCS Bits
  ```

```
:
CHIP    _LA_C0 MACH 110
;-------------------------------PIN Declarations------------------------------
PIN ? /POR         COMBINATORIAL                     ; Power On Reset
NODE 1 POR_INIT
PIN 35 CLK1                                          ; Default Clock on pin 35
PIN ? K_CLK        COMBINATORIAL                     ;
PIN ? K0           REGISTERED                        ;
PIN ? K1           REGISTERED                        ;
PIN ? K2           REGISTERED                        ;
PIN ? K3           REGISTERED                        ;
PIN ? MSW[1]       REGISTERED                        ;
PIN ? MSW[2]       REGISTERED                        ;
PIN ? MSW[3]       REGISTERED                        ;
PIN ? MSW[4]       REGISTERED                        ;
:
;STRING DECLARATIONS
:
STRING S_K0  '/POR*RUN*/K3*/K2*/K1*/K0'              ;Main Control State Bits
STRING S_K1  '/POR*RUN*/K3*/K2*/K1* K0'
:
;--------------------------------------Equations--------------------------
EQUATIONS
-------------------Initialization
POR_INIT.RSTF = POR
STATE
MEALY_MACHINE                                        ;Main  Trace Control State Machine
; Machine C0
M_C0_0  = /K_C0_1*/K_C0_0                            ;C0 Control State Definition
:
M_C0_0 := TR_RD -> M_C0_1
          +-> M_C0_0;
:
M_C0_0.OUTF = /AM_G_CS*/AM_G_OE*/AM_G_WE*/AM_G_ADDR_CK
                     */PM_G_CS*/PM_G_OE*/PM_G_WE*/PM_G_ADDR_CK
;--------------------------------------Conditions-------------------------
CONDITIONS
NULL_TR = /POR*/HIT*S_LSA
:
```

Subsidiary State Machine, Partial Description

By using string and state definitions, you can keep the state-machine definition and state-value definition separate. If you have to add more states, you just change the string and state definitions; in this case, you simply change variables beginning with S_ and M_; the types of equations listed below remain the same.

- Transition equations in the state segment
- Conditional equations in the state segment following the condition keyword
- Boolean equations in the equations segment, which are based on the states

This strategy also applies if you can reduce variables for greater density on the chip. If you implement the control logic as multiple machines, only a few combinations must be changed for any particular machine.

Buried Registers

Allocation of storage for the state variables is a design consideration you should not overlook. Each state variable is declared as a node statement; this is how you specify a buried register in a MACH device. Buried registers do not have a direct connection to I/O pins. Instead, they must be routed to I/O pins via other macrocells with I/O connections.

Usually, the states in state machines must be decoded to provide a control signal that typically leaves the chip. Choosing a buried register for the state bits leaves a layer of logic available to create the control signals between the state machine and the I/O pin. Otherwise, you'd have to use another pin to allow the control signal to leave the chip.

The next figure shows details of state definitions for the supervisory and subsidiary state machines, as defined in the PDS file named LA_KMAIN.PDS.

- The supervisory machine is identified using the letter **K** as the first letter of the signal name.

- The subsidiary machine is identified using the letter K followed by an underscore, **K_**.

C# identifies which subsidiary machine corresponds to the equations.

```
;LA_KMAIN
M_K0  = /K3*/K2*/K1*/K0
M_K1  = */K3*/K2*/K1* K0
M_K2  = /K3*/K2* K1*/K0
M_K3  = /K3*/K2* K1* K0
M_K4  = /K3* K2*/K1*/K0
M_K5  = /K3* K2*/K1* K0
M_K6  = /K3* K2* K1*/K0
M_K7  = /K3* K2* K1* K0
M_K8  =  K3*/K2*/K1*/K0
```

State Definitions for Supervisory State Machine

```
;LA_C0
M_C0_0 = /K_C0_1*/K_C0_0
M_C0_1 = /K_C0_1* K_C0_0
M_C0_2 =  K_C0_1*/K_C0_0
M_C0_3 =  K_C0_1* K_C0_0
```

State Definitions for Subsidiary State Machine

Testing and Observability

During LSA operations, it is important to know what the state machine is doing so you can detect malfunctions. All MACH devices can be placed in a test mode where the internal states can be gated to the I/O pins for observation. This option is only available to PLD programmers.

To design the observability of internal states, you can define a machine-state word connected to the I/O pins. Each bit of the machine-state word for this LSA design, MSW[0] through MSW[15], can be observed. These high-level state indicators are allocated to I/O pins in the declaration segments of the PDS files. One such statement, from LA_C0.PDS, is shown below.

```
PIN ? MSW[10] REGISTERED
```

Floating and Fixed Pin Locations

The question mark, ?, in the location-number field of certain pin and node statements specifies a floating pin location.

```
PIN ? /POR COMBINATORIAL
```

In this case, the signals are automatically assigned to specific pins on the MACH device during compilation. This strategy usually leads to a better use of chip resources and the increased probability of a fit.

There are times, however, when you may want to assign the signal to a specific pin number, as indicated in the following clock-signal declaration.

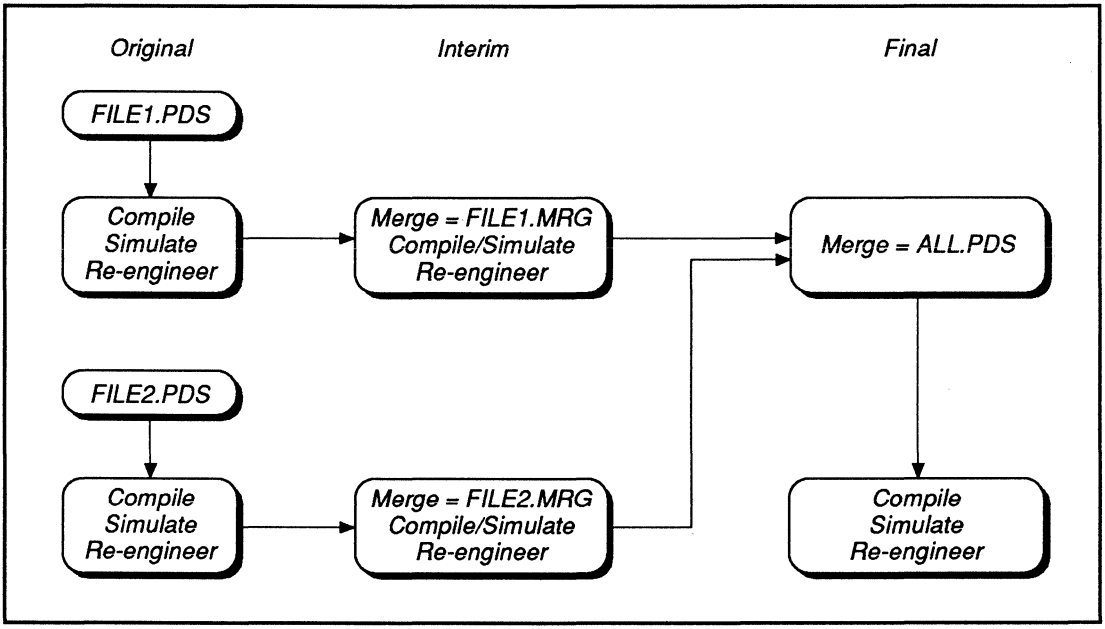
```
PIN 35 CLK1 COMBINATORIAL
```

Merging Design Files

After entering the supervisory state machine and one or more subsidiary machines, you compile each to confirm there are no syntax errors and to determine the percentage of a single chip's resources required for each. After you simulate each bit slice to determine it operates as desired, you can merge two or more bit slices into one design file for a single MACH chip.

Each design is automatically checked for syntax errors when you initially get the file to merge. During this check, state-machine syntax is converted to Boolean equations.

It's a **good idea** to use an iterative approach that includes creating interim files when you merge designs. The approach illustrated next ensures the integrity of converted state-machine designs before you merge them into a single design file. In the long run, this approach can save time you might otherwise spend debugging the combined design.



Iterative Merge Process

The steps below produce the interim and final files.

1. Enter the original PDS file, compile, simulate, and re-engineer as needed.

2. Initiate the merge process, name an interim file where you'll store the single design slice, get the design file, merge it into the interim file, and quit the merge process.

Recommendation: Compile and simulate[19] each interim file to ensure it operates as it did before its conversion to Boolean equations; re-engineer each interim file if needed.

3. Initiate the merge process, again, and create the final file for all state machines in a single MACH chip, then get one interim file and merge it into the final file.[20]

Important: Each time you merge an interim file into the final file, quit the merge process and recompile and resimulate the final design. This ensures the addition did not adversely impact the final combined design.

4. Repeat steps 1 through 3 for each bit slice until the final file contains all slices for a single chip.

File Differences

A partial listing of the final combined file for the supervisory state and one subsidiary state appears next. There is one major difference between this file and the two original files: the state-machine definitions now appear as Boolean equations for each state variable, K0 through K3. The combined file contains Boolean descriptions beginning with K0 :=. When specified this way, subsequent compilation does not combine equations.

Two things that may not be apparent are the unchanged string definitions for states and conditions. These definitions were removed during the merge

[19] The simulation segment of each PDS file is removed during the merge process. Refer to the Simulation for Interim and Combined Design discussion.

[20] Refer to the Logic Assignment on a Single MACH Chip discussion.

process; copies from the original file were manually placed in the combined file using a text editor. This ensures that each stage of the multiple-machine compilation sequence always refers to design variables by the same name.

```
:
CHIP                    _LA_MERGE        MACH 110

;--------------------------------PIN Declarations-------------------------------
PIN ? /POR COMBINATORIAL          ; System On Reset
PIN ? /POR1 COMBINATORIAL         ; Power On Reset
NODE 1 POR_INIT
PIN 35 CLK1 COMBINATORIAL         ; Default Clock on pin 3
PIN ?K_CLK COMBINATORIAL          ;

PIN ? MSW[0] REGISTERED           ;
PIN ? MSW[1] REGISTERED           ;
:
NODE ? K0 REGISTERED              ;
NODE ? K1 REGISTERED              ;
:
;STRING DECLARATIONS.
STRING GL ' (MWS[0])'
STRING DL ' (MWS[1])'
:
;--------------------------------------------Equations---------------------------
EQUATIONS
;-----------------INITIALIZATION
POR_INIT.RSTF=POR
;-----------------OPERATION
K0 := /K3 * /K0 * ACK
         + K0 * /ACK
         + /K3 * /K2 * /K1 * /K0
:
```

Combined Design File

Logic Assignment on a Single MACH Chip

During the merge process, the goal for logic assignment is to determine how to combine individual designs to maximize the functions on a single chip and to reduce the overall number of chips. This is most effective in data-flow design where you isolate bit slices with the same topology and put them into a single chip that will be used repeatedly.

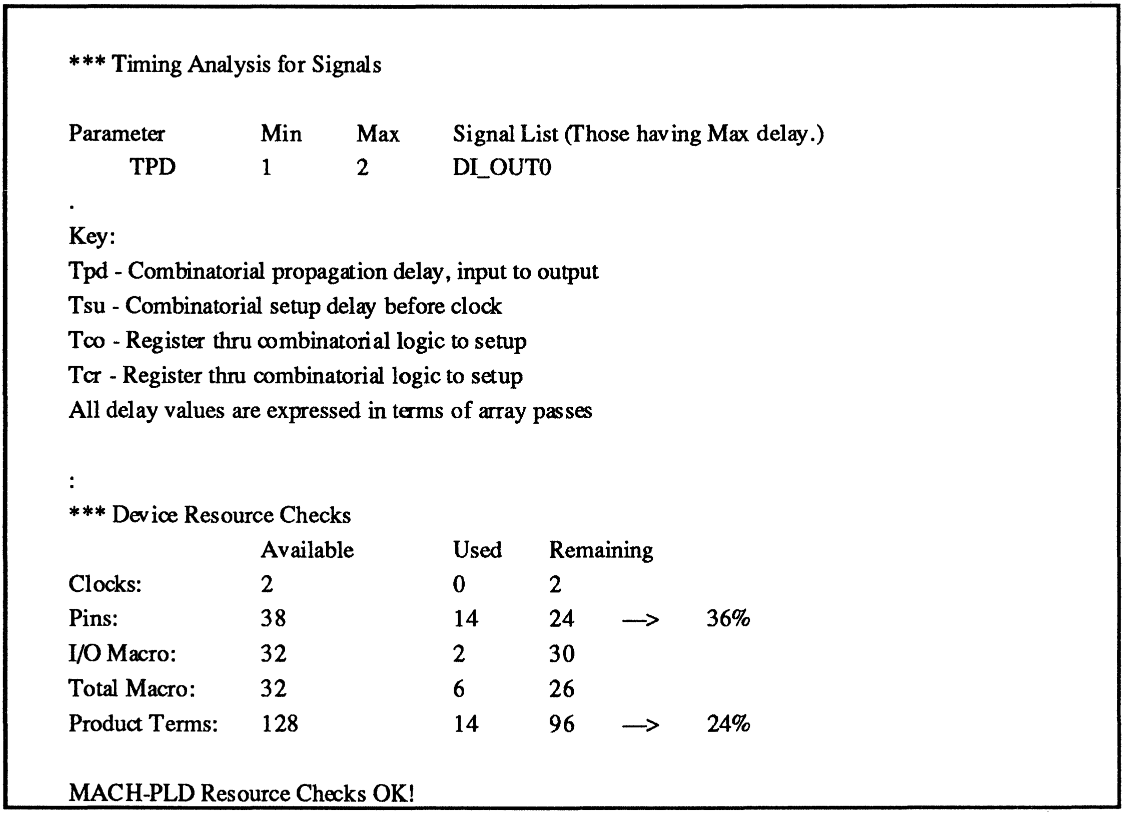
In the control-logic domain, however, you have a much lower probability of finding logic to be used repeatedly. Consequently, the chief parameter of logic assignment is the I/O count.

You look for functions on the basis of their I/O requirements and fit as many as possible in the space remaining on a particular device. Variations on this theme occur if the bit slices to be merged must communicate. In this case, putting the logic on the same chip eliminates the I/O pins required to effect the communication. In general, it's enough to consider the inputs and outputs for the control logic that direct a data flow, which is the case for this design.

The key items that determine state change are the outputs required of the state machine and the inputs received from other state machines. In most cases, there are many more of these I/O variables than the number of state variables in the control-logic implementation. This fact alone indicates the advantage of the buried registers in MACH devices.

The state machine can be implemented from buried registers without using an I/O pin. Both the MACH 110 and the MACH 210 support designs with buried registers. The MACH 110 allows normal I/O registers to be buried, which frees an I/O pin to be used for a signal input instead of a state variable. The MACH 210 has dedicated buried registers in addition to the normal I/O registers.

The supervisory state machine and at least one subsidiary state machine can fit on a single MACH chip because of the I/O count and because you are using just a few registers. You review the MACH report after merging the two designs, LA_KMAIN and LA_C0, and compiling the final design. The segments of the MACH report, shown next, indicate the chip resources used by the two state machines: only 36% of the pins and 24% of the product terms are used.



```
*** Timing Analysis for Signals

Parameter        Min      Max      Signal List (Those having Max delay.)
     TPD          1        2       DI_OUT0

.
Key:
Tpd - Combinatorial propagation delay, input to output
Tsu - Combinatorial setup delay before clock
Tco - Register thru combinatorial logic to setup
Tcr - Register thru combinatorial logic to setup
All delay values are expressed in terms of array passes


:
*** Device Resource Checks
                 Available         Used     Remaining
Clocks:          2                 0        2
Pins:            38                14       24       —>       36%
I/O Macro:       32                2        30
Total Macro:     32                6        26
Product Terms:   128               14       96       —>       24%

MACH-PLD Resource Checks OK!
```

Partial MACH Report for Combined Design

The utilization statistics indicate you can merge another bit slice into the final design to add more logic to this chip. You enter, verify, and add the rest of the control logic to the base chip in the same fashion. Each MACH report, produced as you compile each individual bit slice, tells you whether the new logic is likely to fit in the space remaining on the chip. The sequence of iterative steps is listed below.

- Enter, compile, and simulate each design slice.
- Initiate the merge process and create an interim file for each design slice, then compile and simulate each interim file individually.
- Merge individual interim files into the final design and validate the final design after each addition.
- Simulate the final combined design when all pieces have been merged.

Simulation for Interim and Combined Designs

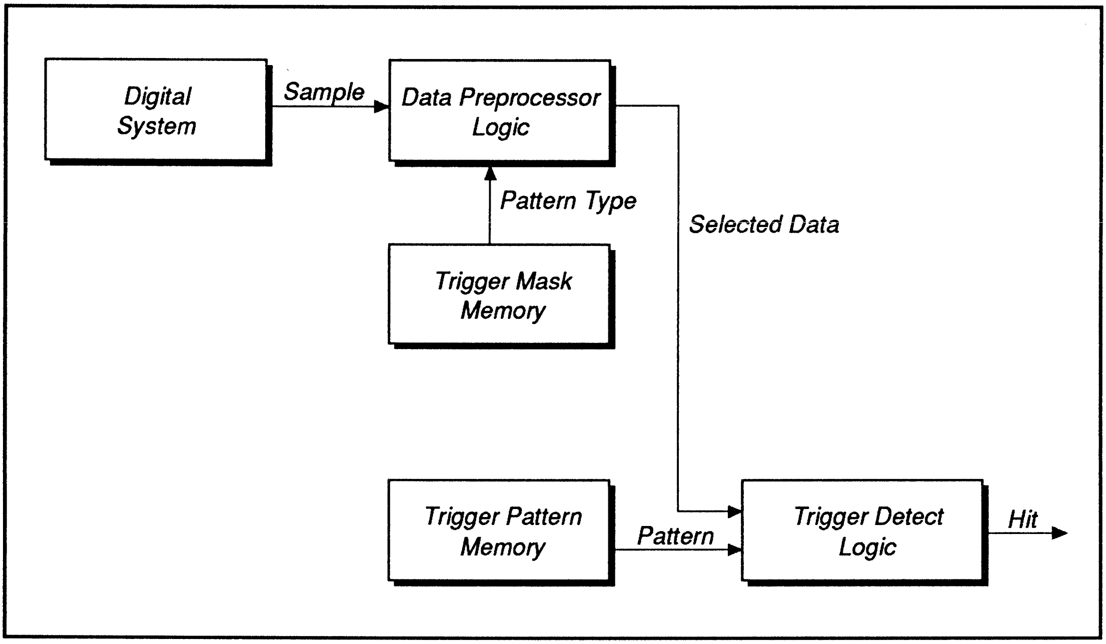
During the merge process, simulation commands are automatically removed from each file. It is best to produce an auxiliary simulation file[21] for each interim file so you can test subsidiary state machines independently.

- To simulate interim files, you copy the simulation segment from the original file into an auxiliary file and simulate.
- To verify the final design after each addition, you create an auxiliary simulation file for the combined design.
- To verify the final design after merging all files, create a single auxiliary simulation file to test the entire chip.

[21] Refer to the *PALASM 4 User's Manual*, Chapters 6 and 9, for details.

5.2 SINGULAR CONTROL FUNCTION

Trigger-detection sequential state-machine logic must be supported by combinational logic that transforms the input samples into architectural primitives. The following functional block diagram places the trigger-decoding logic in the context of the LSA data flow. The decode logic reduces multiple signals to a single evaluation, detect-or-no-detect; an example of singular control functions.



Trigger Detector Block Diagram

5.2.1 Trigger Detection Analysis

The Sample signal and Trigger Mask Memory block feed two circuits in the Data Preprocessor block. These two circuits are among the architectural primitives mentioned earlier.

- A decoder in the Data Preprocessor block must translate the input-signal patterns to combinations of signals that represent levels and edges.

- A filter in the Data Preprocessor block, fed by the Trigger Mask Memory, must screen for the occurrence of rising edges, falling edges, high levels, low levels, or glitches captured by the preprocessor.

When a signal pattern matches the mask condition, the pattern appears on the selected data bus. The Trigger Pattern Memory stores patterns that indicate each signal to be considered. Trigger Detect Logic compares input signals with user-defined patterns to determine when they match. When a match occurs, the controller looks for the next coincidence. Tracing begins when the last coincidence occurs. Each match is called a hit.

5.2.2 Singular Control Implementation

The preceding figure has two blocks indicating logic. The remaining blocks address memory or the external system under test. This design explicitly excludes memory functions as candidates for MACH implementation and the external system lies outside the scope of LSA architecture. That leaves the Data Preprocessor and Trigger Detect functions for current consideration.

The chief functions of the logic blocks are to convert samples to architectural primitives and to compare those primitive patterns. Since the basic architectural primitives have already been identified, in discussion 3.3, the function of these logic blocks can be readily implemented by combining the primitives in a single logic circuit.

The Dual-Bit Condition/Decode Logic figure shows an example of such a circuit. It is configured as the circuitry required to detect the logic states for two bits of the input signals. This logic is essentially a decoder in the binary sense.

The output lines are labeled to show which type of signal is detected.

- Input signals are independent of one another.
- Output signals are independent of one another.

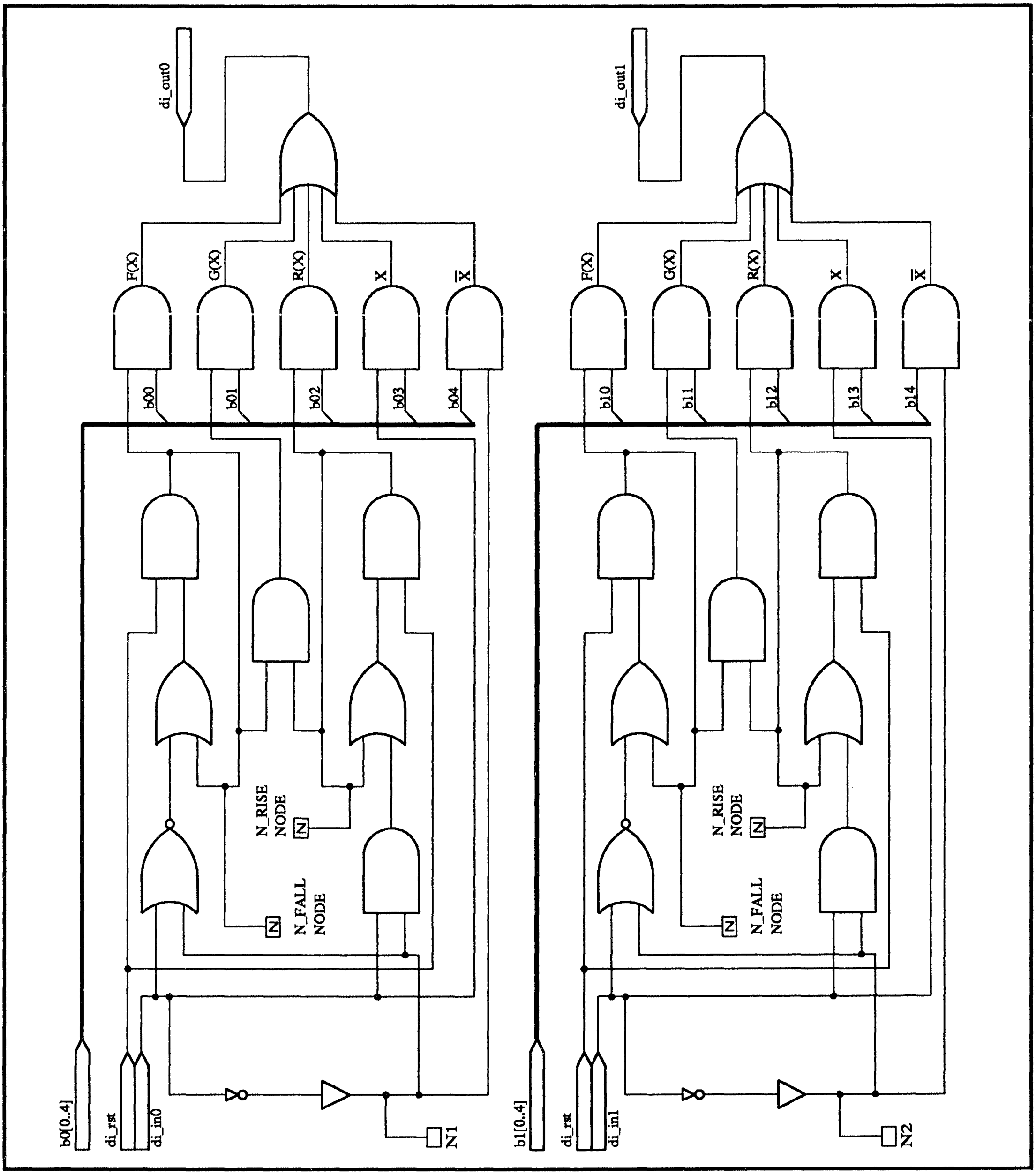
Due to signal independence, the logic associated with a single bit represents a unit, or, in other words, the bit-slice of the logic that decodes input signals.

A small number of input signals fan out to a larger number of output signals. The fanout property is the attribute that informs you there is no chance for unit reduction at this stage. Each output must be masked with the user-specified trigger conditions. Including more logic with this unit would increase the I/O count because the mask bits also have to enter the chip.

From an I/O standpoint, the chosen partition uses six pins per unit: one input pin and five output pins. You'd expect to get six units in a MACH 110 device. To get a rough idea of how many actually fit in the device, you define the two bits as a schematic-based design for the device and compile it.

The logic in the next figure was generated using OrCAD/SDT III with the AMD-supplied MACH library and using the PALASM 4 software.

Note: As you can see in the figure, wherever you have a feedback loop, you must add a node macro, such as N_RISE or N_FALL, to ensure an extra pass around the logic array before completing the loop.



Dual-Bit Condition/Decode Logic

The schematic was then compiled using the PALASM 4 software, as usual; the results of the compilation appear next. The device-resource segments of the MACH report show these two bits use about one third of the resources for the entire MACH 110 device or half that for a MACH 210 device. This verifies the earlier estimate.

```
*** Device Resource Checks
                 Available      Used    Remaining
Clocks:          2              2       0
Pins:            38             7       31     —>     18%
I/O Macro:       32             4       28
Total Macro:     64             14      50
Product Terms:   256            14      200    —>     20%

MACH-PLD Resource Checks OK!
:
Device Utilization........*:14%
```

MACH Report Device-Utilization Statistics

5.2.3 Adjusting Design-Portion Size

After choosing a portion of the design, either a function leaf or a feature leaf, for implementation and verifying it will fit in a MACH chip, you should ensure the size is optimal. The input-decode logic fits in a MACH chip, as shown by the trial compilation. You double check the appropriateness of the initial choice by considering changes to the pin count and logic content of the portion selected.

Consider changing the size of the input-decode logic. The LSA requires a mask to indicate which input-sample bits to include in a trigger cycle. If you added mask logic to the input-decode logic, the I/O count changes only slightly. The five outputs would be replaced by five inputs; one additional output, bit_coincidence, would be added. Again, from an I/O standpoint, you could expect five units to fit in a single MACH 110 device.

However, the additional function increases the logic count by six gates per bit, which is about a 50% increase in the number of gates per bit. Such an increase in logic reduces the number of bits per device by three from the number you'd get if you did not add the logic. Thus, the slight increase in I/Os would be accompanied by a significant increase in logic requirements. This indicates the unit count would be smaller than you'd expect based on I/O count alone.

The MACH 210 device supports twice as much logic as the MACH 110 device. You can fit the additional logic and meet the optimum partitioning on a bits-per-device basis using a MACH 210 device. The resolution hinges on the difference in dollar cost between the two devices.

The next discussion focuses on integrating the two domains.

6 DATA AND CONTROL INTEGRATION

After implementing both data-flow and control-flow logic, you complete the design by integrating the two domains. Again, only a segment of the design is discussed here; you repeat these techniques to complete the integration.

The control logic for the trigger memory is an example of a subsidiary state machine. You may recall, this machine is implemented in the file named LA_C0.PDS, shown earlier. In LA_C0.PDS, the sequences of state changes required to load the trigger were implemented; however, the actual signals that control the memory chips were not. At this point, you integrate the total design by implementing signals to control the memory chips.

This design uses static RAM for the trace memory. Static memory chips have two control inputs: a chip select and a read/write line. The control circuitry also regulates the trigger-address counter. Control lines are identified below.

• G_CS	global chip select
• G_WE	global write enable
• ADDR_CK	address clock

You add control outputs to pin declarations in the PDS file by placing the name of the output line in appropriate pin statements as follows.

```
PIN ? G_CS COMBINATORIAL ;global chip select
```

Again, the question mark, ?, in the location-number field specifies a floating pin number; the actual pin number is assigned automatically during compilation. Using the word combinatorial in the storage field defines the output to be non-registered. Everything after the semicolon is a comment and is ignored during processing; in the example above, the comment reminds you of the global nature of the signal. In this case, global means the signal affects more than one chip.

The control outputs become active at a certain time in the memory-access cycle, which goes through the steps below. Each step is a state in LA_C0; the outputs occur at a specific state.

A. The address is presented to the memory.

B. A delay occurs, which allows the data to appear on the memory outputs.

C. The data that appears is latched and the address can change to select the next trigger.

The classical attributes of associating an output with a state independent of input belong to a Moore-type state machine. That's why you see the MOORE_MACHINE keyword and a Moore-type output definition in the PDS file.

```
MOORE_MACHINE                 ;Read Trigger State Machine
M_C0_0 = /K_C0_1*/K_C0_0      ;C0 Control State Defined
M_C0_1 = /K_C0_1*/K_C0_0
M_C0_2 = /K_C0_1*/K_C0_0
M_C0_3 = /K_C0_1*/K_C0_0
```

Moore Machine

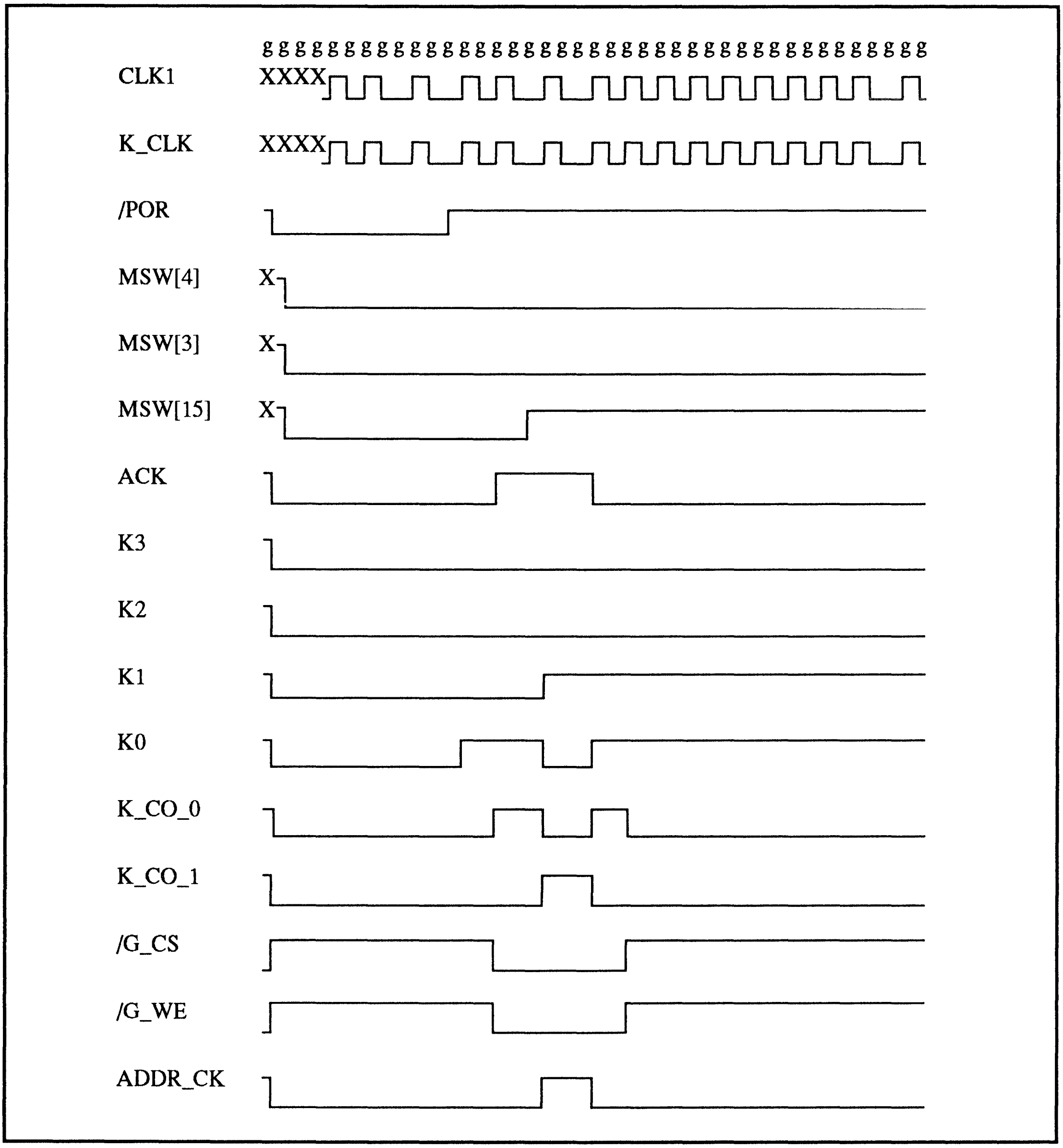
Moore machines produce a single output per state. The output may be the values of several variables. There is no reason to limit the Moore-state output to a single pin. You define the Moore output by using a Boolean term to specify the output value associated with the state. The memory-control equation for the third state is shown next.

```
M_C0_0.OUTF = /G_CS*/G_WE*/ADDR_CK
M_C0_1.OUTF =  G_CS* G_WE*/ADDR_CK
M_C0_2.OUTF =  G_CS* G_WE*ADDR_CK
M_C0_3.OUTF = /G_CS*/G_WE*/ADDR_CK
```

Memory-Control Equation

The outputs with names on the right side of the equation are true during the state, named M_C0_2. Another way to interpret the equation is to realize you are assigning active-high values to G_CS, G_WE, and ADDR_CK. When G_CS has an active-high value, its complement, /G_CS, has an active-low value required at the static RAM.

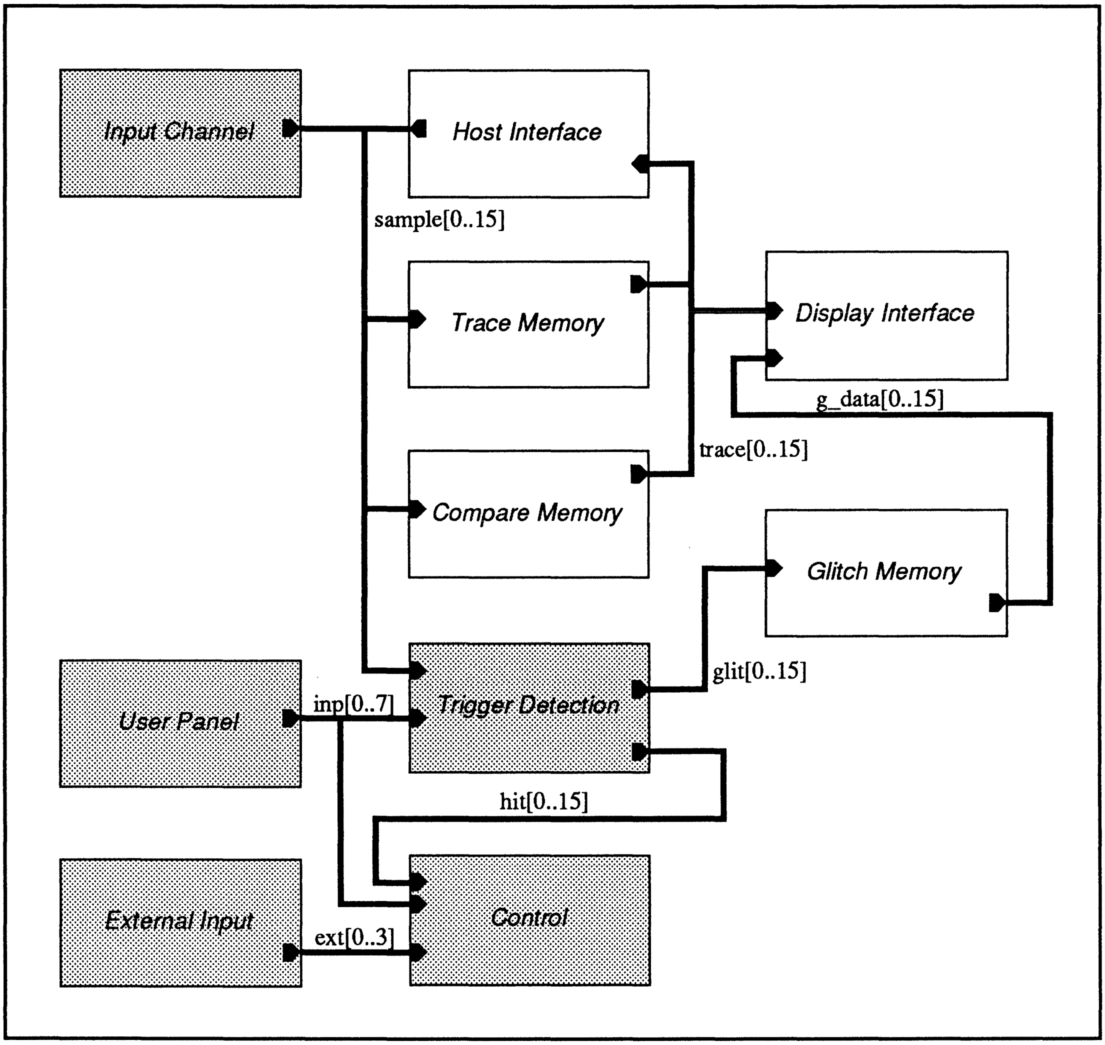
Compilation and simulation show the output pins assigned to the new signals do change at the proper state. You repeat the techniques above for each point in the data flow controlled by a state machine. Output statements are defined in the PDS file that corresponds to the controlling state machine. The next figure shows the simulation results.



Simulation Results

6.1 MACH CHIP INTEGRATION

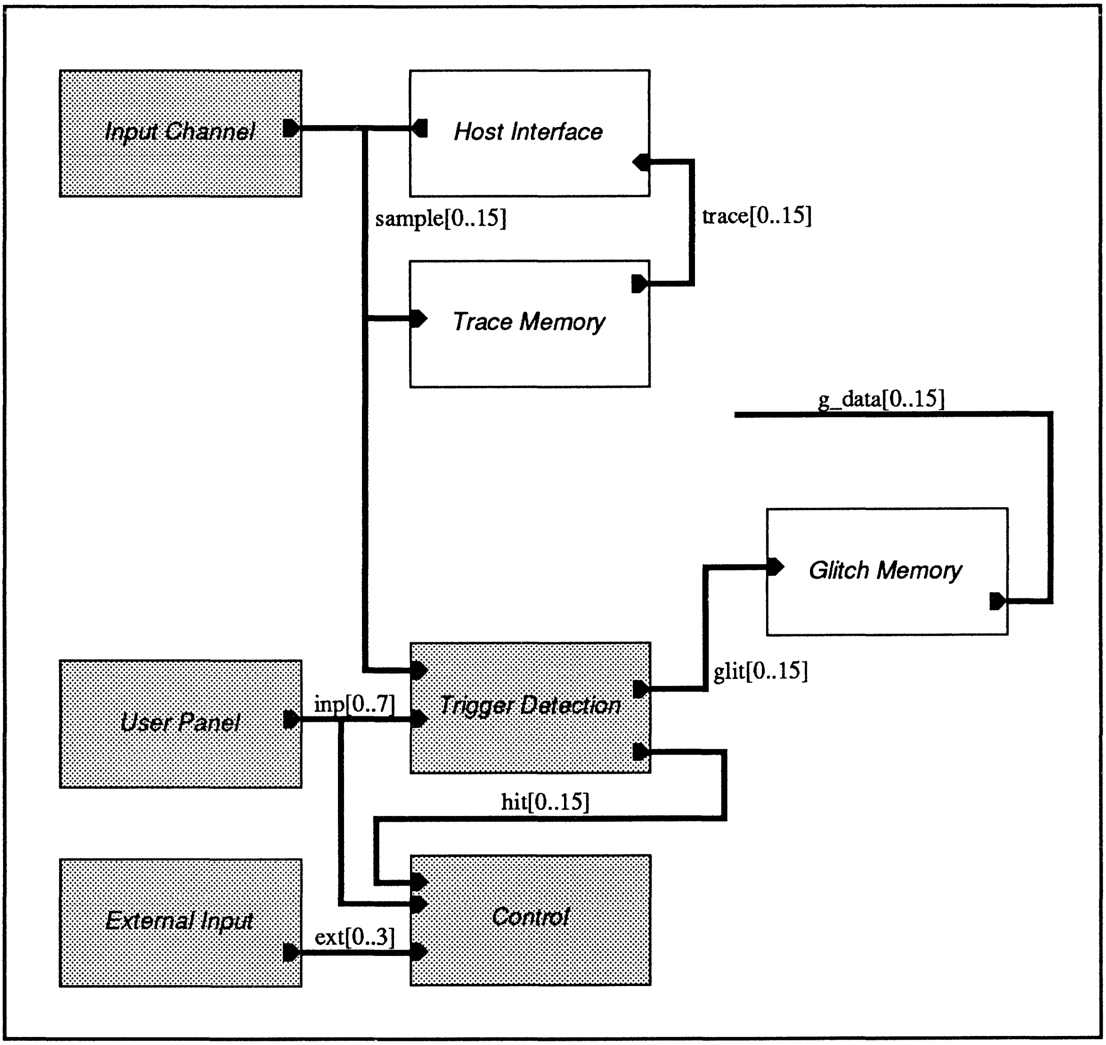
When you complete data- and control-flow domain integration, the stage is set to integrate the design into MACH chips. Shaded blocks on the next block diagram represent areas for which you have created MACH designs.



LSA System Block Diagram

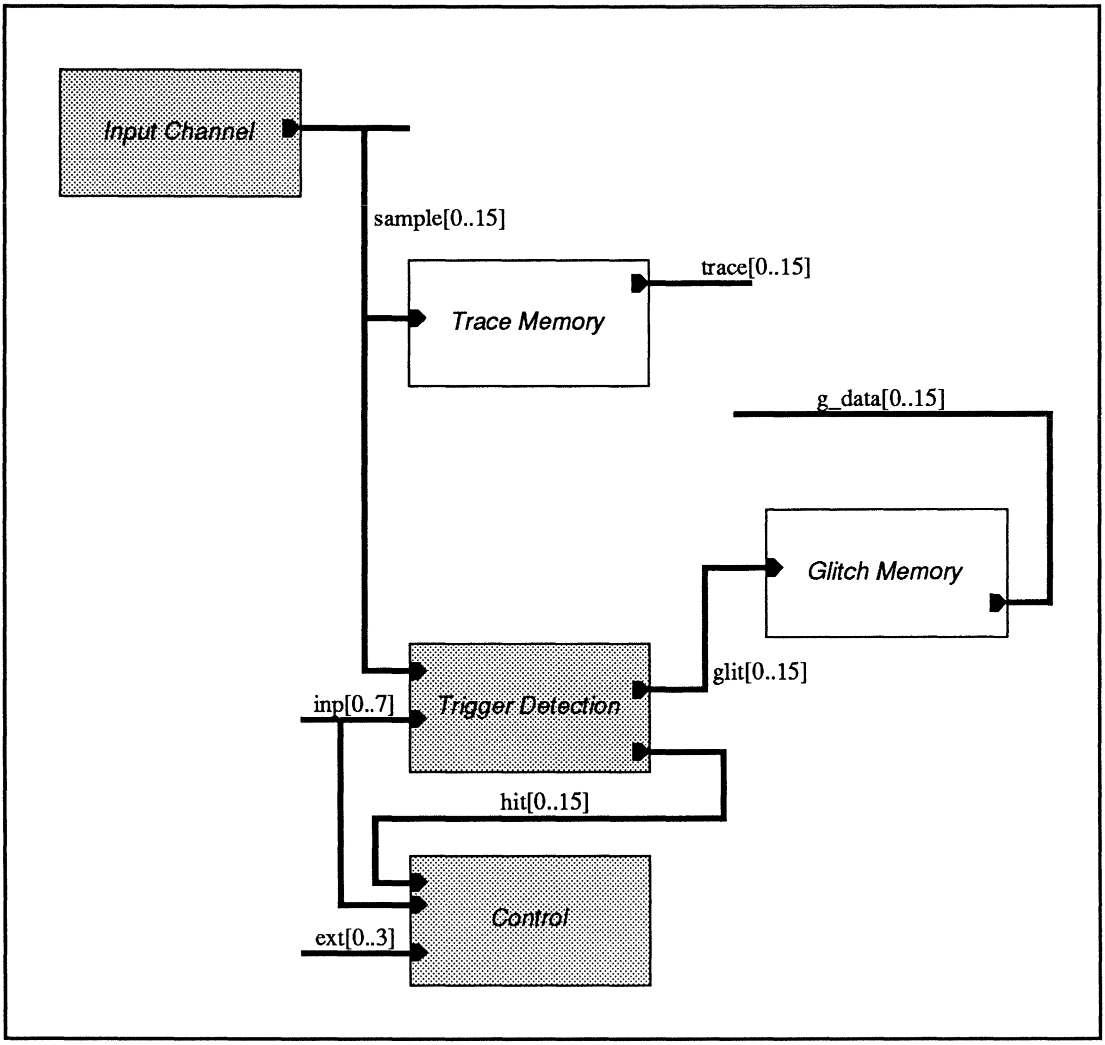
Functions in the previous figure are not yet associated to particular chips, which is the next step.

The next figure shows the block diagram after removing functions you can implement in the host system, using software and host-system memory. The host system can also take care of display needs. Remaining memory-block data registers are still MACH candidates.



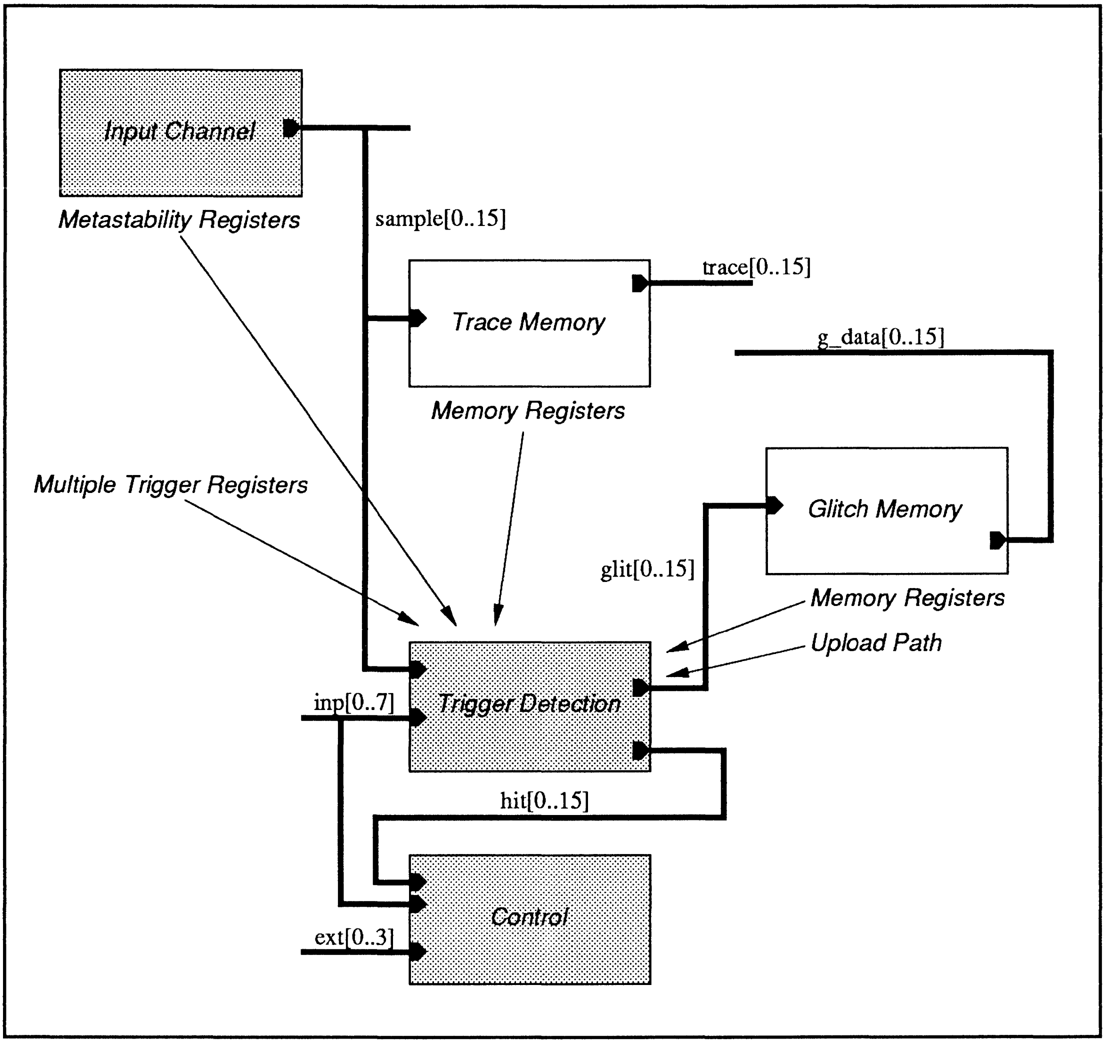
LSA System without Host-System Functions

The next figure shows the minimal block diagram for MACH chip implementation, which results when you remove the user-panel and external-input functions from the previous system diagram. The host-interface depends on the actual processor; the user-panel and external-input functions are among the options you can add. Design files for these functions are included on the PALASM 4 installation diskettes.



LSA System without User-Panel and External-Input Functions

The remaining functions, shown next, are the complete set to be implemented on MACH chips to realize this LSA design.



LSA System for MACH Implementation

6.2 TRADEOFFS AND REDESIGN STRATEGIES

The fitting process is iterative and involves tradeoffs and the following re-design strategies.

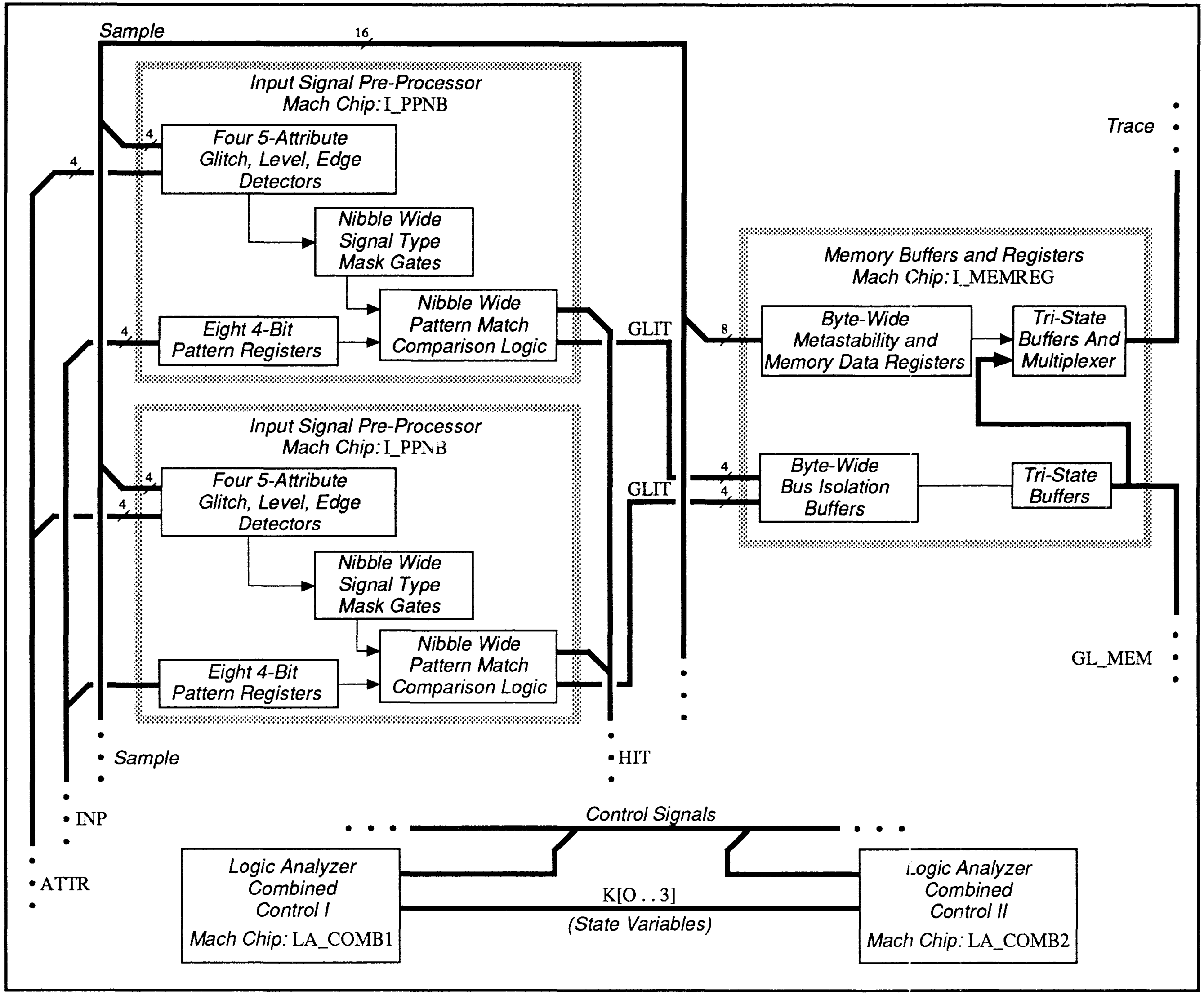
- TTL macro registers, the original choice for data registers, were ultimately replaced with MACH flip-flops to maximize functional density.

- The drastic reduction of the data flow leaves no path from the Glitch Memory to the Host, except with additional logic.

  The g_data bus [0..15] was dropped altogether and the glit[0..15] bus was changed to be three-state and bidirectional to accommodate host upload of Glitch Memory data.

The final results of integrating the LSA design to MACH chips appears in the next block diagram. These chips are used multiple times to realize the LSA design, as shown in discussion 1 and in the LSA schematics provided on the PALASM 4 installation diskettes.

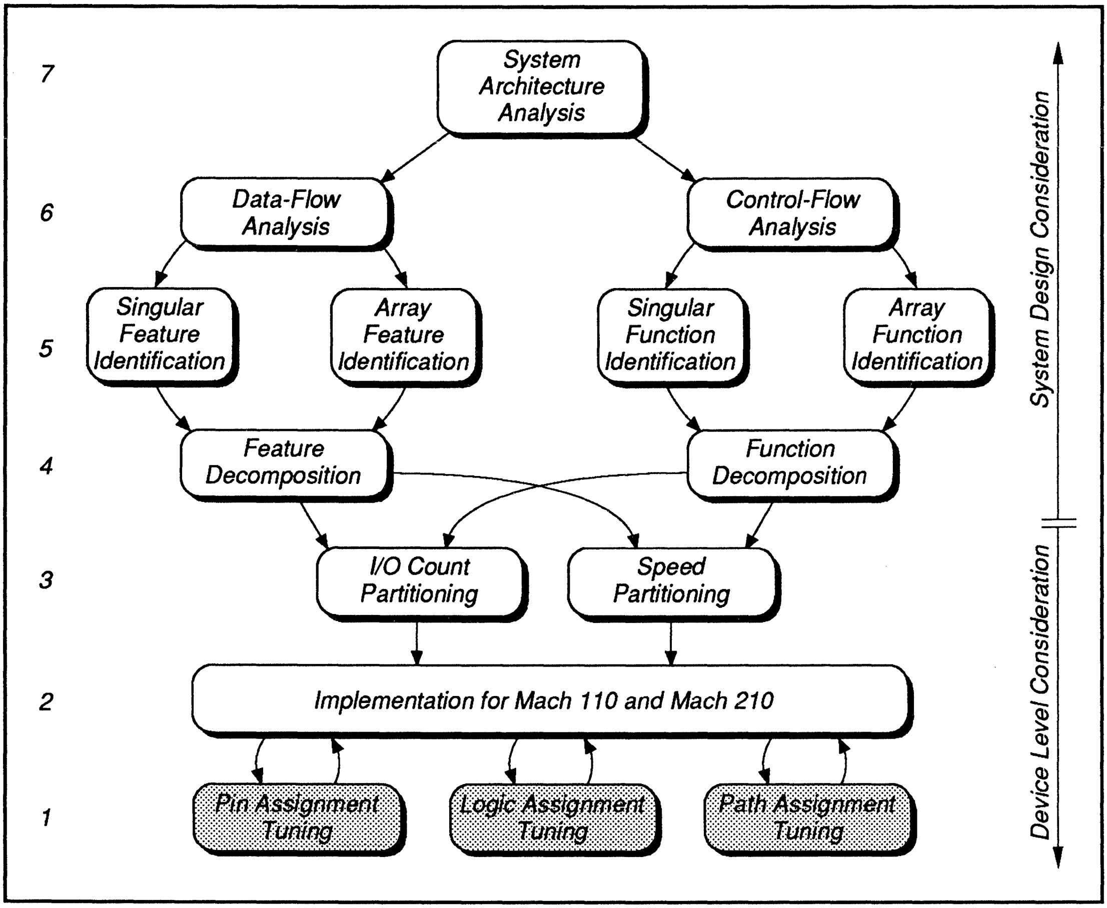
MACH-Based LSA



7 TUNING

Can some of the ideas you just used be applied to optimize this LSA design? How can you tell if optimization is even possible? Classically, at this point you'd begin the tuning process by reviewing the MACH report.

Again, the MACH report describes the results of the fitting process and includes information you can use to determine the degree to which each of the final designs fits on the selected chips.


7
System Architecture Analysis
6
Data-Flow Analysis
Control-Flow Analysis
5
Singular Feature Identification
Array Feature Identification
Singular Function Identification
Array Function Identification
4
Feature Decomposition
Function Decomposition
3
I/O Count Partitioning
Speed Partitioning
2
Implementation for Mach 110 and Mach 210
1
Pin Assignment Tuning
Logic Assignment Tuning
Path Assignment Tuning
System Design Consideration
Device Level Consideration

Tuning Phase of Design Process

The tuning phase is typically divided into three stages.

- Locate and correct non-optimal pin assignments.
- Reorganize non-optimal logic assignments.
- Reposition non-optimal path assignments.

Due to the bit-slice nature of this LSA design, tuning occurs at a higher level. Discussion 7.1 summarizes resource allocation for this design. General tuning considerations are discussed under 7.2, 7.3, and 7.4.

7.1 LSA RESOURCE-ALLOCATION SUMMARY

You may recall, earlier partial-fitting processes were used to develop the size of various bit slices. After designing all bit slices for this LSA design, you look for the optimal aggregation.

During the first attempt to fit an LSA bit-slice following the integration of the two domains, a 16-bit word of data-flow logic is compiled for a single MACH chip. The word-sized aggregation is selected on the basis of pin counts: one word yields 16 pins in and 16 pins out for data flow. Data registers are chosen as the data-flow elements because they fit on a single MACH chip and leave lots of unused storage and logic resources. However, only six pins are left for use by other functions.

Although resource use is clearly not optimal, the trial does allow for a quick sizing. It's obvious the chip count can be improved by choosing another bit-slice combination.

Based on the initial sizing of the two bits of preprocessing logic, which require about one-third of a single MACH 110, the next bit-slice combination that's chosen is a byte. The rationale here is the array structure of the design is sufficiently regular that the fitting algorithm might be able to squeeze eight bits into a chip instead of the expected six, or two bits per third. This is also a good test of the degree of slack in the fitting algorithm.

The result is 98% use of MACH resources; however, two signals could not be routed automatically. Subsequent runs reduce the number of unconnected signals to one. The chip is upgraded to a MACH 210 and a fit occurs immediately.

After concluding the explorations above, the natural-sized bit unit becomes a four-bit nibble. The nibble is chosen because it is smaller than the optimal six bits and because of its standardization in the digital world.

Fitting nibble-sized data and control flow on the chips consists of creating and compiling files that contain four-bit slices of the required elements rather than one. Each time the fitting process is successful, you review the MACH report to find out how much space remains on the chip; then you use the percentage of remaining resources to determine whether more logic can fit on the chip. It is also important to look at the physical layout of the chip presented in the feedback-map and logic-map segments of the MACH report. These pictures give you an idea of how well the logic is clustered into the chip.

The partitioning technique used in this design was so successful that subsequent additions of logic to a chip barely disturbed portions that were already in place. When 70% of the chip's resources are used, simply adding logic and compiling can produce diminishing returns. That's when you use specific MACH fitting options to help with logic placement.

For example, most fitting options are initially disabled for this LSA design and only the one below was used.

```
FITTING OPTIONS
When compiling                      Run until first success
```

More logic is added as long as the signals can be routed during fitting. When no paths are available, the following options were used.

```
FITTING OPTIONS
When compiling                          Select one combination
Maximize packing of logic blocks?                Y
```

More functions are added until paths are exhausted again. Then another option is enabled, as shown below. Empty parts of the placement map begin to fill in, which results in an 85% utilization.[22]

```
FITTING OPTIONS
When compiling                          Select one combination
Maximize packing of logic blocks?                Y
Expand small PT spacing?                         Y
```

The final option, Expand all PT spacing, is not needed because no more logic is needed on the chip.

You can enable the gate-splitting option to automatically split wide terms into sizes that match the maximum size for the selected chip.[23]

Another case of gate splitting you should know about is a by-product of the minimization process during compilation. The more product terms you can include in a single pass through the logic array, the faster the resulting implementation. Each pass through the array adds a 15 or 20 ns propagation delay, which is sound justification for meeting the objective of single-pass minimization. The software ensures a single pass through the array by converting all pure combinatorial specifications, where no storage elements are involved, to a sum of products. Each sum then corresponds to a product term for the logic array.

[22] Designs that require up to 70% of MACH-device resources can be achieved with very little effort. This LSA design shows MACH-device utilizations of greater than 70% can be achieved using various combinations of language syntax and software fitting options. The degree of fit varies from design to design.

[23] Refer to the *PALASM 4 User's Manual*, Chapter 9, for details about this option.

This LSA design once included a 16-bit OR gate that collected hit conditions to indicate the presence of a trigger event. Each hit condition consisted of other combinational terms. When the resulting Boolean equation was reduced to a two-level sum of products, the equation had more product terms than allowed for a single equation. An error occurred when the design file was compiled.

If a schematic-based file contains separate, distinct gates that result in too many terms, an error occurs when it is converted to Boolean equations.

The following figure shows a PDS file segment. The equation for MATCH has too many terms.

```
EQUATIONS

MATCH = (((_3_M29_2 * NB_AT0) + ((_3_M29_2 * _3_M19_2) * NB_AT1) + (_3_M19_2
    * NB_AT2) + (NB_IN0 * NB_AT3) + (NB_AT4 * /NB_IN0)) * PAT0) +
    (((_3_M35_2 * NB_AT5) + ((_3_M35_2 * _3_M20_2) * NB_AT6) + (_3_M20_2
    * NB_AT7) + (NB_IN1 * NB_AT8) + (NB_AT9 * /NB_IN1)) * PAT1) +
    (((_4_M29_2 * NB_AT10) + ((_4_M29_2 * _4_M19_2) * NB_AT11) +
    (_4_M19_2 * NB_AT12) + (NB_IN2 * NB_AT13) + (NB_AT14 * /NB_IN2)) *
    PAT2) + (((_4_M35_2 * NB_AT15) + ((_4_M35_2 * _4_M20_2) * NB_AT16) +
    (_4_M20_2 * NB_AT17) + (NB_IN3 * NB_AT18) + (NB_AT19 * /NB_IN3)) *
    PAT3)

_3_M29_2 = DI_RST * (/(NB_IN0 + /NB_IN0) + _3_M29_2)

_3_M19_2 = (_3_M19_2 + (NB_IN0 * /NB_IN0)) * DI_RST

_3_M35_2 = DI_RST * (/(NB_IN1 + /NB_IN1) + _3_M35_2)

_3_M20_2 = (_3_M20_2 + (NB_IN1 * /NB_IN1)) * DI_RST

_4_M29_2 = DI_RST * (/(NB_IN2 + /NB_IN2) + _4_M29_2)
```

Too Many Product Terms for MATCH

To correct this kind of situation, you can manually split the equation into several distinct equations. The next equation is an example of the fix.

MATCH = GRP1+GRP2+GRP3+GRP4

When the equation was converted from schematic-based information, all the terms in each of the GRP1 through GRP4 equations were lumped into the MATCH Boolean equation. Simple redefinition solves the problem, as shown below.

```
EQUATIONS

     cmp_end =/pat0+/pat1+/pat3+/pat3

     MATCH =GRP1+GRP2+GRP3+GRP4

     grp1=  (((_3_M29_2 * NB_AT0) + ((_3_M29_2 * _3_M19_2) * NB_AT1) + (_3_M19_2
          * NB_AT2) + (NB_IN0 * NB_AT3) + (NB_AT4 * /NB_IN0)) * PAT0)

     grp2=  (((_3_M35_2 * NB_AT5) + ((_3_M35_2 * _3_M20_2) * NB_AT6) + (_3_M20_2
          * NB_AT7) + (NB_IN1 * NB_AT8) + (NB_AT9 * /NB_IN1)) * PAT1)

     grp3=  (((_4_M29_2 * NB_AT10) + ((_4_M29_2 * _4_M19_2) * NB_AT11) +
         (_4_M19_2 * NB_AT12) + (NB_IN2 * NB_AT13) + (NB_AT14 * /NB_IN2)) *  PAT2)

     grp4=    (((_4_M35_2 * NB_AT15) + ((_4_M35_2 * _4_M20_2) * NB_AT16) +
         (_4_M20_2 * NB_AT17) + (NB_IN3 * NB_AT18) + (NB_AT19 * /NB_IN3)) *
         PAT3)

     _3_M29_2 = DI_RST * (/(NB_IN0 + /NB_IN0) + _3_M29_2)

     _3_M19_2 = (_3_M19_2 + (NB_IN0 * /NB_IN0)) * DI_RST

     _3_M35_2 = DI_RST * (/(NB_IN1 + /NB_IN1) + _3_M35_2)

     _3_M20_2 = (_3_M20_2 + (NB_IN1 * /NB_IN1)) * DI_RST
```

Corrected Equations

The result of grouping the inputs allows the compilation process to finish. However, when you review the MACH report you see that the propagation delay increased by one unit from 2Tpd to 3Tpd, as shown next.

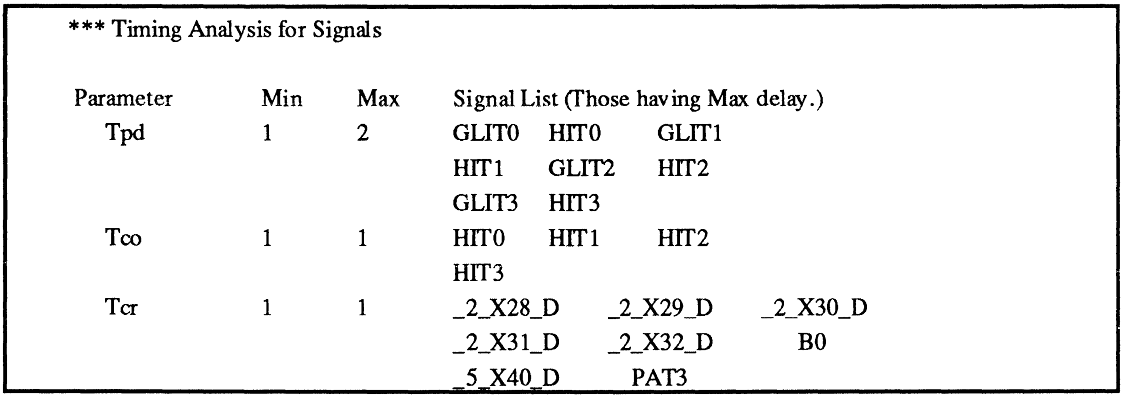
```
*** Timing Analysis for Signals

Parameter      Min     Max     Signal List (Those having Max delay.)
   Tpd          2       3      MATCH
   Tco          1       2      MATCH
   Tcr          1       1      _5_X9_D        _5_X10_D        _5_X11_D
                               _5_X12_D       _5_X13_D        B0
                               _8_X5_D        PAT3
.


Key:
Tpd - Combinatorial propagation delay, input to output
Tsu - Combinatorial setup delay before clock
Tco - Register clock to combinatorial output
Tcr - Register thru combinatorial logic to setup
All delay values are expressed in terms of array passes
```

Propagation Delay Increases After Grouping Inputs

The solution to the problem of added delay could well have been re-engineering the comparison logic to use pipeline parallel-processing techniques. In fact, the logic contained signals that were needed off chip. So both the match condition and the required signals were assigned to pins. That reduced the on-chip delay to 2Tpd and opened options to use off-chip logic with faster Tpd for the comparison. The lesson is to **consider the design globally, as well as on a chip-by-chip basis**. The next report segment shows the result of the pin assignment on propagation delay.



```
*** Timing Analysis for Signals

Parameter        Min     Max     Signal List (Those having Max delay.)
   Tpd            1       2      GLIT0    HIT0     GLIT1
                                 HIT1     GLIT2    HIT2
                                 GLIT3    HIT3
   Tco            1       1      HIT0     HIT1     HIT2
                                 HIT3
   Tcr            1       1      _2_X28_D     _2_X29_D     _2_X30_D
                                 _2_X31_D     _2_X32_D        B0
                                 _5_X40_D       PAT3
```

Pin Assignment Changes Propagation Delay

7.2 LOCATE AND CORRECT NON-OPTIMAL PIN ASSIGNMENTS

Non-optimal pin assignments occur when automatic resource allocation causes reduced logic capacity due to wiring congestion. You can often determine pin-assignment problems by looking at the signal segments, tabular and equations, of the MACH report for areas of the chip that contain the following.

- More functions than fit in a single block
- Functions associated with I/O pins that are not nearby

To correct non-optimal pin assignments, you can group logic into specific blocks of a MACH device using the appropriate reserved word, MACH_SEG_A through MACH_SEG_D, as a group name. This associates logic with specific blocks that are more conveniently located. Then you recompile the design.

7.3 REORGANIZE NON-OPTIMAL LOGIC ASSIGNMENTS

Non-optimal logic assignments occur when the automatic-placement algorithm has inappropriate or inadequate information. This results in logic groups not being optimally placed; large blocks of logic may be placed in areas of the chip that lack sufficient resources.

You review the logic-map segment of the MACH report to locate large blocks of logic placed in the corners of the chip. Logic located toward the center of the chip can be expanded in two directions. Logic in the corners poses a problem due to lack of resources.

To correct the problem, you can move logic using the Group command with the appropriate reserved word, MACH_SEG_*block*, as a group name. In addition, you can enable the following logic-synthesis option during compilation.[24]

```
Use automatic gate splitting?          Y  Max = #
```

7.4 REPOSITION NON-OPTIMAL PATH ASSIGNMENTS

Non-optimal path assignments occur when logic-placement decisions block routing paths to functions that must communicate. Usually, candidate functions are optimally placed. However, placement may be less than optimal when software algorithms do not identify related functions.

This type of problem can be detected by reviewing the fanout statistics and feedback map segments of the MACH report. These data reveal the degree of communication required by functions placed on the chip and can help you determine if a better placement can result in a fit.

[24] Refer to the *PALASM 4 User's Manual*, Chapter 5, for details about gate splitting during compilation and fitting, and to Chapter 11 for a detailed discussion on splitting functions.

8 COMPLETE LSA SYSTEM IMPLEMENTATION

The LSA design presented in this study includes the key functions needed for state triggering and logic tracing.[25] However, a complete logic analyzer needs other support functions, such as a keyboard interface, memory storage for trace attributes, trigger patterns, etc. You can add such functions to the logic analyzer in this study to customize it.[26]

Key analysis and tracing design functions are contained in two data-flow schematics and one control file.

- I_PPNB.SCH
- I_MEMREG.SCH
- LA_COMB.PDS

Each schematic is a hierarchical file, which includes sub-schematics that provide the details of a particular aspect of the data flow. The control file is a PDS file that contains the state-machine designs for the trigger and trace operations. This too is a hierarchical design, in the sense that a higher-level machine, LA_KMAIN, controls the lower-level machines, LA_C0, LA_C1, etc.

To view design discussions from the perspective of a completed design, you can assume that the final vehicle for the logic analyzer is a PC add-in card, like the one shown in the design description, under discussion 1. In this case, you combine into a single chip as much of the control function as possible and configure the chip interface to be compatible with microprocessor control. Generally, microprocessor-controlled chips have a control register loaded by the CPU via a command. The chip uses the data in the

[25] Files that support these functions are introduced throughout this study and are provided on the PALASM 4 installation diskette, as defined in Appendix A.

[26] Files to support additional functions are introduced in Appendix A and are also provided on the installation diskettes.

control register to implement the function represented by the code.

The control for this design is stored in a file named LA_COMB, which includes the six state-machine functions listed below.

- Logic to implement each of the four logic analyzer trigger and trace modes
- One machine loads the attribute memory
- Another machine loads the internal trigger registers on the preprocessor chip

This LSA uses internal registers to optimize trigger-detection speed and uses 5 attribute bits per trace signal. The 5-bit attribute condition determines a specific configuration for the attribute memory. Embedding these control functions with the core logic-analysis functions ensures the data-flow chips can be properly loaded and unloaded.

The **attribute memory** has a 20 bit word width per nibble sampled. Since the normal width of data sources is eight bits, the machine that loads the attribute memory assumes a data path composed of a 5-byte pipeline with taps for each bit's attribute set at the output of each byte. The machine loads the pipeline sequentially, then transfers the entire 20-bit word to the attribute memory in parallel. The logic accomplishing the attribute load is the same as the logic in the file LA_LD_AT.PDS.

The preprocessor chip contains eight **internal registers** that determine which bits of the input data to use for trigger detection. The internal registers are loaded from the pattern memory by the function in the file named LA_RLOAD.PDS, which is embedded with other logic in the combined file, LA_COMB.PDS. The logic assumes patterns are a maximum of eight triggers long. The final trigger of a sequence is followed by a zero trigger pattern. Thus, to trigger on three patterns, the pattern memory should be loaded with the three patterns followed by an all zero pattern.

All functions embedded on the control chip can be accessed by writing the following patterns to the MSW bits: MSW[5], MSW[4], MSW[3].

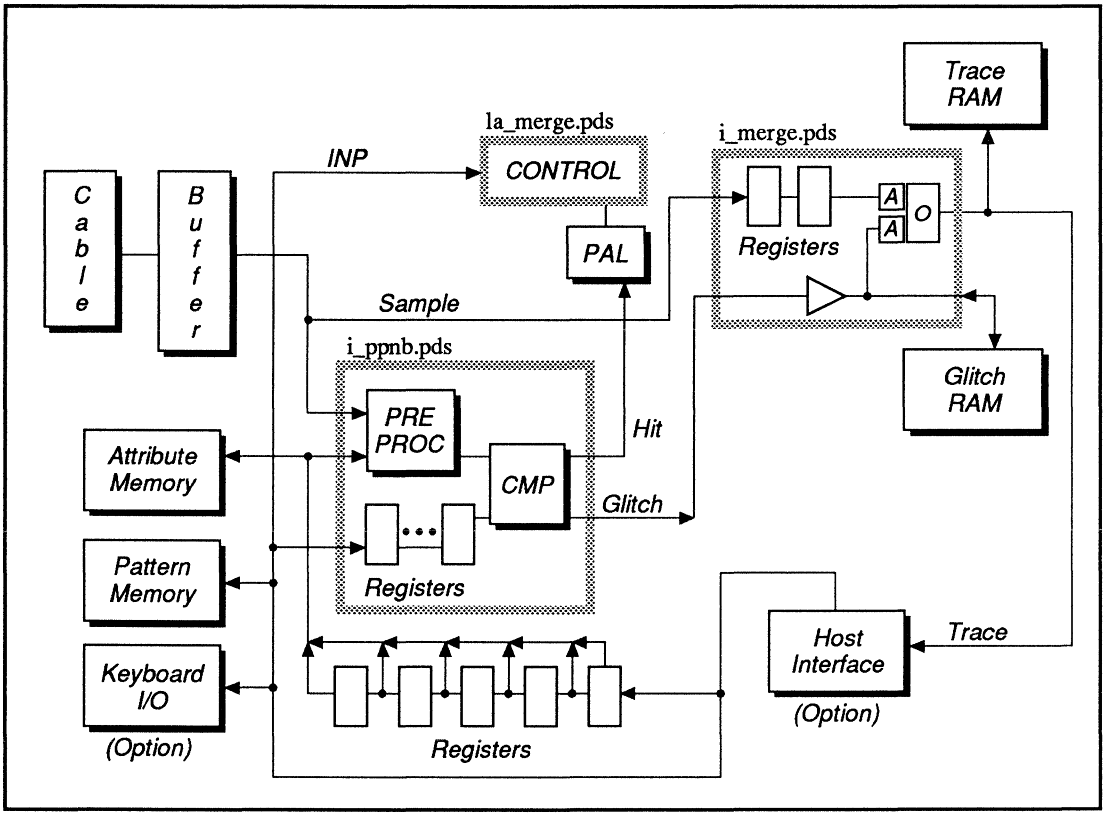
MSW	
543	FUNCTION
0 0 0	Trace During Detect
0 0 1	Trace To Detect
0 1 0	Trace After Detect
0 1 1	Trace Between Detects
1 0 0	Load Internal Registers
1 0 1	Load Attribute Memory
1 1 0	Reserved
1 1 1	Reserved

MSW Patterns

8.1 IMPLEMENTATION

The key to a compact design is to find the basic functions to build the total system. Attempts to implement the functionality of the original preliminary data flow immediately, without completing the iterative process described herein, results in the use of more chips than the final count.

The following figure shows the final implementation of the logic analyzer using the chips defined in this study. The final architecture embeds many of the functions, such as metastability registers, attribute registers, and pattern registers, into data flows internal to a chip. This highlights the iterative nature of chip design and resource fitting.



LSA Implementation on an Add-In Card

8.2 RE-ENGINEERING CONSIDERATIONS

Design assumptions determine the final chip implementation. If your LSA design involves a different set of assumptions from those in this study, your final design can differ subtly or dramatically from the previous figure.

Slight differences mean you can probably re-engineer the design using existing files as a basis for your work. For example, you can change the logic to account for the uses of different memory chips, or change the pin out to account for layout constraints.

The ultimate in modification support lies in MACH-device reprogrammability. Also, the PALASM 4 software supports modification of MACH-resource use even after fitting. Depending upon the available resources, you can

- Change the logic inside a chip design and keep current pin assignments.
- Keep the same internal logic and change pin assignments.

Defining pin locations is called annotation. If you float pin locations in a design, the PDS file contains a question mark in the location-number field rather than a specific pin location. During compilation and fitting, specific pin assignments are made automatically and recorded in one segment of the MACH report. Later, you can back annotate[27] to automatically write pin assignments from the last successful placement in the location field of pin and node statements in the PDS file. If an error occurs, data is stored in a design.PBK file and the PDS file is not updated.

[27] Refer to the *PALASM 4 User's Manual*, Chapter 9, for details about the Back annotate signals command.

The files LA_BKCHG.PDS and LA_BKPIN.PDS are back-annotated versions of the combined control file, LA_COMB.PDS.

- LA_BKCHG.PDS shows the primary state-machine logic of LA_KMAIN.PDS is changed.

- LA_BKPIN.PDS shows the logic of the main state machine is not changed but pins have been swapped: Pin 17, /AM_G_WE and Pin 21, /PM_G_WE.

In reviewing the MACH report for LA_BKCHG, you can see the new design compiles, is assigned logic, and fits in the chip while maintaining the same pin outs as LA_COMB.PDS. LA_BKPIN also compiles, is assigned to chip resources, and fits with the requested pin changes.

Changes should be restricted to functions that use the same block. In general, it is safe to change logic because the software groups all logic that pertains to a particular function. The same general rule applies to pin changes. Thus, intra-block changes can usually be achieved without problems. As the design grows and uses more chip resources, macro cells, I/O cells, and wiring channels, it becomes more difficult to change the design and maintain pin assignments. In this case, it may be necessary to move entire logic functions to blocks adjacent to the target pins to maintain former pin assignments.

For any particular design and resource-use combination, it may not be possible to maintain a former pin out. In such cases, the job may not be possible at all. The way to approach it, however, is to successively relax the constraint that all pins must remain as assigned. You do this by converting one specific pin assignment to a question mark, ?, then compile and fit. Repeat this procedure with individual pins.

9 DESIGN REVIEW

Previous discussions focused on the design process in the context of a specific LSA implementation. Now it is time to consider the forest rather than the trees. What can be derived that is general and lasting? Following discussions review the paths taken and highlight useful items for this, and other, designs. Tuning to optimize the design is also discussed.

During the course of this study, you have seen how to take an idea and refine it successively to the point where you can be sure parts of the design fit into a single MACH chip. It's worth reviewing some of the recurring themes in this design process, which can form the basis of your personal design kit when you use MACH devices to realize your own designs.

9.1 SYSTEM CONSIDERATIONS

The lasting part of the system considerations lie in how the structure leads to the chips. The entire purpose of evaluating system-related factors was to find the pieces that would fit into a single MACH chip. The fit is determined first by the pin count of the chosen pieces, then by the product terms. If a function has more pins than are supported on a chip, it doesn't matter that the logic may require only a single gate.

Once the design is split into MACH-sized pieces in terms of pin-out requirements, you can focus on the logic requirements. In this case, logic refers to the combinatorial logic and storage elements. Logic must be divided so the requirements of finished pieces are lower than the resources available in a single device. It's fairly easy to find parts of the design that do not overflow available resources. Given that, the entire design can be implemented immediately. However, what is not straightforward is finding pieces that minimize the number of chips required to implement the design while maximizing the obtainable speed.

9.2 LOGIC ASSIGNMENT

The benefit of the system-partitioning technique in this study is that it leads to design slices that are optimally sized for the MACH device. If one bit-slice combination does not result in the best fit, it's easy to scale the design for another fit using fewer or different bit slices. You use the following MACH fitting option for initial fits.

FITTING OPTIONS	
When compiling	Run until first success

During the tuning phase, you can use different options to pack product terms as closely as possible and to adjust spacing for product terms. For example, space can be left for functions with lots of internal connections by enabling one of the expand PT spacing options on the MACH Fitting Options form.

- Expand small PT spacing allocates an empty macrocell between those that contain small product terms, which means those with four or fewer variables that fit into a single macrocell.

  Large product terms have more than four variables and require more than one macrocell.

- Expand all PT spacing allocates an empty macrocell between each used macrocell.

Design slices obtained from splitting the system data flow according to array and singular function content are already separated into two groups: those with many internal connections and those with only a few. Portions of the design are placed on certain areas of the chip based on how much they communicate. This design process helps you split the logic into pieces with minimal communication, which ensures you can fit a piece of the design into any MACH chip with enough space remaining. Actual placement can be controlled using logic-block assignment commands, such as GROUP MACH_SEG_*block*, followed by the signal list.

In addition, during the fitting process, a measure of intra-function communication, called affinity, is calculated automatically and used to assign logic to areas of the MACH chip. Strong affinity keeps logic grouped together; little or no affinity allows arbitrary placement. Control-flow logic reflects logic with strong affinity. Data-flow logic reflects low affinity. In the lateral sense, data-flow signals do not cross one another.

9.3 STATE MACHINES

When designing state machines, partitioning is largely a matter of personal style. Complex machines with many states should generally be decomposed to multiple state machines that cooperate to achieve the desired operations. When you can mentally keep track of all states in the machine, the simplest method is to use state-transition equations, like the one shown below, to create the design file.

```
M_C0_2 := NULL_TR -> M_C0_3
```

During compilation, variables are automatically assigned to states, and default values are assigned to unused states, according to the rules you set. When you use state-transition equations, associated output equations are activated during compilation according to the states and your specifications. All states are defined and all outputs are created accordingly. The execution-log file tells you which values were assigned, as shown next.

```
I>WARNING E1351 Automatically assigning state bit _ST0 to ?NODE
...
STATE REGISTERS USED
---------------------------------
PIN NUMBER:    PIN NAME:
? NODE         _ST0
? NODE         _ST1
? NODE         _ST2
? NODE         _ST3
STATE BIT ASSIGNMENT USED
-----------------------------------------
STATE NAME:              STATE REGISTERS VALUES:
                         _ST3    _ST2    _ST1    _ST0
M_K0                     0       0       0       0
M_K1                     0       0       0       1
M_K2                     0       0       1       0
M_K3                     0       0       1       1
M_K4                     0       1       0       0
M_K5                     0       1       0       1
M_K6                     0       1       1       0
M_K7                     0       1       1       1
M_K8                     1       0       0       0
```

Partial Execution-Log File Detailing State Assignments

The entire MACH device is treated as a single entity for state design; all state-machine flip-flops are considered to be a part of one and the same machine. However, there are times when all your state-machine flip-flops are not part of the same state-machine design. And there are times when you have too many states to track easily. In such cases, you design individual state machines in separate files and merge them into one file. Each state machine alone is not likely to make efficient use of a complete MACH device.

- Keeping some state machine values independent is good practice when you need to know the state of the system for status or debugging purposes.

- Keeping state-machine values independent was essential for this design.

By implementing independent state machines in this LSA design, machines could be placed into any MACH chip with enough remaining space, regardless of the presence of other state machines on the chip. In fact, the main control chip was designed in just that fashion.

Important: **Do not** just copy all transition equations into a single PDS file to merge designs. Errors will occur when you compile the file.

You use the Merge design files command on the File menu instead. During the merge process, state-machine syntax is converted to Boolean equations. You then copy string statements and transition equations into the combined PDS file and compile. No errors should occur and this won't force all state machines to be treated as part of the same design.

This concludes the case study of an LSA design implemented using MACH devices.

APPENDIX A

FILE DESCRIPTIONS

CONTENTS

APPENDIX A: FILE DESCRIPTIONS

Files you can print or review at the workstation are stored on the PALASM 4 installation diskette under the following directory. A readme file is included in this directory to identify its organization.

PALASM\EXAMPLES\CB

Each file is described in this appendix, which is divided as follows.

- Included files, A1, discusses the designs covered in this LSA study.
- Optional files, A2, discusses files you can use for a customized LSA implementation.

Note: File descriptions are organized alphabetically.

A.1 INCLUDED FILES

The following files are available on the PALASM 4 installation diskettes and are required for this LSA design.

- Schematics include two data-flow files, I_MEMREG.SCH and I_PPNB.SCH.
- Text files have been merged into one file, LA_MERGE.PDS, which contains the state-machine designs for trigger and trace operations on the control chip.

Each schematic file is hierarchical and includes subschematics that contain the details of a particular aspect of the data flow. The state machine file is also hierarchical. For example, one state machine, LA_KMAIN, controls the operations of all subsidiary state machines defined in individual PDS files. Additional details are provided in the next three discussions.

I_MEMREG.SCH

This memory-register schematic contains the sample metastability registers and a multiplexer for the glitch signals. The following occurs during sample collection.

- Data must be collected from the input lines to determine the state of the input.
- Data must be collected from the preprocessor logic to track glitches occurring during the sample process.

Separate memories are used to track each sample type since the data are collected at the same time. When the trace completes, the multiplexer routes data from the glitch memory to the host-interface bus. The host processor then interleaves the glitch data and the sample data for simultaneous presentation.

I_PPNB.SCH

This preprocessor schematic contains the logic to detect digital events that occur on input-signal lines. The chip processes four bits of input data. Each bit is checked for five conditions.

- Active-high level
- Active-low level
- Rising edge
- Falling edge
- Glitch conditions

Although all patterns are checked, the chip masks reported events to correspond to the particular pattern selected by the user.

- The attributes to be checked enter the chip via the NB_AT lines.

- The bits to be checked enter the chip via the INP lines.

- The masks determining the bits to include in the test enter the chip via the IN lines.

Coincidences are reported on the HIT output lines; the GLIT output lines are used in the control logic for trigger detection.

LA_MERGE.PDS

This text-based file contains all the control logic needed to accomplish the trigger and trace operations discussed earlier. This file contains a full implementation of the flow diagrams presented in discussion 5. There, only a single path and a single subsidiary state machine were discussed. By reviewing this file, you can correlate the discussion to an actual file that implements the flow diagrams presented earlier.

Each LA_Cx.PDS file corresponds to a node on the main flow diagram. The machine, LA_KMAIN.PDS, coordinates the operation of these subsidiary machines by activating them as required to realize a vertical path through the main flow diagram. When a subsidiary machine is active, the main machine normally waits for the active machine to complete its function. The sequencing is controlled by a handshaking protocol implemented in the machines LA_REQ.PDS, request, and LA_RPL.PDS, reply.

- LA_KMAIN starts a process and a request at the same time.
- LA_KMAIN waits for the process to signal completion by starting a reply.
- LA_KMAIN then goes to its next state.

A.2 OPTIONAL FILES

Optional functions are included here to allow you to customize the design to your particular needs. For example, the sample board layout presented in the design description, under discussion 1, shows the profile of a PC add-in card that belongs to a major computer vendor. IBM™ PC clones have a comparable board area. In actually constructing the LSA, you can use either a PC or a stand-alone configuration. The optional functions discussed next serve as guidelines for either path.

I_KB_INT.PDS

If you choose to implement the LSA on a PC add-in card, you may not need a keyboard interface. This file contains the logic to scan a keyboard for characters. This is a 9-bit interface where the ninth bit serves as a Shift indicator. This LSA design uses the Shift to signal the occurrence of attribute settings, such as glitch, active high, etc. In these cases, the logic substitutes a pattern for the coded data appearing on the keyboard input lines: COL_DAT, or column data. The complete attribute mask for a particular data bit is assembled by ORing the results of successive shifted keystrokes.

If the Shift bit is low, the other eight bits are clocked to the INP bus, where the bits are loaded directly to pattern memory as mask data. However, if the keyboard data is encoded, for example, bit address and bit value, the bits are presented to decode logic.

LA_BKCHG.PDS

This file is a version of the LA_COMB.PDS in which actual pin assignments have been back annotated to the PDS file. This provides an example of re-engineering the logic in a chip while maintaining constant pin assignments.

LA_BKPIN.PDS

This file is a version of the LA_COMB.PDS in which actual pin assignments have been back annotated to the PDS file. This provides an example of maintaining the logic in a chip while changing pin assignments.

LA_COMB.PDS

This file contains the control-machine designs discussed in the study in addition to some optional functions required for the add-in card version of the LSA design. The design is divided among two chips, LA_COMB1.PDS and LA_COMB2.PDS, which correspond to Control I and Control II on the sample layout for the add-in card LSA implementation.

LA_LD_GL.PDS

You can implement functions in a personal computer and use the computer memory for data post processing. In this case, you must upload data from the glitch memory to the computer memory. This file contains the state machine to effect that operation. This function, along with other optional functions, such as the keyboard scanning function, can be added to a single additional MACH chip as identified in the figure under discussion 1.

LA_RD_GL.PDS

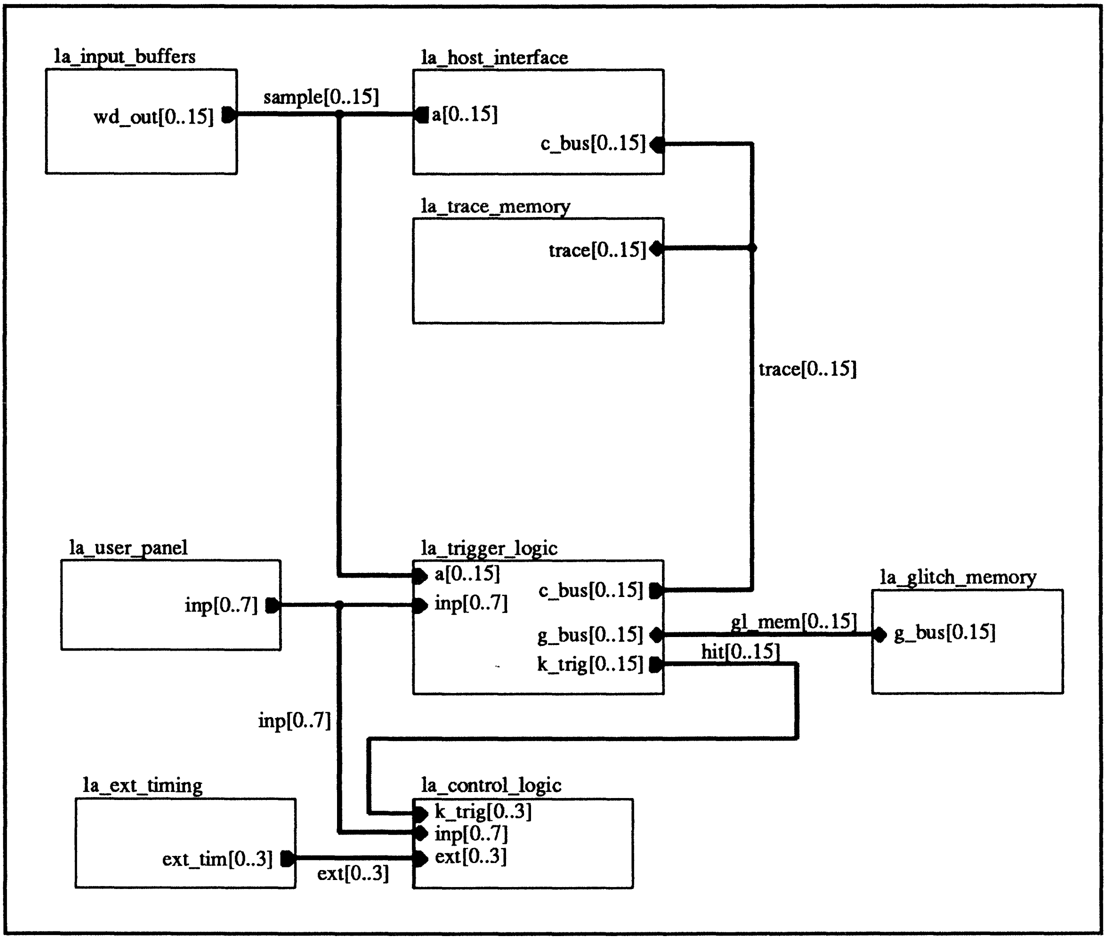
If you implement a glitch memory and want to upload the data through the memory-register chip to the host interface, you can use the LA_RD_GL.PDS file. This file contains the state machine that controls both a static RAM and the memory-register chip to effect an upload operation to the host interface bus: the C_bus. Since host interfaces vary in detail, this file assumes a single-byte transfer.

LA_RLOAD.PDS

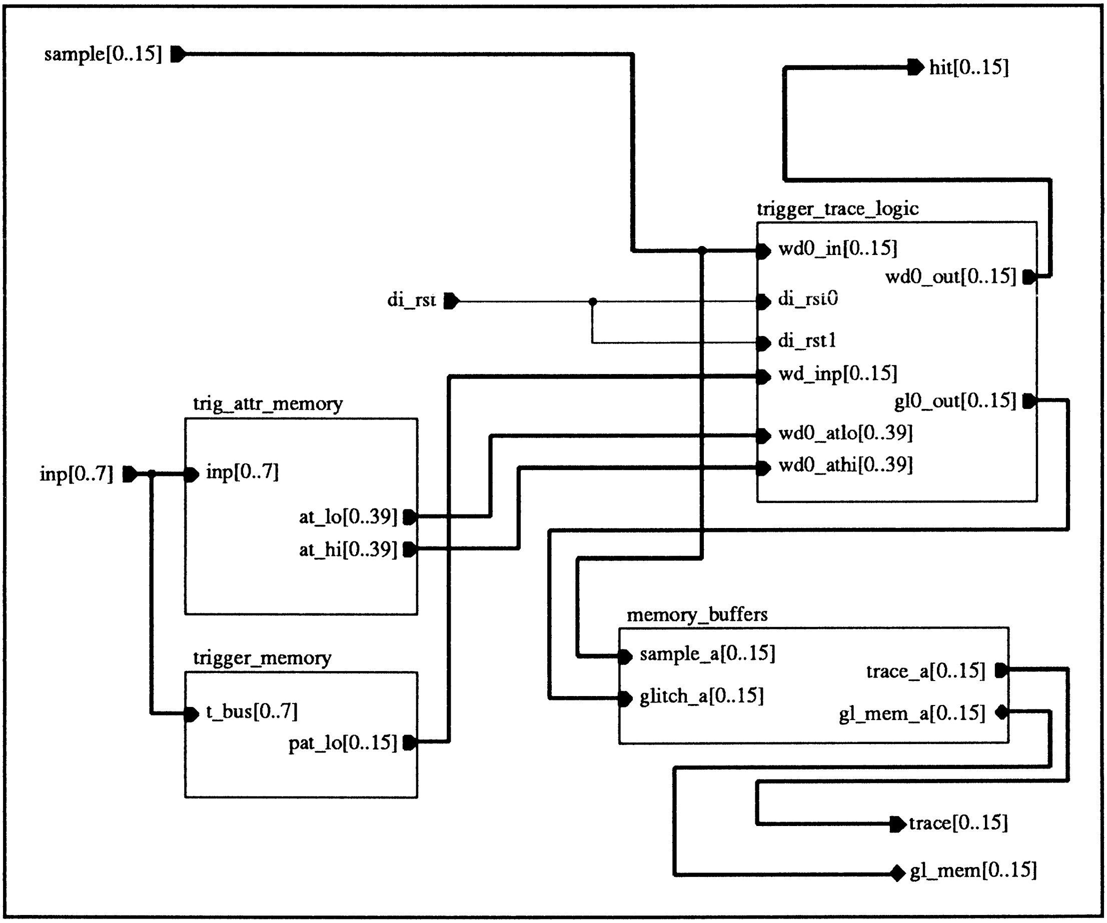
If you choose to implement a stand-alone LSA, you must load the internal registers on the preprocessor chip. The LA_RLOAD.PDS file contains the state machine that loads the data from an external port to the preprocessor chip. The port is assumed to be three-stated to the INP[0..7] bus.

A.3 SCHEMATICS

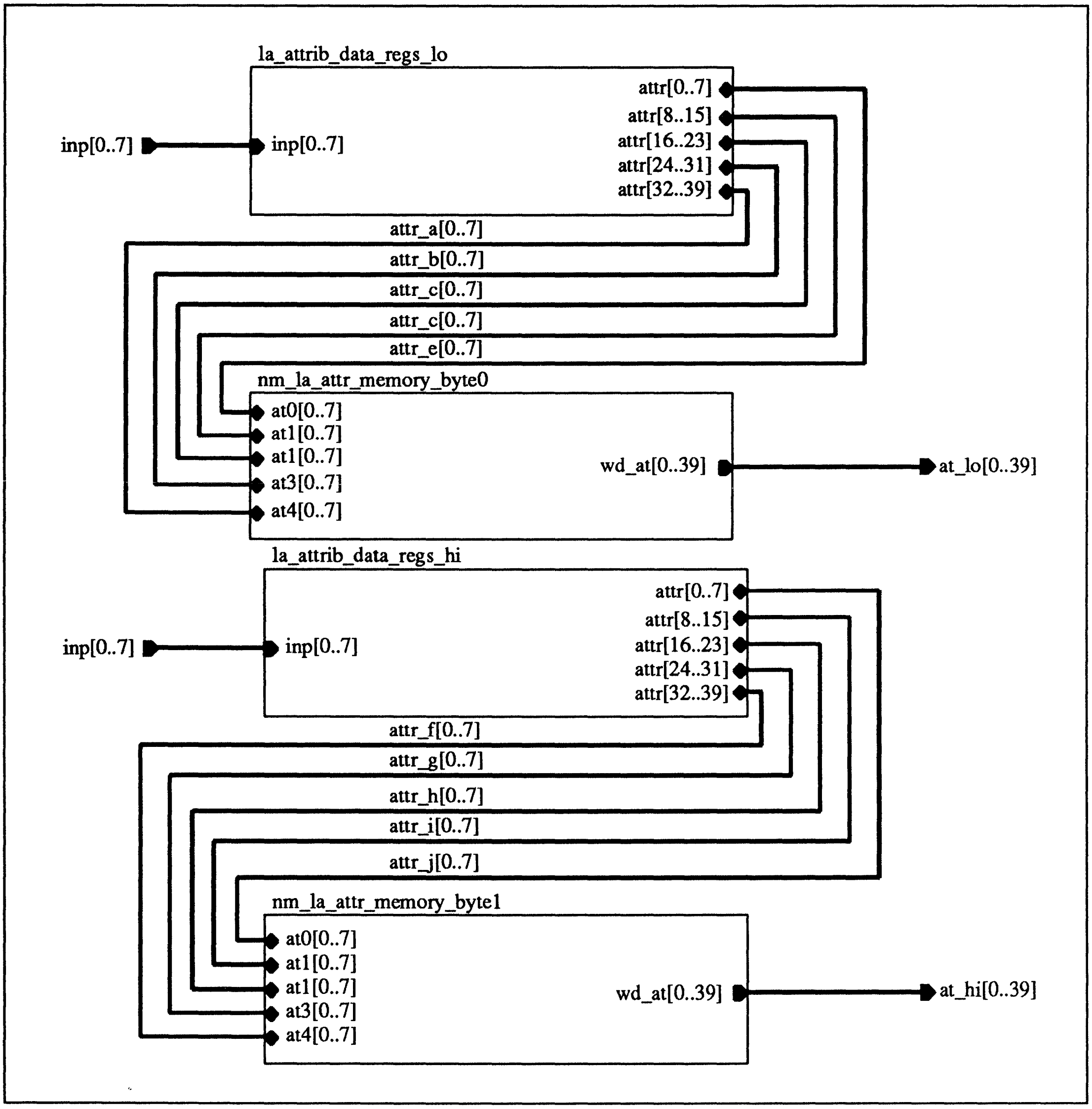
The LSA schematics are included here for your review.



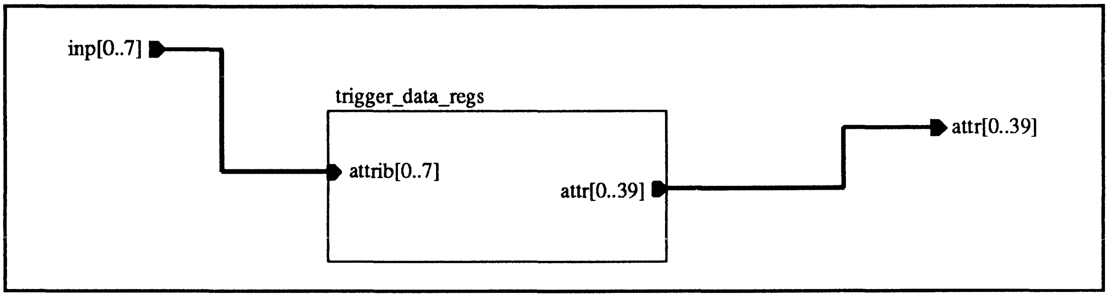
LSA Data Flow



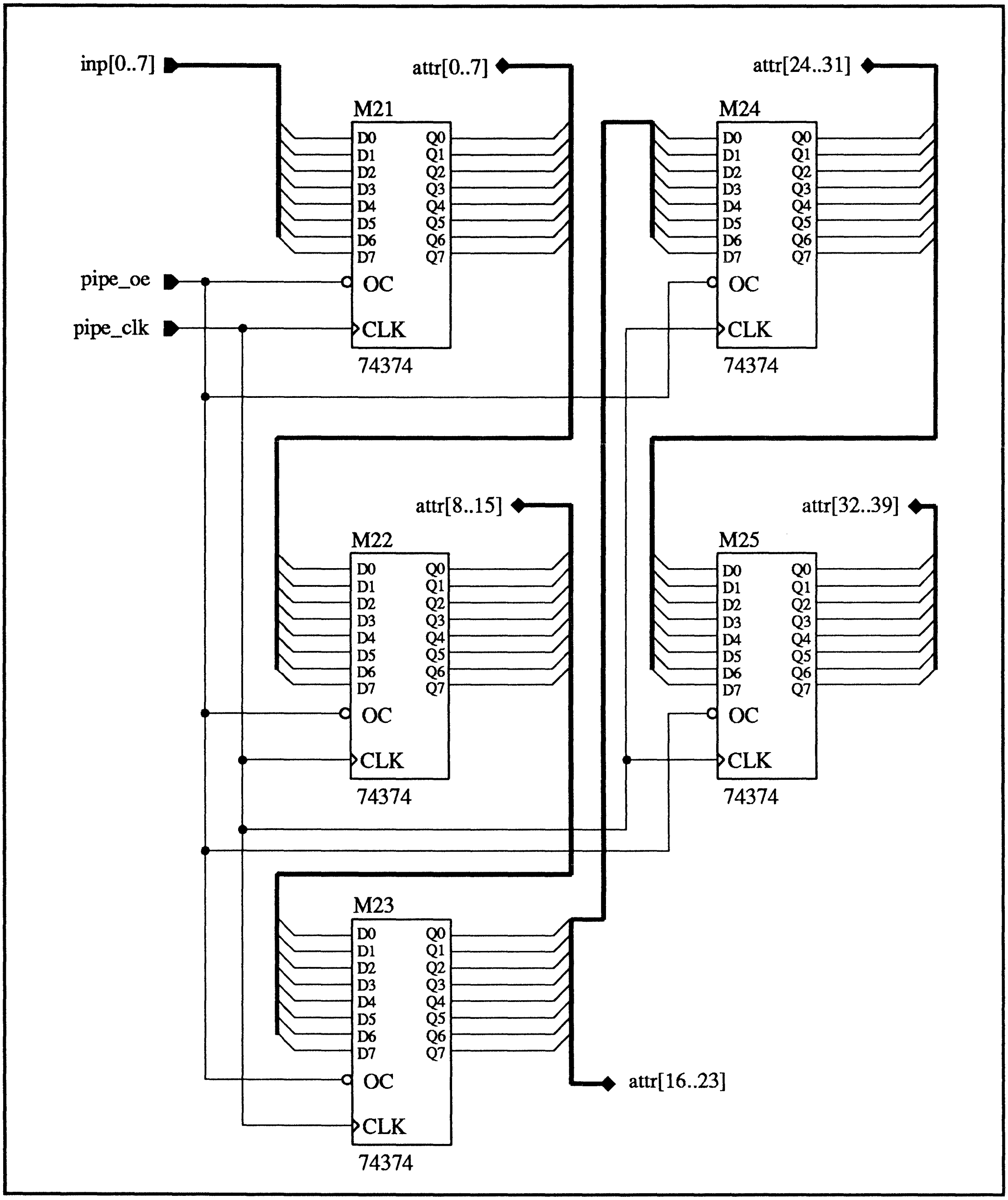
Trigger Logic



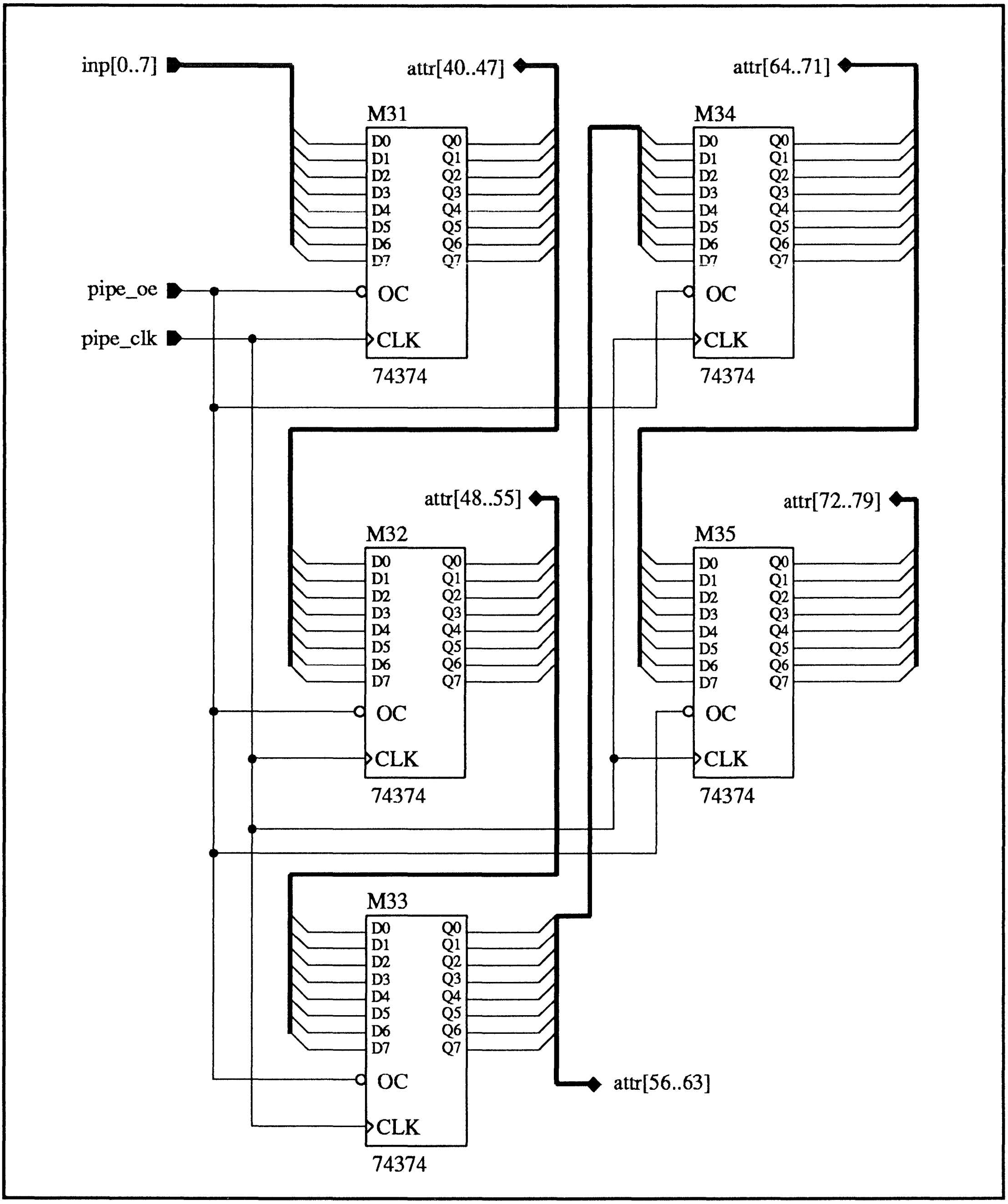
Trigger Mask Memory



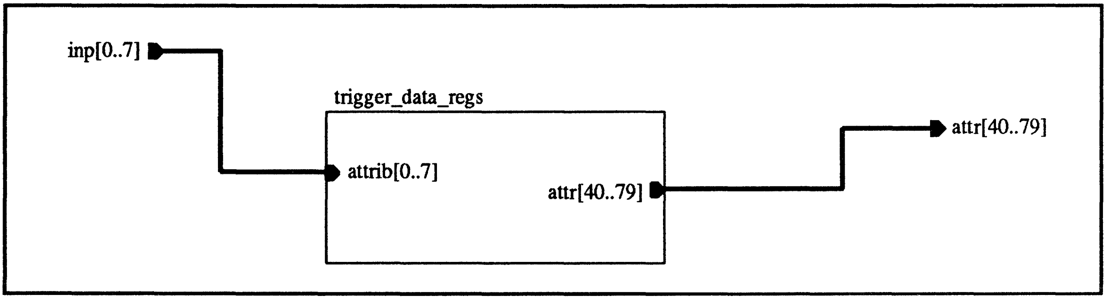
Trigger Data Block, Byte 0



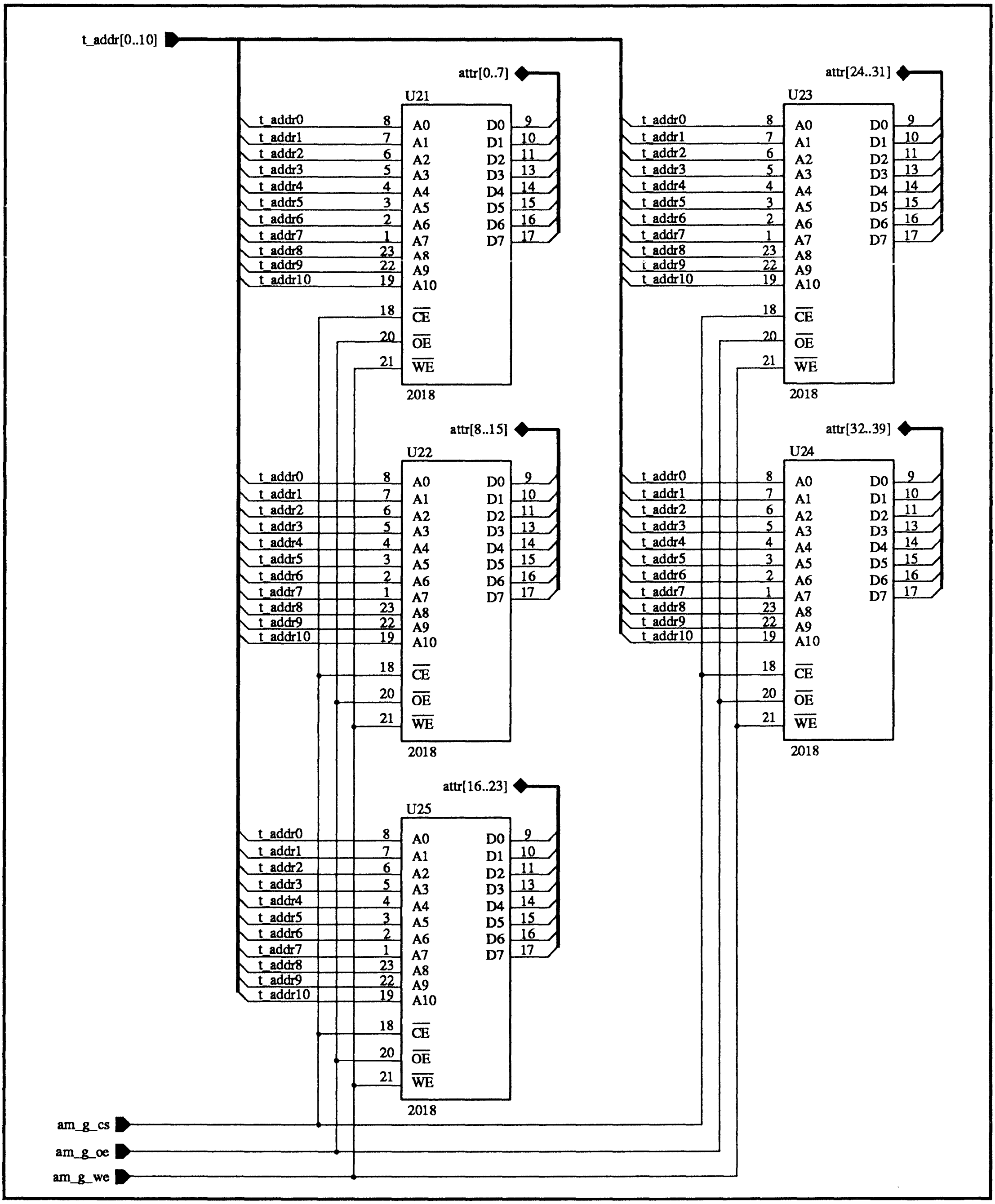
Attribute Memory Data Registers



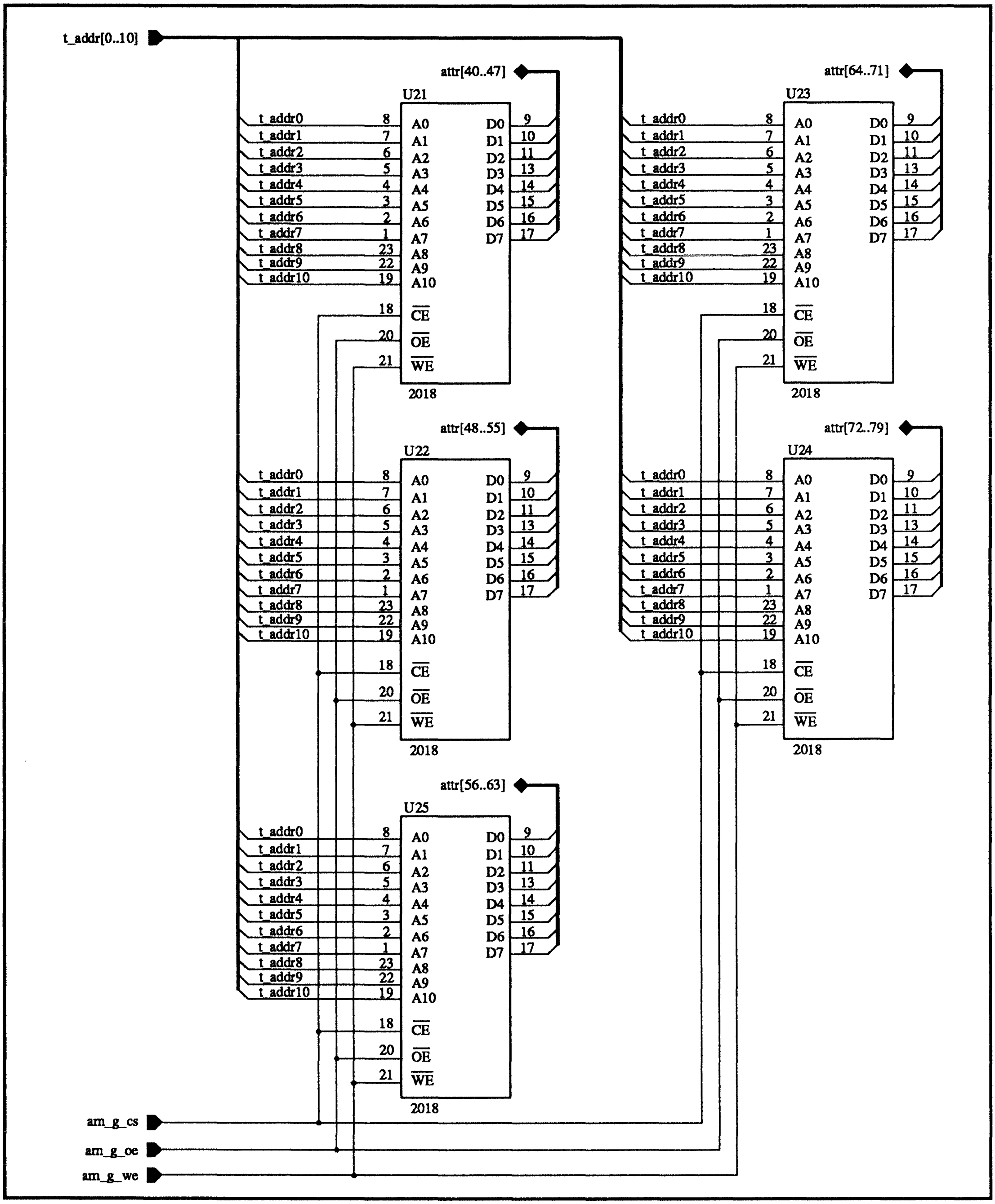
Attribute Memory Data Registers



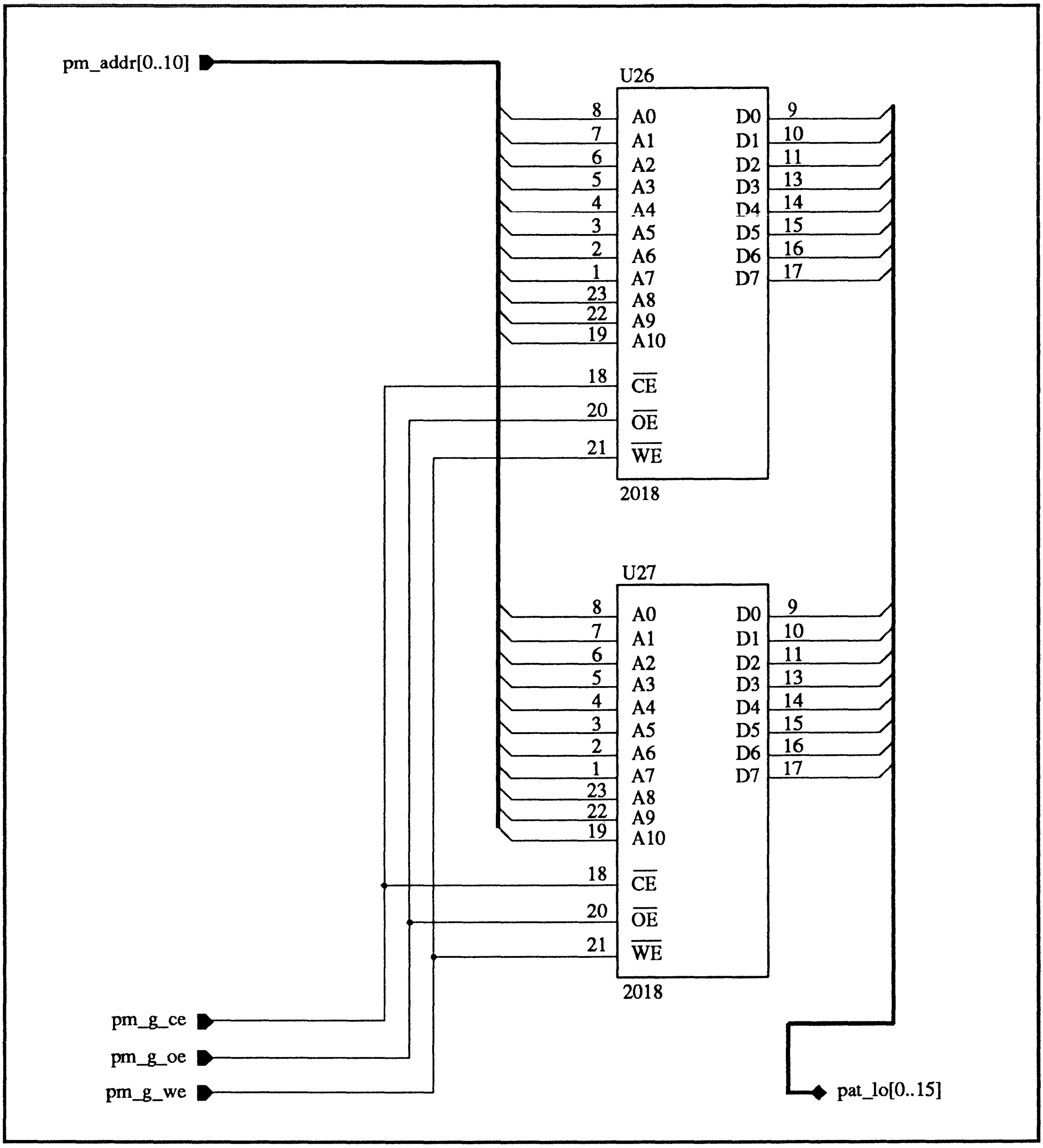
Trigger Data Block, Byte 1



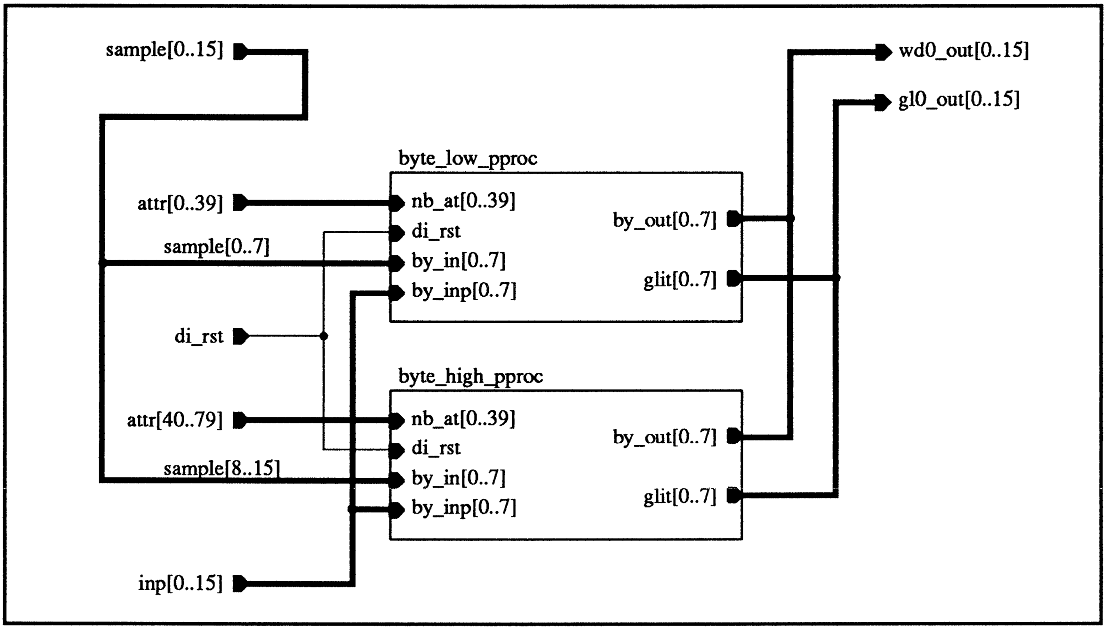
Attribute Memory



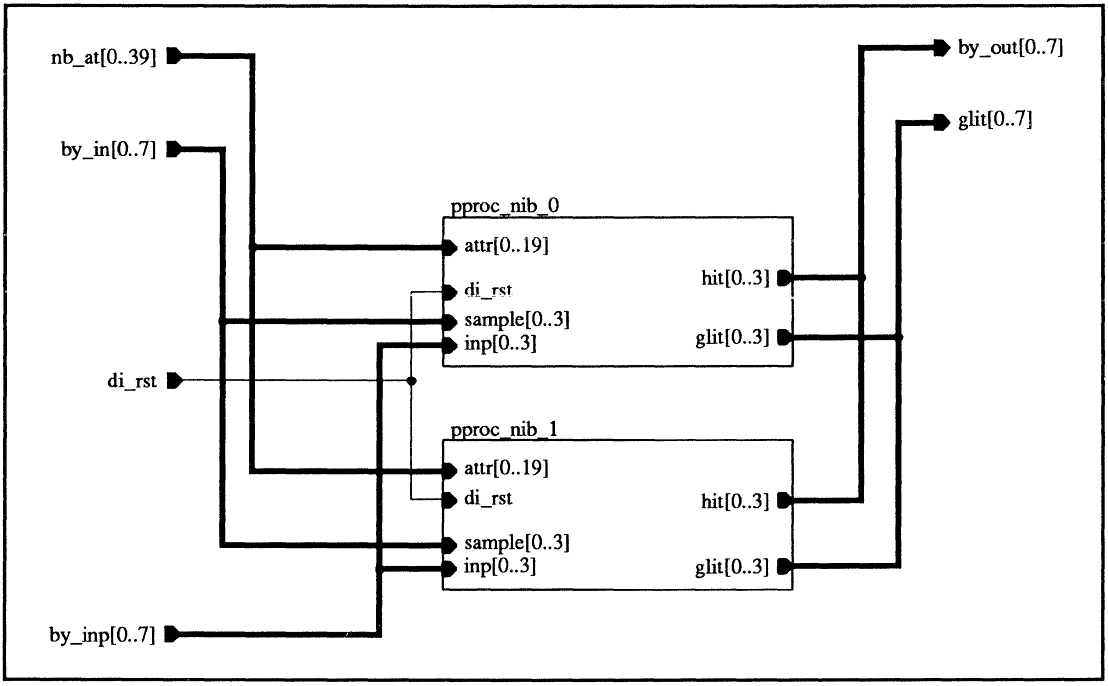
Attribute Memory



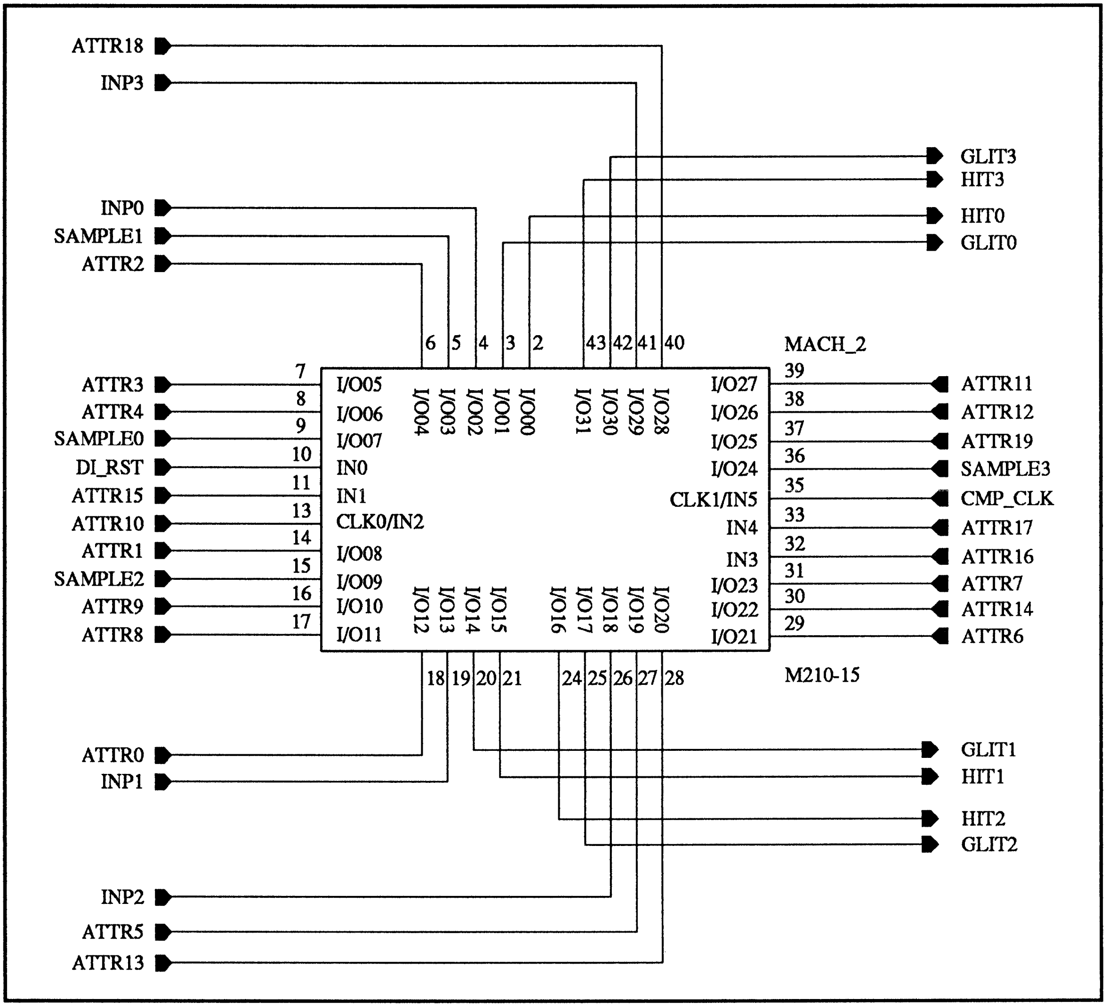
Pattern Memory



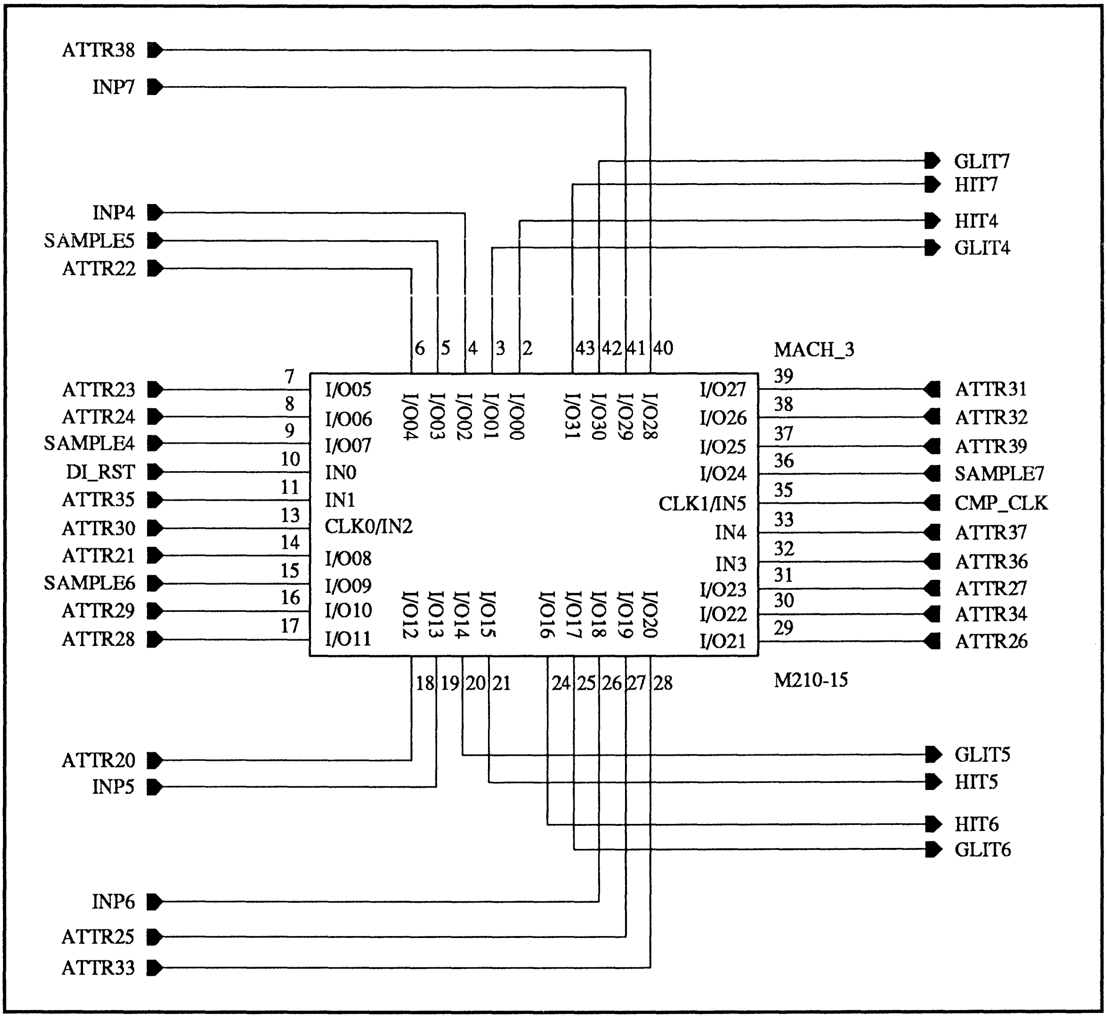
Pre-processor Blocks, Byte 0



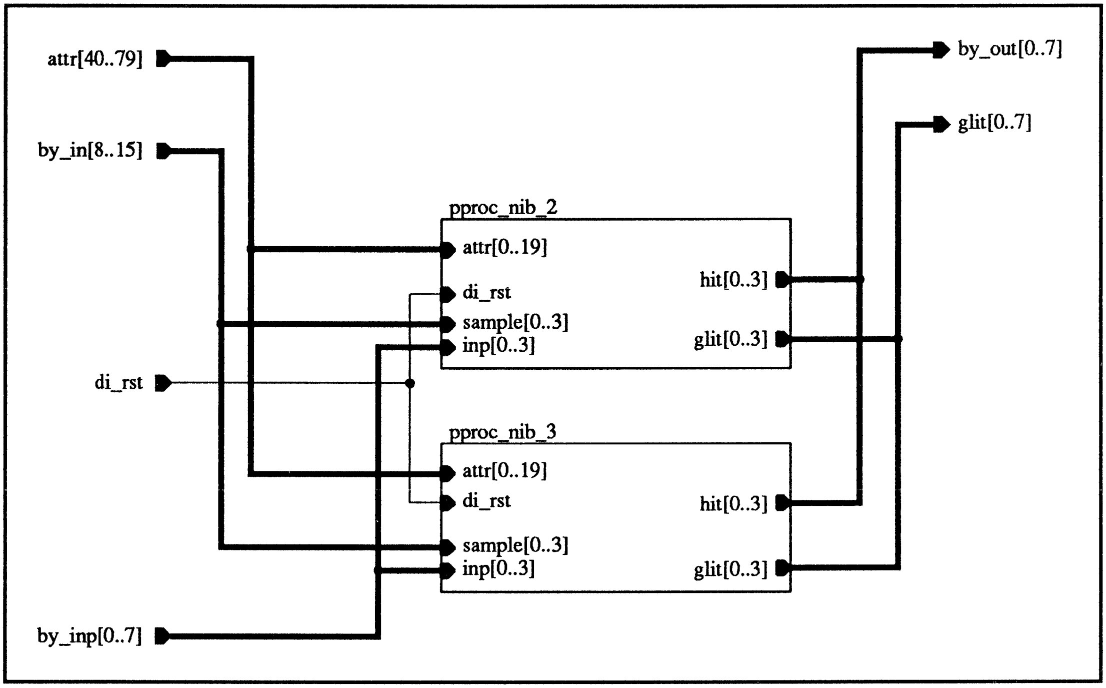
Nibble Blocks, Byte 0



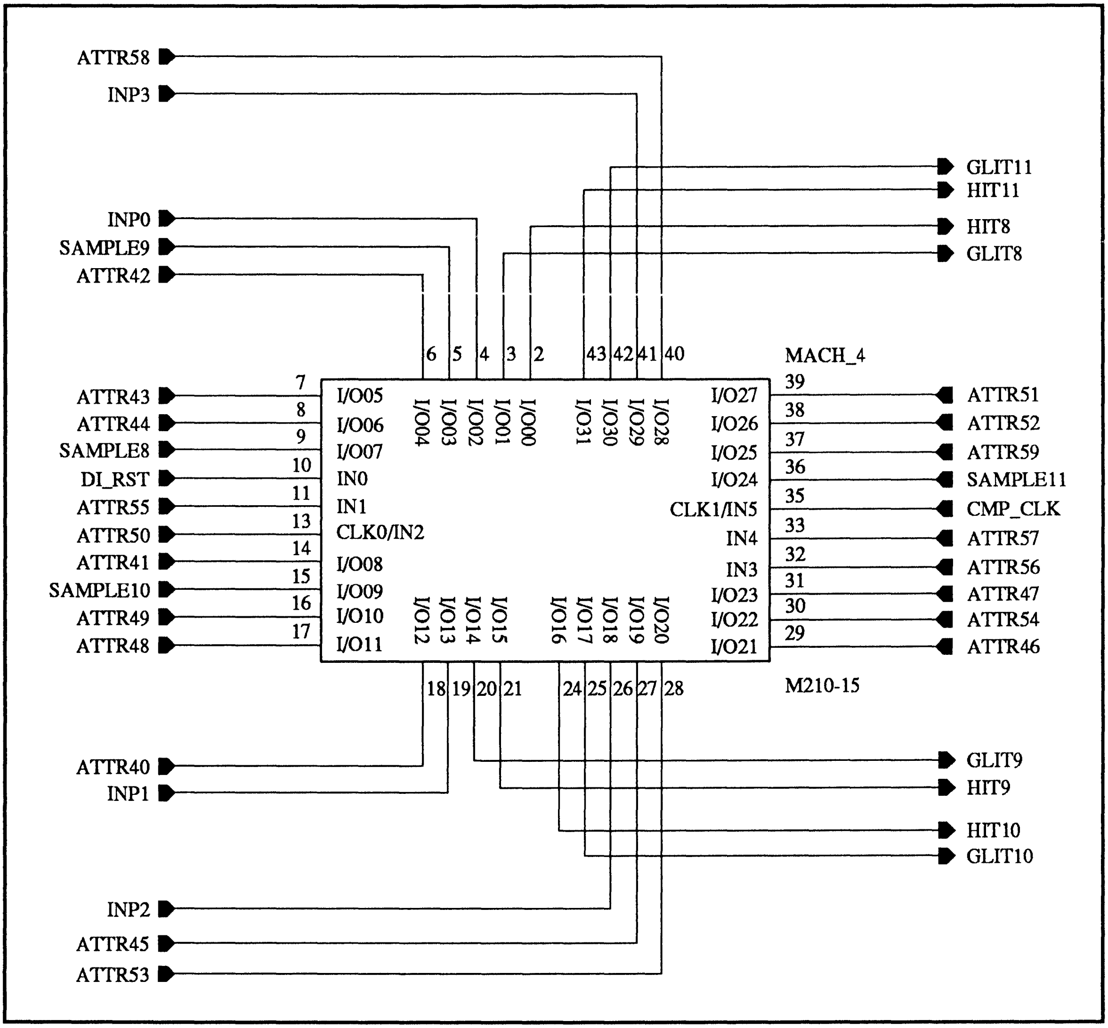
Pre-processor Logic, Nibble 0



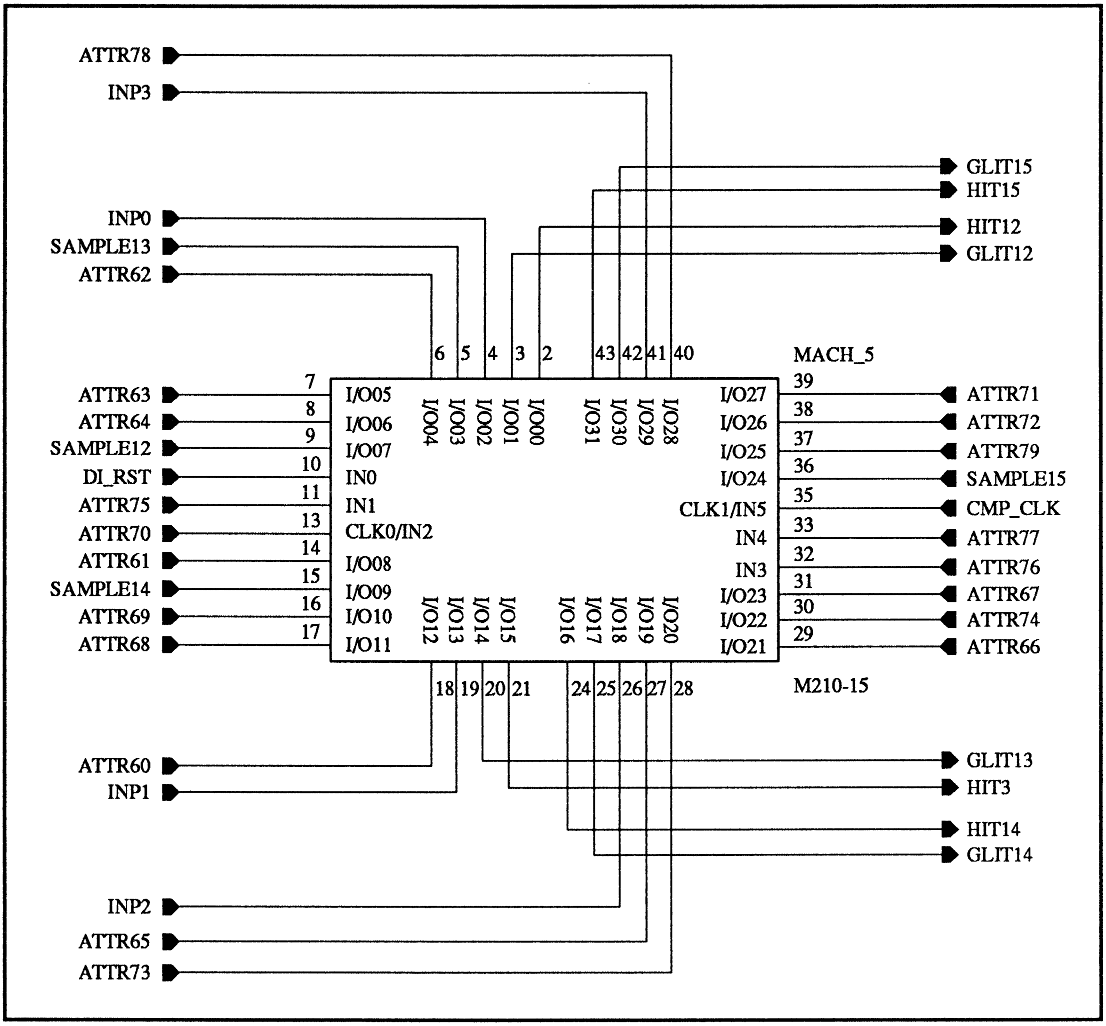
Pre-processor Logic, Nibble 1



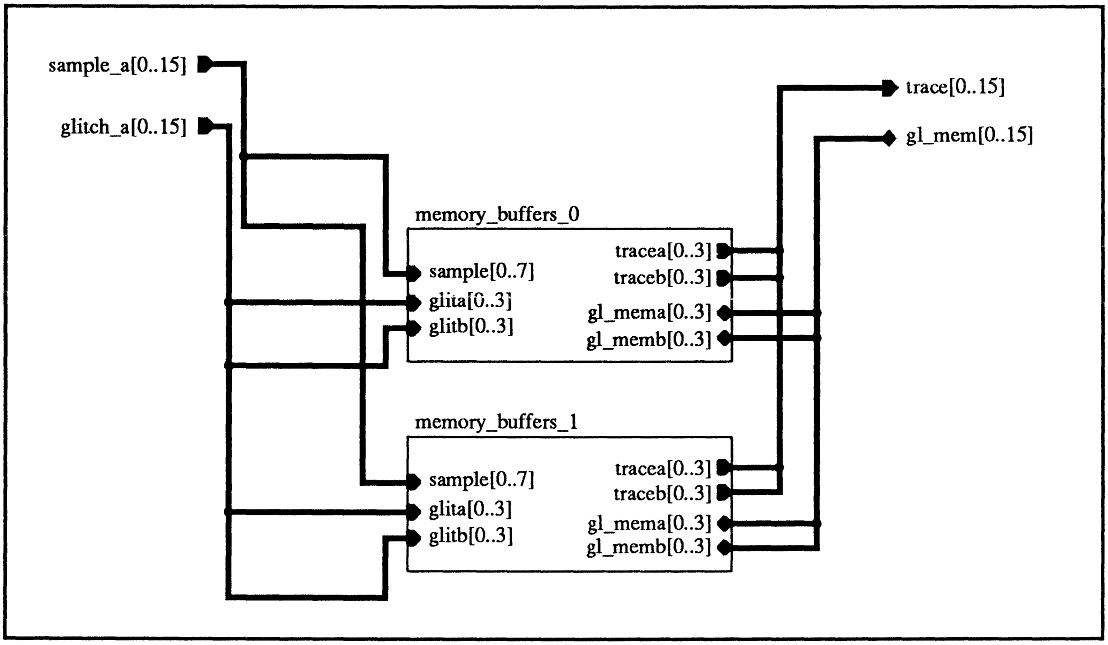
Nibble Blocks, Byte 1



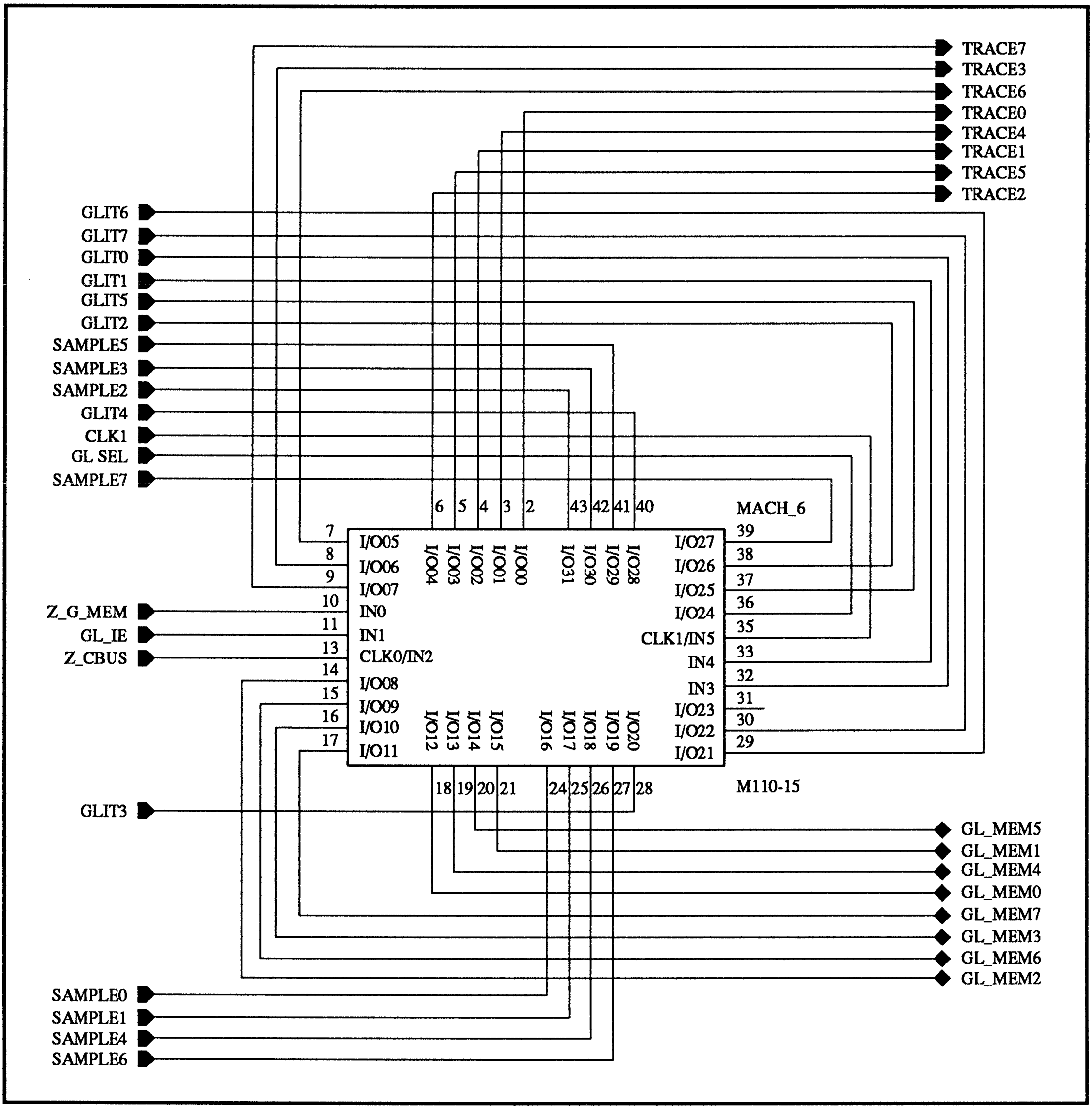
Pre-processor Logic, Nibble 2



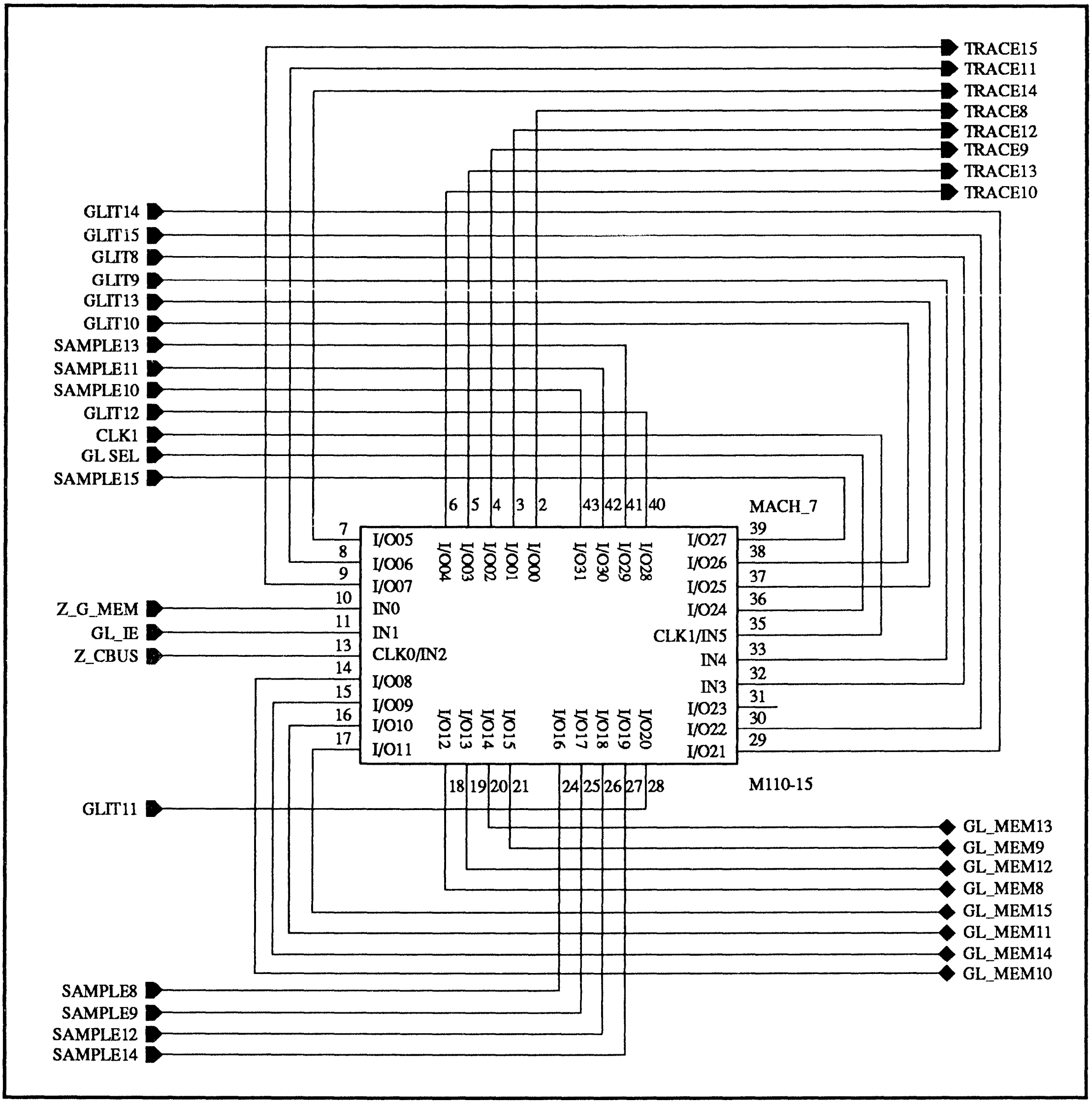
Pre-processor Logic, Nibble 3



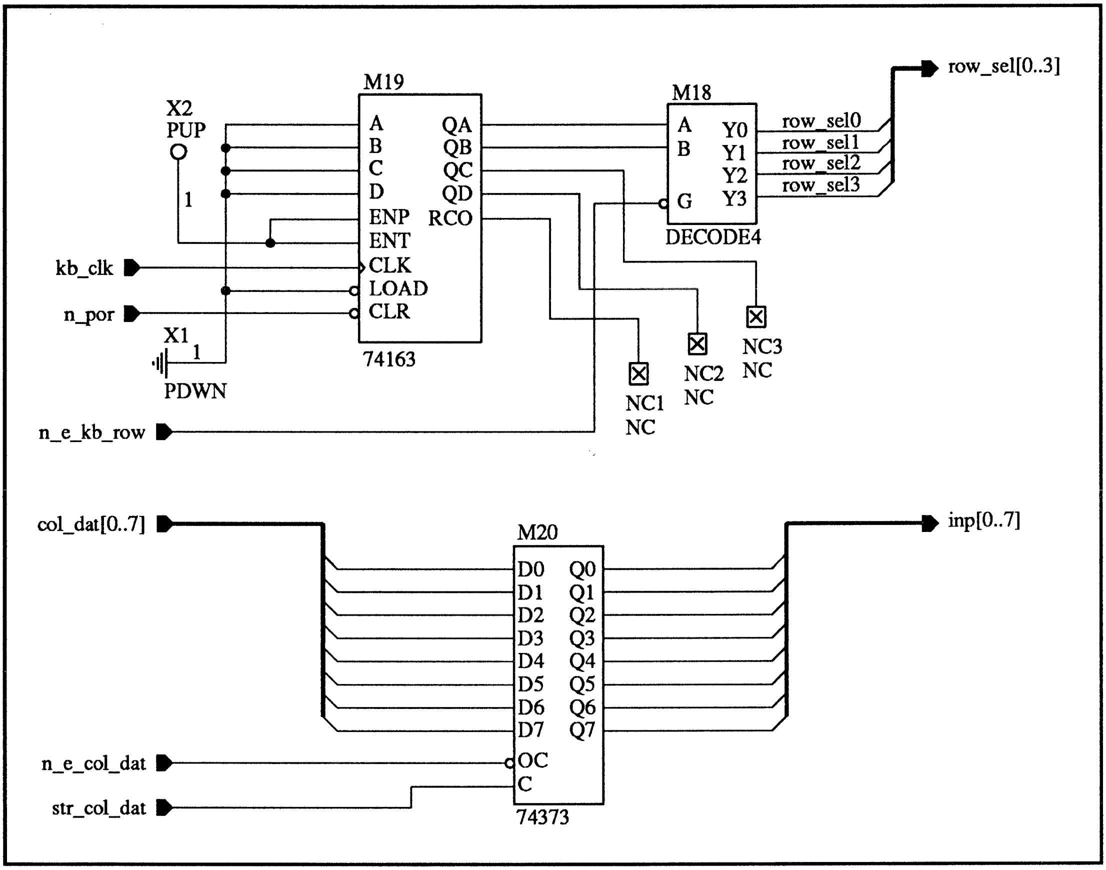
Memory Buffers and Bus Multiplexers



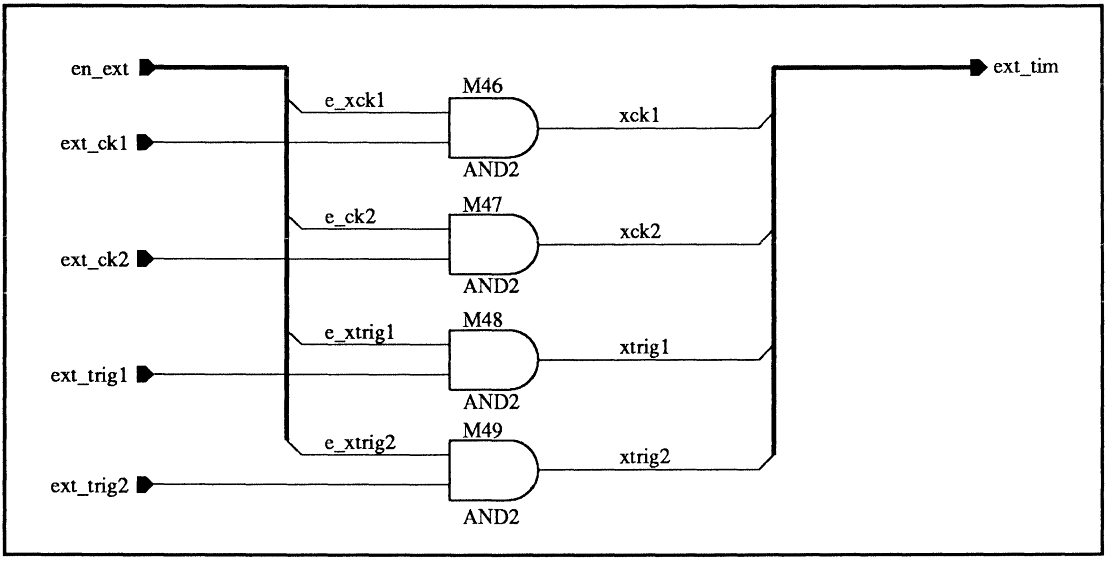
Memory Register Logic, Byte 0 and Bus Multiplexer



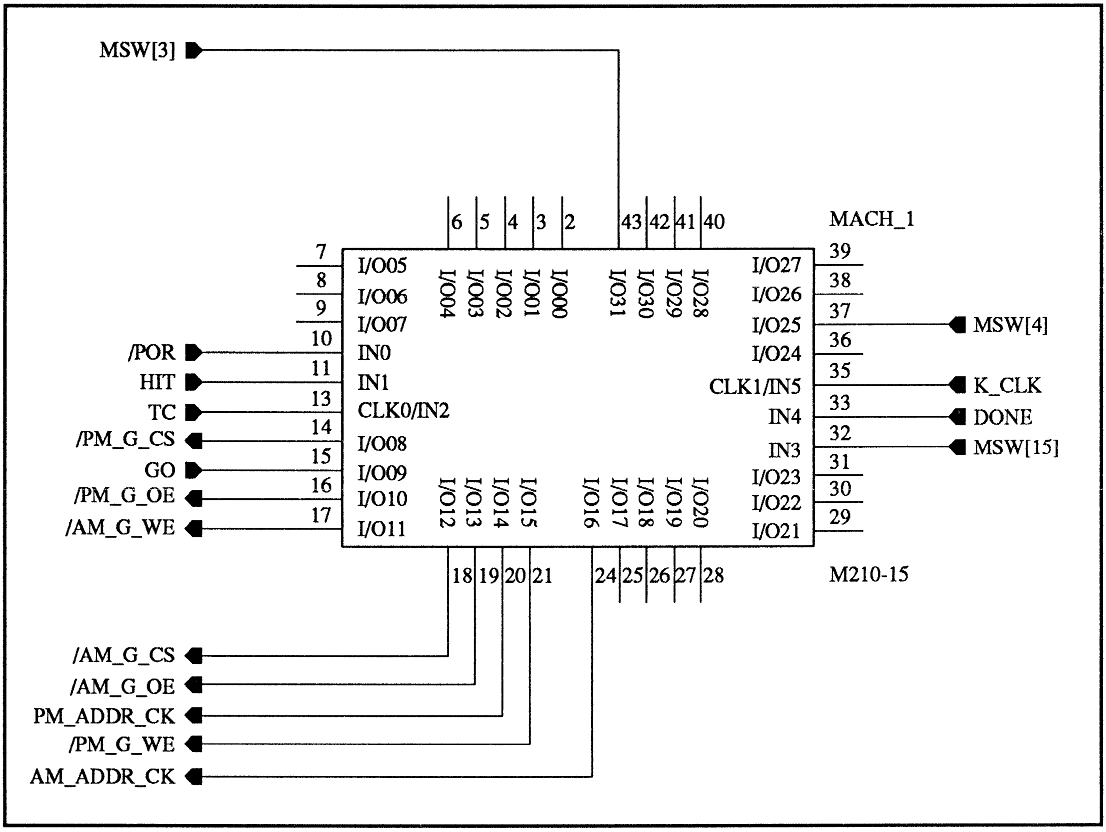
Memory Register Logic, Byte 1 and Bus Multiplexer



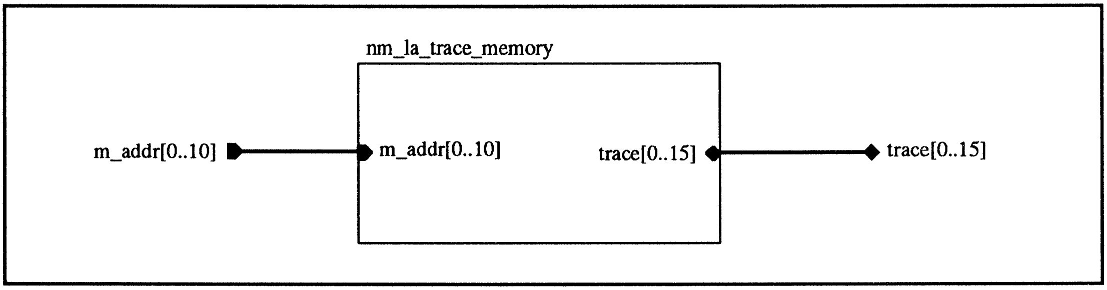
User Panel Keyboard Interface



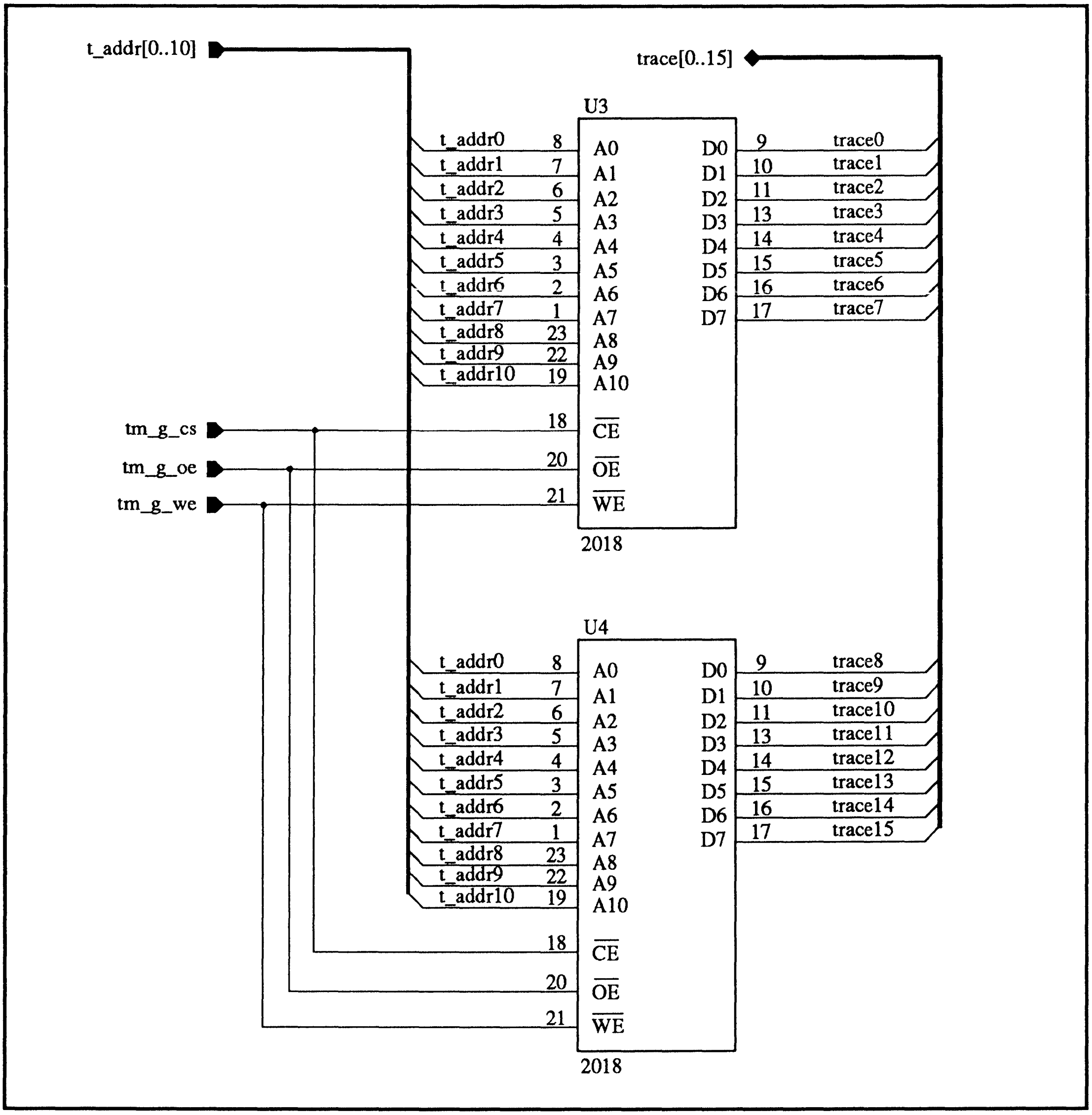
External Timing



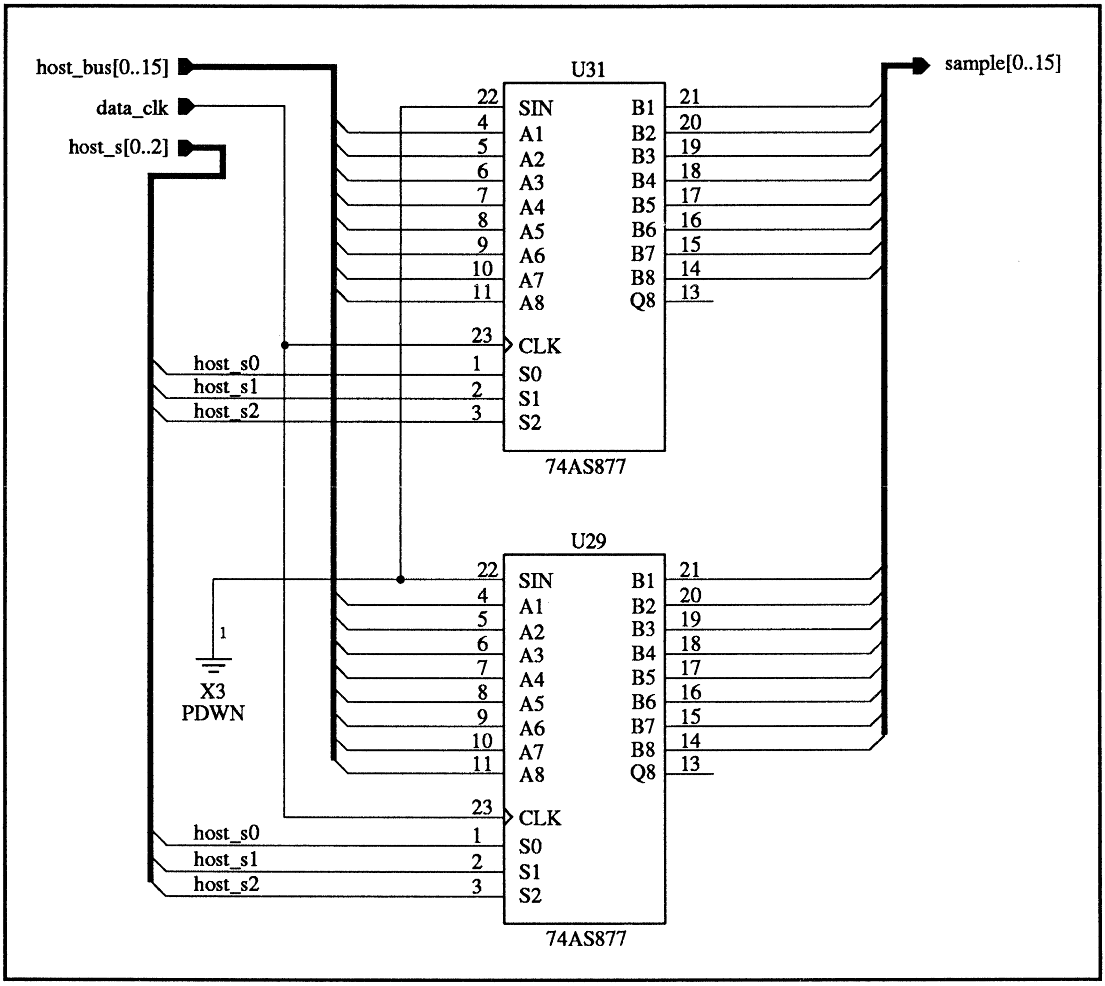
Control Logic



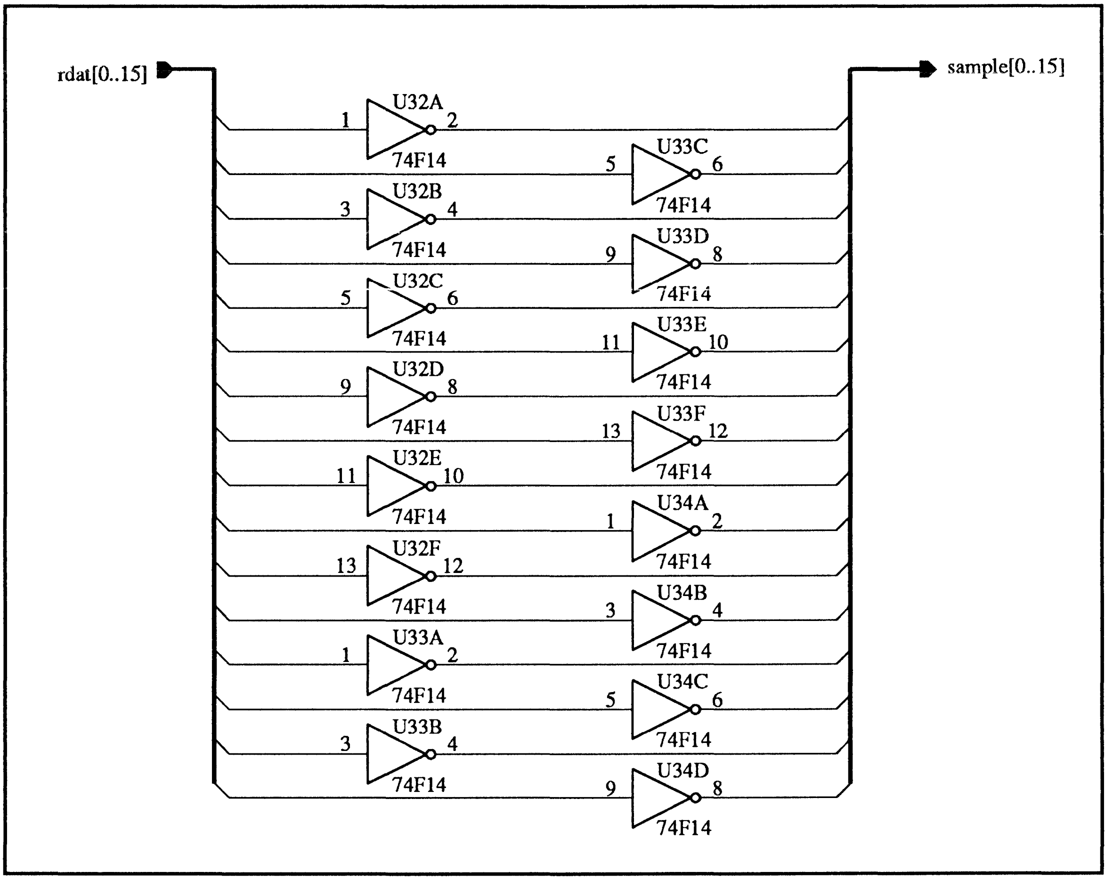
Trace Memory, Word Level


t_addr[0..10]
trace[0..15]
U3
t_addr0 8 A0 D0 9 trace0
t_addr1 7 A1 D1 10 trace1
t_addr2 6 A2 D2 11 trace2
t_addr3 5 A3 D3 13 trace3
t_addr4 4 A4 D4 14 trace4
t_addr5 3 A5 D5 15 trace5
t_addr6 2 A6 D6 16 trace6
t_addr7 1 A7 D7 17 trace7
t_addr8 23 A8
t_addr9 22 A9
t_addr10 19 A10
tm_g_cs 18 CE
tm_g_oe 20 OE
tm_g_we 21 WE
2018
U4
t_addr0 8 A0 D0 9 trace8
t_addr1 7 A1 D1 10 trace9
t_addr2 6 A2 D2 11 trace10
t_addr3 5 A3 D3 13 trace11
t_addr4 4 A4 D4 14 trace12
t_addr5 3 A5 D5 15 trace13
t_addr6 2 A6 D6 16 trace14
t_addr7 1 A7 D7 17 trace15
t_addr8 23 A8
t_addr9 22 A9
t_addr10 19 A10
18 CE
20 OE
21 WE
2018

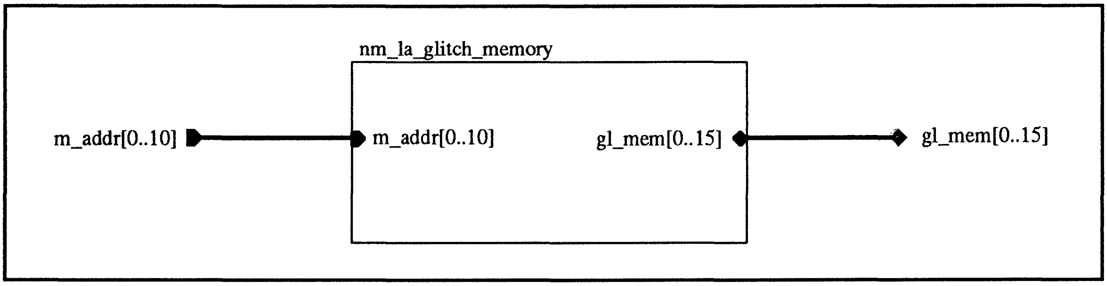
Trace Memory



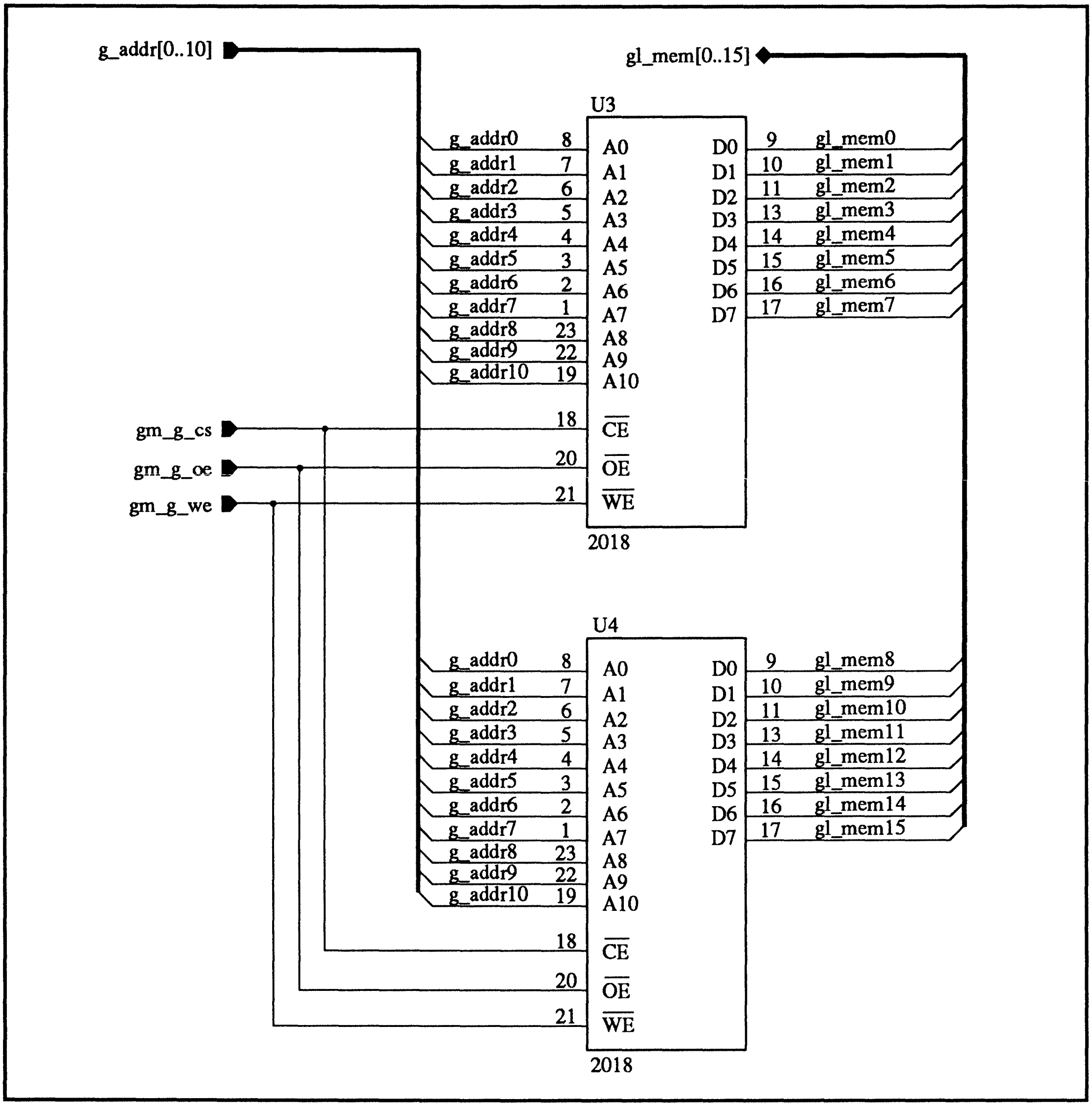
Host Interface



Input Buffers



Glitch Memory, Word Level



Glitch Memory

INDEX

Sales Offices

North American

ALABAMA	(205) 882-9122
ARIZONA	(602) 242-4400
CALIFORNIA,	
Culver City	(213) 645-1524
Newport Beach	(714) 752-6262
Sacramento(Roseville)	(916) 786-6700
San Diego	(619) 560-7030
San Jose	(408) 452-0500
Woodland Hills	(818) 992-4155
CANADA, Ontario,	
Kanata	(613) 592-0060
Willowdale	(416) 224-5193
COLORADO	(303) 741-2900
CONNECTICUT	(203) 264-7800
FLORIDA,	
Clearwater	(813) 530-9971
Ft. Lauderdale	(305) 776-2001
Orlando (Longwood)	(407) 862-9292
GEORGIA	(404) 449-7920
IDAHO	(208) 377-0393
ILLINOIS,	
Chicago (Itasca)	(708) 773-4422
Naperville	(708) 505-9517
KANSAS	(913) 451-3115
MARYLAND	(301) 381-3790
MASSACHUSETTS	(617) 273-3970
MINNESOTA	(612) 938-0001
NEW JERSEY,	
Cherry Hill	(609) 662-2900
Parsippany	(201) 299-0002
NEW YORK,	
Liverpool	(315) 457-5400
Brewster	(914)279-8323
Rochester	(716) 272-9020
NORTH CAROLINA	
Harrisburg	(704) 455-1010
Raleigh	(919) 878-8111
OHIO,	
Columbus (Westerville)	(614) 891-6455
Dayton	(513) 439-0268
OREGON	(503) 245-0080
PENNSYLVANIA	(215) 398-8006
TEXAS,	
Austin	(512) 346-7830
Dallas	(214) 934-9099
Houston	(713) 376-8084
UTAH	(801) 264-2900

International

BELGIUM, Bruxelles	TEL	(02) 771-91-42
	FAX	(02) 762-37-12
	TLX	846-61028
FRANCE, Paris	TEL	(1) 49-75-10-10
	FAX	(1) 49-75-10-13
	TLX	263282F
GERMANY,		
Bad Homburg	TEL	(49) 6172-24061
München	TEL	(089) 4114-0
	FAX	(089) 406490
	TLX	523883
HONG KONG,	TEL	(852) 865-4525
Wanchai	FAX	(852) 865-1147
	TLX	67955AMDAPHX
ITALY, Milan	TEL	(02) 3390541
		(02) 3533241
	FAX	(02) 3498000
	TLX	843-315286
JAPAN,		
Atsugi	TEL	(0462) 29-8460
	FAX	(0462) 29-8458
Kanagawa	TEL	(0462) 47-2911
	FAX	(0462) 47-1729
Tokyo	TEL	(03) 3346-7550
	FAX	(03) 3342-5196
	TLX	J24064AMDTKOJ
Osaka	TEL	(06) 243-3250
	FAX	(06) 243-3253

International *(Continued)*

KOREA, Seoul	TEL	(82) 2-784-7598
	FAX	(82) 2-784-8014
LATIN AMERICA,		
Ft. Lauderdale	TEL	(305) 484-8600
	FAX	(305) 485-9736
	TLX	5109554261 AMDFTL
NORWAY, Hovik	TEL	(03) 010156
	FAX	(02) 591959
	TLX	79079
SINGAPORE	TEL	65-3481188
	FAX	65-3480161
	TLX	55650 AMDMMI
SWEDEN,		
Stockholm	TEL	(08) 733 03 50
(Sundbyberg)	FAX	(08) 733 22 85
	TLX	11602
TAIWAN	TEL	(886) 2-7213393
	FAX	(886) 2-7723422
	TLX	886-2-7122066
UNITED KINGDOM,		
Manchester area	TEL	(0925) 828008
(Warrington)	FAX	(0925) 851219
	TLX	851-628524
London area	TEL	(0483) 740440
(Woking)	FAX	(0483) 756196
	TLX	851-859103

North American Representatives

CANADA	
Burnaby, B.C. - DAVETEK MARKETING	(604) 430-3680
Calgary, Alberta - DAVETEK MARKETING	(403) 291-4984
Kanata, Ontario - VITEL ELECTRONICS	(613) 592-0060
Mississauga, Ontario - VITEL ELECTRONICS	(416) 676-9720
Lachine, Quebec - VITEL ELECTRONICS	(514) 636-5951
ILLINOIS	
HEARTLAND TECH MKTG, INC	(312) 577-9222
INDIANA	
Huntington - ELECTRONIC MARKETING CONSULTANTS, INC	(317) 921-3450
Indianapolis - ELECTRONIC MARKETING CONSULTANTS, INC	(317) 921-3450
IOWA	
LORENZ SALES	(319) 377-4666
KANSAS	
Merriam – LORENZ SALES	(913) 469-1312
Wichita – LORENZ SALES	(316) 721-0500
KENTUCKY	
ELECTRONIC MARKETING CONSULTANTS, INC	(317) 921-3452
MICHIGAN	
Birmingham - MIKE RAICK ASSOCIATES	(313) 644-5040
Holland – COM-TEK SALES, INC	(616) 392-7100
Novi – COM-TEK SALES, INC	(313) 344-1409
MINNESOTA	
Mel Foster Tech. Sales, Inc.	(612) 941-9790
MISSOURI	
LORENZ SALES	(314) 997-4558
NEBRASKA	
LORENZ SALES	(402) 475-4660
NEW MEXICO	
THORSON DESERT STATES	(505) 883-4343
NEW YORK	
East Syracuse – NYCOM, INC	(315) 437-8343
Woodbury – COMPONENT CONSULTANTS, INC	(516) 364-8020
OHIO	
Centerville – DOLFUSS ROOT & CO	(513) 433-6776
Columbus – DOLFUSS ROOT & CO	(614) 885-4844
Strongsville – DOLFUSS ROOT & CO	(216) 899-9370
OREGON	
ELECTRA TECHNICAL SALES, INC	(503) 643-5074
PENNSYLVANIA	
RUSSELL F. CLARK CO.,INC.	(412) 242-9500
PUERTO RICO	
COMP REP ASSOC, INC	(809) 746-6550
WASHINGTON	
ELECTRA TECHNICAL SALES	(206) 821-7442
WISCONSIN	
HEARTLAND TECH MKTG, INC	(414) 792-0920

Advanced Micro Devices reserves the right to make changes in its product without notice in order to improve design or performance characteristics. The performance characteristics listed in this document are guaranteed by specific tests, guard banding, design and other practices common to the industry. For specific testing details, contact your local AMD sales representative. The company assumes no responsibility for the use of any circuits described herein.

Advanced Micro Devices, Inc. 901 Thompson Place, P.O. Box 3453, Sunnyvale, CA 94088, USA
Tel: (408) 732-2400 • TWX: 910-339-9280 • TELEX: 34-6306 • TOLL FREE: (800) 538-8450
APPLICATIONS HOTLINE & LITERATURE ORDERING • TOLL FREE: (800) 222-9323 • (408) 749-5703

3/5/91
Printed in USA

ADVANCED
MICRO
DEVICES, INC.
901 Thompson Place
P.O. Box 3453
Sunnyvale
California 94088-3453
(408) 732-2400
TWX: 910-339-9280
TELEX: 34-6306
TOLL-FREE
(800) 538-8450

APPLICATIONS
HOTLINE & LITERATURE
ORDERING
(800) 222-9323
(408) 749-5703

Printed in USA
CP-28M-4/91-0
15592A