T.O. 33-D7-32-12-11

9100 Series

Applications Manual

P/N 813840
FEBRUARY 1988

FLUKE®

LIMITED WARRANTY

John Fluke Mfg. Co., Inc. (Fluke) warrants your 9100/9105A to be free from defects in material and workmanship under normal use and service for 90 days from the date of shipment. Software and firmware products are provided "AS IS." We do not warrant that software or firmware products will be error free, operated without interruption or that all errors will be corrected. This warranty extends to you if you are the original purchaser and does not apply to fuses, batteries or any product which, in our sole opinion, has been subject to misuse, alteration or abnormal conditions of operation or handling.

To obtain warranty service, contact a Fluke Service Center or send the product, with the description of the difficulty, postage prepaid, to the nearest Fluke Service Center. Fluke assumes no risk for damage in transit.

Fluke will, at our option, repair or replace the defective product free of charge. However, if we determine that the failure was caused by misuse, alteration, or abnormal condition of operation or handling, you will be billed for the repair. The repaired product will be returned to you, transportation prepaid.

THIS WARRANTY IS EXCLUSIVE AND IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR USE. FLUKE WILL NOT BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OR LOSS WHETHER IN CONTRACT, TORT, OR OTHERWISE.

9100 Series Applications Manual

by
Douglas Hazelton

John Fluke Mfg. Co., Inc.
Everett, Washington

Acknowledgements

I would like to express my appreciation to Marshall Scott for the original concept of this manual and for his encouragement and support through the writing process. In addition, I would like to thank Pat Donahoo, Bob Cuckler, Marshall Scott, and Tony Vannelli for the "Considerations for Testing and Troubleshooting" material presented in Section 4. I am also appreciative of the assistance of Frank Tomlinson in developing the TL/1 programs for the Demo/Trainer UUT and to Greg Sheahen and George Hong for their valuable suggestions concerning the user interface graphics. And finally, I would like to thank Ted Thwing for his editorial suggestions and for coordinating the development of this manual.

Contents

Section	Title	Page

(This page is intentionally blank.)

Figures

Figure	Title	Page

Figure	Title	Page

Figure	Title	Page

(This page is intentionally blank.)

Where Am I?

Manual		Description
Getting Started		A description of the parts of the 9100A/9105A, what they do, how to connect them, and how to power up.
Automated Operations Manual		How to run pre-programmed test or troubleshooting procedures.
Technical User's Manual		How to use the 9100A/9105A keypad to test and troubleshoot your Unit Under Test (UUT).
Applications Manual	You Are Here	How to design test or troubleshooting procedures for your Unit Under Test (UUT).
Programmer's Manual		How to use the programming station with the 9100A to create automated test or troubleshooting procedures.
TL/1 Reference Manual		A description of all TL/1 commands arranged in alphabetical order for quick reference.

(This page is intentionally blank.)

Section 1
Introduction

ORGANIZATION OF THIS MANUAL 1.1.

This manual provides an organized approach to testing and troubleshooting a UUT (Unit Under Test) with the 9100A/9105A. The intended reader is someone who will be writing test programs or test procedures for use with the 9100A/9105A.

Additional information on the various parts of the 9100A/9105A system is available in the *Getting Started* booklet. More information about using the operator's keypad for testing and troubleshooting is available in the *Technical Users's Manual.* And more information on programming the 9100A/9105A is available in the *Programmer's Manual* and in the *TL/1 Reference Manual.*

This manual is organized into three major parts:

Sections 1 to 3 give an overview of the capabilities of the 9100A/9105A and the process of developing functional tests and automated troubleshooting procedures.

Section 4 describes some typical functional blocks for a microprocessor-based UUT. For each typical functional block, you will find a summary of things to consider for testing and troubleshooting, a procedure for using the operator's keypad of the 9100A/9105A for functional testing, a 9100A/9105A programmed functional test, and a set of stimulus programs.

NOTE

Each of the functional blocks described in Section 4 are parts of a real UUT, the Fluke Demo/Trainer. It is not necessary that you have the Demo/Trainer UUT to use this manual, but you may wish to purchase the Demo/Trainer from Fluke so you can try out the example procedures and programs.

Sections 5 to 7 show you how to build on the block functional tests to develop functional tests for your whole UUT and how to develop automated troubleshooting procedures using Guided Fault Isolation (GFI).

PREPARING FOR TESTING AND TROUBLESHOOTING 1.2.

The 9100A/9105A is both a testing and a troubleshooting system. As a test station, it determines whether functional blocks of digital circuitry pass or fail. As a troubleshooting station, it determines which nodes or circuit connections are faulty.

The 9100A/9105A has many built-in functions which are useful for functional testing, stimulation of nodes, and measurement of node or component behavior. In addition, the 9100A has a powerful programming language, called TL/1, that is used to customize the capabilities of the 9100A/9105A to match the testing and troubleshooting requirements for your UUT.

The Programmer's Interface option of the 9100A is used to enter UUT information and to create programs that become the building blocks for automated testing and troubleshooting. This interface also provides an automated process for collecting and storing node responses from a known-good UUT. When the 9100A/9105A is used for testing and troubleshooting, measurements on a node are compared with these stored, known-good node responses to determine whether the measured node response is good or bad.

The 9100A is easily programmed. The operator's keypad and display allow you to explore the operation of your UUT by pressing keys on the keypad. Then, as you develop successful test and troubleshooting procedures, you can put these procedures into TL/1 programs to automate the process. Or, if you prefer, you can write the TL/1 programs directly and then check their operation with the debugger built into the 9100A.

The 9100A/9105A is very flexible; it can be used with several different levels of investment in programming. As you increase the level of programming, you increase the degree of automation and the ease of testing and troubleshooting. Five typical levels of programming effort are summarized below and are also shown graphically in Figure 1-1.

- *No programming effort:* Use the keys of the operator's keypad to initiate testing and troubleshooting actions. This level is appropriate for testing or troubleshooting one-of-a-kind UUTs, where investment in programmed testing and troubleshooting is not cost-effective. It is also valuable for keystroke testing and troubleshooting prior to the completion of programmed testing and troubleshooting. Keystroke testing and troubleshooting requires a skilled technician operator.
- *Level 1 Programming:* Create stimulus programs that cause predictable activity at a node and characterize that node activity on a known-good UUT. You may choose to create the node list and the reference designator list at this level also. If you do so, you will be able to backtrace from a bad node to the fault which causes it. You do this by pressing the GFI key on the operator's keypad and specifying the failing node as the starting point.
- *Level 2 Programming:* Create functional tests for each functional block of your UUT. These tests determine whether the functional block passes or fails. Some block

LEVEL OF PROGRAMMING	TESTING AND TROUBLESHOOTING CAPABILITY AT THIS LEVEL
Level 1 • Stimulus Programs for Nodes • Learned Node Responses from Known-Good UUT • Node List and Reference Designator List (Both Optional)	• Can Determine Whether Nodes Are Good or Bad • Can Backtrace from a Bad Node to the Fault (If the Node List and Reference Designator List Are Complete)
Level 2 Functional Tests of Entire Functional Blocks	• Can Use Level 1 Capabilities to Determine Whether Functional Blocks Pass or Fail • Can Use Built-In Functional Tests to Determine Whether Functional Blocks Pass or Fail
Level 3 Go/No-Go Test for the Entire UUT	• Includes Level 1 and Level 2 Capabilities, and • Can Determine Whether the UUT Passes or Fails
Level 4 Go/No-Go Test for the Entire UUT, with Fault Isolation to the Block Level	• Includes Level 1, Level 2, and Level 3 Capabilities, and • Can Isolate the Failing Functional Block and Generate Hints to Start GFI

Figure 1-1: Recommended Programming Sequence

functional tests will use stimulus programs from Level 1, and others will have independent functional test programs.

- *Level 3 Programming:* Create a go/no-go test for the entire UUT, by using all of the necessary functional block tests to create a functional test of the whole UUT. This test determines whether a UUT is good or bad, but does not usually isolate the fault.
- *Level 4 Programming:* Add procedures to the go/no-go test that will isolate the faulty block for any UUTs that fail the go/no-go functional test. This addition to the go/no-go test provides efficient starting points for automated troubleshooting with GFI. If you have not already done so in Level 1, create the node list and the reference designator list. Your program will then be able to backtrace from a bad node to the fault which causes it. Or you can backtrace by pressing the GFI key on the operator's keypad and specifying a failing node as the starting point.

The 9100A/9105A is the center of a expandable system. For example, fixturing can be added to improve functional test throughput in high-volume applications. In addition, the 9100A/9105A can be integrated with manufacturing systems or host computers.

WHERE TO BEGIN 1.3.

The 9100A/9105A system can be operated manually from the operator's keypad in an "immediate" (keystroke) mode, or it can be programmed in TL/1 with functional tests and GFI procedures using the programmer's interface of the 9100A.

A good overview of the full capabilities of the 9100A/9105A will be helpful before you begin using it in either mode. One good way to explore the use of the 9100A/9105A is to adopt the techniques shown in this manual to your own UUT. While reading Section 4, you might try some of the reads, writes, and built-in tests on your own UUT. To try Guided Fault Isolation (GFI), you could treat a small portion of your UUT as if it were the entire system to be tested and diagnosed. Two or three components connected to the microprocessor bus are usually sufficient for such an introductory exploration.

This manual does not assume that you know the TL/1 programming language, although examples of TL/1 programs are included throughout the manual. As you look over these programs and their explanations, you will find many of them quite understandable. However, in some places, you may want to refer to the *Technical User's Manual*, the *Programmer's Manual*, or the *TL/1 Reference Manual* to learn how specific keys or commands work.

Section 2
Overview of Testing and Troubleshooting

"Testing" determines whether a circuit is good (passes) or bad (fails). "Troubleshooting" finds the faulty component or node causing a circuit to fail.

Before microprocessors, a circuit board was tested by applying a sequence of patterns to inputs at the board's edges or at selected nodes within the board's circuitry and then measuring the output. However, for circuit boards that use microprocessors, the most comprehensive coverage is provided by controlling the UUT from the microprocessor bus. One common method of doing this is to plug in a tester at the microprocessor socket.

Testers that control the microprocessor bus must be able to apply stimuli and capture responses at specific times during the cycles. As an example, consider a buffer on a microprocessor data bus: since data is only stable during a small period of the bus cycle, the outputs of the buffer must be measured at the proper time during the bus read/write cycle.

The basic functions of a test system and the basic functions of a troubleshooting system are similar. During either task, the system must emulate bus cycles and measure levels and signal patterns. But the two tasks have different goals. During testing, the goal is to determine whether a UUT is good or bad; it is not necessary to know where the faults are. However, in troubleshooting the goal is to determine what component is bad or what node is bad so that the UUT can be repaired.

Figure 2-1 shows a testing, troubleshooting, and repair cycle. Some users consider testing and troubleshooting to be completely separate tasks. Other users consider them to be almost identical. In situations where volumes of each type of board tested are high, and where many of the boards are likely to be good, the testing and troubleshooting tasks are often separated. But if board volumes are low or if many of the boards tested are faulty, the testing and troubleshooting tasks are often combined into a single process.

The 9100A/9105A can perform testing and troubleshooting as a single task or as separate tasks. In either case, the system's TL/1 programs are very similar because of the modular structure encouraged by the 9100A programming environment. This manual discusses a broad variety of test and troubleshooting techniques; you can then determine how the techniques should be linked and to what degree the entire process should be automated for your application.

EMULATIVE TESTING 2.1.

The 9100A/9105A is an emulative tester and troubleshooter. By taking control of the UUT's microprocessor bus, the 9100A/9105A can perform all operations, apply all stimuli, and capture any responses that the UUT microprocessor could.

The 9100A/9105A is designed for testing microprocessor-based hardware. The emulative testing approach of the 9100A/9105A should not be confused with in-circuit emulators which also plug into the microprocessor socket and are designed to test software. The in-circuit emulators are difficult to use for board testing because they work with assembly language (which is different from one microprocessor to another). They also require the use of breakpoints to allow examination of UUT registers and memory to check out operation of the UUT. In contrast, the TL/1 programming language of the 9100A/9105A has

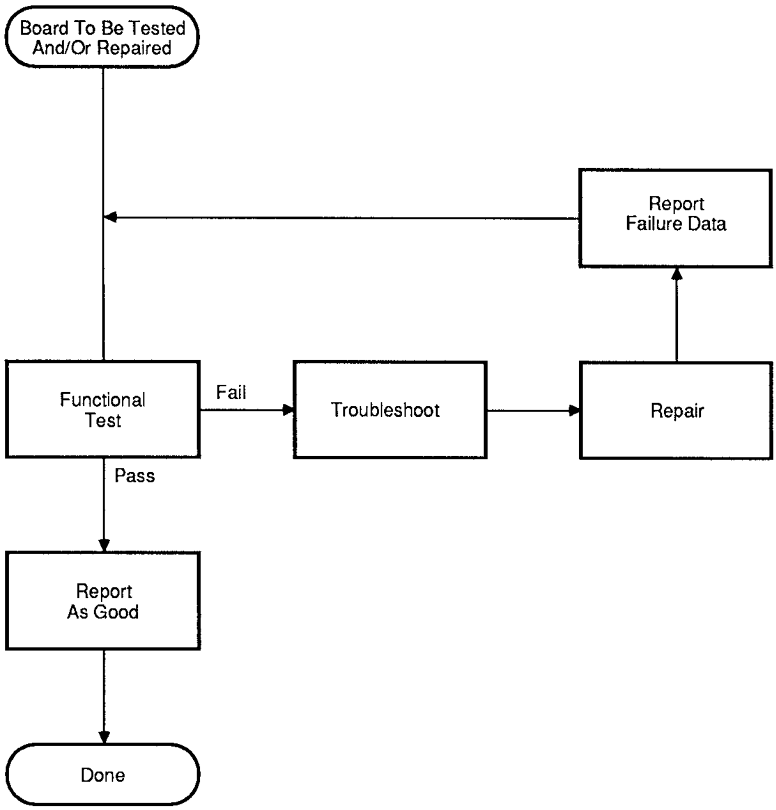


Figure 2-1: Testing, Troubleshooting, and Repair

commands to perform read or write accesses without requiring that you write any assembly language.

The basic elements of the 9100A/9105A system's emulative testing are:

- Stimulation and response sensing at the microprocessor bus by the pod.
- Stimulation of circuitry by the pod, probe, and I/O module.
- Measurement of stimulation responses with the pod, probe, and I/O module as the signals propagate throughout the UUT.
- High-level programming language (independent of the target microprocessor) to control microprocessor accesses and operations.

Figure 2-2 illustrates these capabilities. The method of emulative testing allows the pod to read from and write to any components that the microprocessor can access. The pod can initialize and program components in the UUT, such as DMA controllers, PIAs, serial ports, and video controllers.

In addition to controlling the UUT from the microprocessor bus, the pod senses loaded or faulty lines at the socket where the pod plugs into the UUT. For example, if a data line has a short to ground, the pod will detect that the line cannot be driven when the pod attempts to drive the line high.

The I/O modules and the probe can measure and stimulate all of the UUT's digital circuitry, including circuits not directly accessible by the pod. The pod, I/O module, and probe are used together or individually to provide a stimulus and to capture responses.

The 9100A/9105A can characterize nodes with CRC signatures, level histories (asynchronous or synchronous), transition counts, and frequencies using the single-point probe or 40-line I/O modules. I/O modules accommodate clip modules that fit

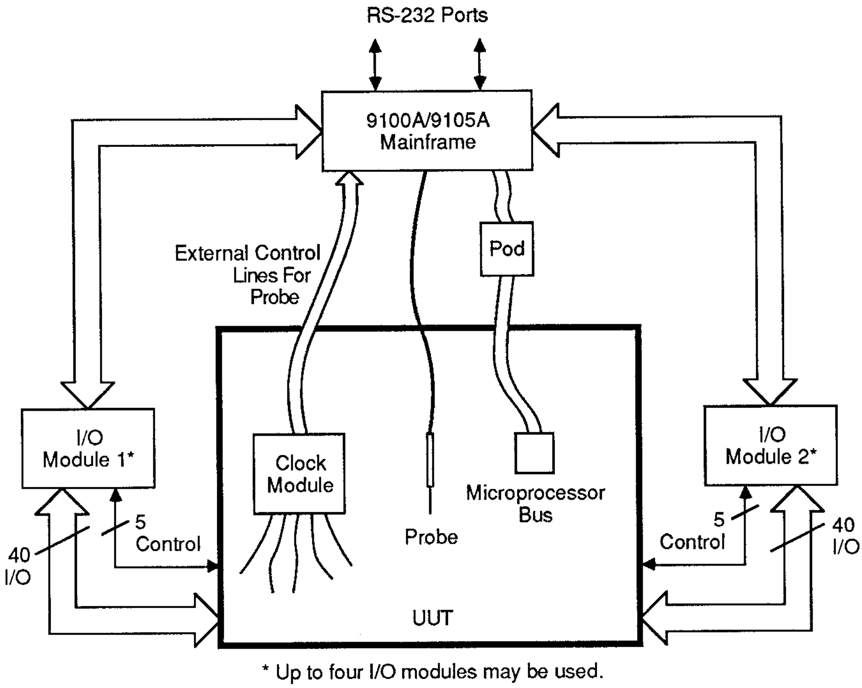


Figure 2-2: Emulative Testing With the 9100A/9105A

various IC packages. The I/O modules can also be used in fixturing.

When the 9100A/9105A stimulates the UUT through the microprocessor bus, an I/O module or the probe can measure the signals as they propagate through the UUT. Or, the I/O modules can stimulate nodes and the pod can measure the activity from the microprocessor bus.

A powerful feature of the 9100A/9105A is that it can perform measurements which are synchronized to microprocessor operations. For example, consider the microprocessor bus. It is a flurry of activity when examined with an oscilloscope, but the 9100A/9105A can control this activity and can examine the signals on the data bus at times when the signals are valid.

The probe and I/O modules can be synchronized to data, address, and other pod cycles, as well as to external Clock, Start, Stop and Enable inputs provided on the 9100A/9105A's I/O module and clock module. The external sync modes are valuable for measuring events asynchronous to the microprocessor, such as video signals and free-running counters.

NODE CHARACTERIZATION 2.2.

Node characterization is the process of finding a description of the correct activity at a node, given an appropriate stimulus to the UUT to exercise the node. A quality characterization is one that is repeatable from one measurement to another, from one UUT to another, and from one day to another. In addition, incorrect activity at the node should result in a value that is different from the characterization for correct node activity. The 9100A/9105A uses the probe or the I/O module to measure five node characteristics:

- *CRC signature:* This measures high and low levels relative to a series of events (called "clock" or "sync") and then encodes a Cyclic Redundancy Check (CRC) number representing both level and timing. The signature, if stable, is the most accurate characterization of a node. If the node changes states at or near the clock transition, the signature is considered marginal because a slight relative time change between clock and data will change the signature.

- *Asynchronous level history:* This indicates whether the node was ever at a high, low, or invalid level at any time during a specified period.
- *Clocked (synchronous) level history:* This indicates whether the node was ever at a high, low; or invalid level at any clock or sync edge during a specified period.
- *Transition count:* This measures how many times the node goes low-to-high during the measurement period. When a given node is measured, a single count value is returned. Learned responses stored in a response file, however, may appear as a range of counts. If a range of counts is specified, the measurement will be considered good if it is within the specified range.
- *Frequency:* This measurement is done during a set time interval and is unrelated to clock or sync modes. As with transition counts, learned responses stored in a response file may appear as a range of frequencies.

STIMULUS AND MEASUREMENT CAPABILITIES 2.3.

Figure 2-3 is an overview of the stimulus and measurement capabilities of the 9100A/9105A. The devices used for this include the:

- Pod.
- Probe (with clock module).
- I/O module.

The following sections describe the capabilities of each of these devices.

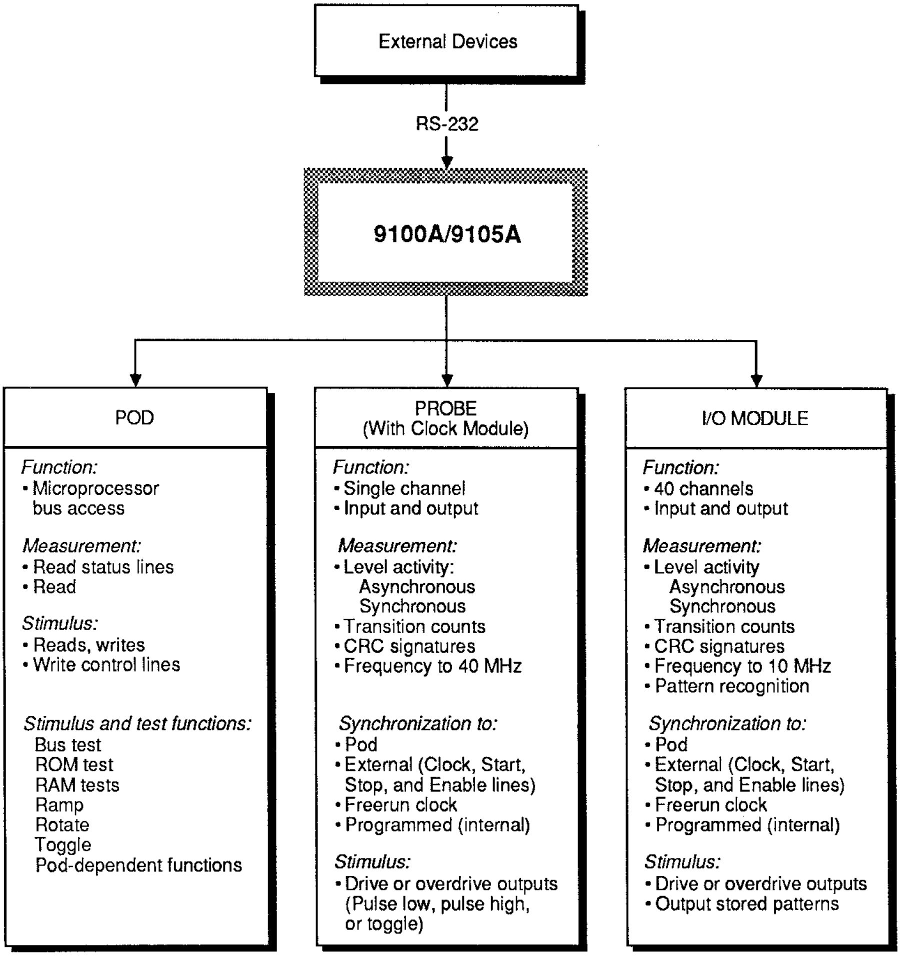


Figure 2-3: 9100A/9105A Stimulus and Measurement Capability

Pod Capabilities 2.3.1.

The Fluke interface pods provide the interface between the 9100A/9105A and the microprocessor bus of a UUT. The pod has two modes of operation: normal mode (where the microprocessor in the pod exercises the UUT microprocessor bus while monitoring the activity on this bus) and RUN UUT mode (where the microprocessor in the pod runs programs stored in UUT memory). A wide variety of stimulus and measurement commands are available either from the operator's keypad or from programs written for automated implementation.

Additional information about pods, their use, and their specifications is contained in section 2.4 of the *Technical User's Manual,* the pod manual for the pod you are using, the *Supplemental Pod Information for 9100A/9105A Users Manual,* and section 3.5 of the *Programmer's Manual.*

Probe Capabilities (With The Clock Module) 2.3.2.

The probe can provide either measurement or output at any selected node of a UUT.

The probe can measure CRC signatures, asynchronous level histories, clocked (synchronous) level histories, transition counts, and frequencies. It has built-in lights to show the current asynchronous level (or levels) at the probe tip or to show the level (or levels) last seen by the synchronous level history latches. The probe can be set up to use one of three different sets of logic thresholds for its measurements: TTL, CMOS, or RS-232.

The probe can also be used as an output device to output a series of pulses. The pulses can be high, low, or can toggle between high and low. The probe has sufficient drive capability (200mA for less than 10μsec or 5mA continuously) to overdrive most circuit nodes.

The probe is synchronized to other events by four synchronization modes: freerun clock, pod data or address sync, external sync (using the external control lines of the Clock Module), and internal sync (for use under program control only). The external control lines of the Clock Module use TTL-level thresholds.

Additional information about the probe, its use, and its specifications is contained in section 2.5 of the *Technical User's Manual,* Appendix D of the *Technical User's Manual,* and section 3.6 of the *Programmer's Manual.*

I/O Module Capabilities 2.3.3.

Each I/O module can make simultaneous connection with up to 40 UUT nodes. I/O module adapters provide an interface between the general-purpose connectors on the I/O module and components on a UUT. The smaller clip modules can be plugged into either side A or side B of the I/O modules, and the larger clip modules use both connectors.

An I/O module can measure CRC signatures, asynchronous level histories, clocked (synchronous) level histories, transition counts, and frequencies. Unlike the probe, an I/O module can measure multiple pins at the same time. An I/O module can be set up to use one of two different sets of logic thresholds for its measurements: TTL and CMOS.

In addition, I/O modules can recognize words that exist across selected UUT nodes. Recognition of specified words generates a Data Compare Equal (DCE) condition, sends a signal out the DCE pin at the side of the I/O module, and terminates any RUN UUT in progress.

I/O module outputs can be latched high or low, pulsed high or low, or allowed to float (high-impedence). In addition, it can use TL/1 commands to drive patterns out of each output. Responses can be measured at any pin while the I/O module is driving a pattern. An I/O module has sufficient drive capability (2A for less than 10μsec or 200mA continuously) to overdrive most circuit nodes.

An I/O module is synchronized to other events by four synchronization modes: freerun clock, pod data or address sync, external sync (using the external control lines located on the I/O module itself), and internal sync (for use under program control only). The external control lines use TTL-level thresholds.

Additional information about the I/O modules, their use, and their specifications is contained in section 2.5 of the *Technical User's Manual,* Appendix D of the *Technical User's Manual,* and section 3.6 of the *Programmer's Manual.*

TESTING AND TROUBLESHOOTING WITH THE 9100A/9105A — 2.4.

The 9100A/9105A can be used for:

- Functional testing.
- Troubleshooting.
- Combined testing and troubleshooting.

As a functional tester, the 9100A/9105A can determine whether a UUT passes or fails a series of tests. As a troubleshooter, the the system can first isolate the failing functional block and then identify a starting location from which detailed fault isolation can locate the node or component causing the failure.

When testing and troubleshooting are performed at the same test station, the 9100A/9105A performs the following sequence of operations:

1. Perform a go/no-go (pass/fail) test of the UUT.
2. Diagnose a failing UUT to determine which functional block is failing.
3. Identify a starting point for fault isolation.
4. Locate the the node or component causing the failure.

If testing and troubleshooting are performed at separate stations, the 9100A/9105A would perform Step 1 at the testing station and Steps 2 through 4 at the troubleshooting station.

In situations where Step 1 is performed by another type of tester, which identifies suspect functional blocks to the 9100A/9105A, the 9100A/9105A can verify that the problem is really in the indicated block before detailed fault isolation is begun. Occasionally, the real problem is in a different functional block than that indicated by functional testing; for example, a functional tester might indicate a fault in the interrupt circuit, whereas the real fault may lie in the serial I/O circuit. If the failure is not in the indicated functional block, the 9100A/9105A at the troubleshooting station can perform its own full functional test to determine the location of the problem.

The 9100A/9105A has very fast built-in functions to test the microprocessor bus, ROM, and RAM, as well as powerful built-in fault condition handling capabilities that ease the communication between the testing functions and the troubleshooting functions.

After stimulus programs and a reference list of parts have been developed for a UUT, the process of testing can be greatly simplified with the TL/1 programming language's *gfi test* command, which uses portions of the 9100A/9105A's Guided Fault Isolation (GFI) database to automate much of the data collection and comparison needed for evaluation of test results.

GFI Troubleshooting:

The 9100A/9105A uses the backtracing method (from bad to good) for its built-in Guided Fault Isolation troubleshooting capability. A functional test locates outputs that appear bad, and GFI starts backtracing from those outputs to locate quickly the failing node. In doing this, GFI uses its database of IC pinouts (the part library, largely supplied by Fluke) and your node list (with part-number references).

The built-in GFI algorithm is efficient at backtracing. However, troubleshooting time can be further reduced by having functional tests provide suggested starting points for GFI (called "GFI hints") as close as possible to the failing node or component. Hints which are close to the fault improve the efficiency of GFI by decreasing the number of nodes that GFI must trace through before reaching the fault.

You can improve GFI's backtracing by:

- Developing functional tests for intermediate functional blocks wherever practical. If a functional test for a major block fails, test the intermediate functional blocks and provide hints which are close to the failure.
- Designing functional tests that, upon failure, measure intermediate nodes in order to provide hints close to the failure. Functional tests can also include fault condition handlers that interpret diagnostic messages to determine where the failure might be located.

(This page is intentionally blank.)

Section 3
Developing Procedures and Programs

UNDERSTANDING THE UUT 3.1.

A UUT should be well understood before functional tests and troubleshooting routines are developed. Taking time at the beginning to study the UUT will result in quicker program development, greater fault coverage, and more accurate fault detection.

Before developing functional test programs and troubleshooting routines:

- Learn what each circuit does, how it works, and how to initialize it.
- Determine the UUT memory map.
- Determine the initialization procedures for each programmable chip.

PARTITIONING THE UUT 3.2.

Circuit partitioning involves dividing the entire circuit into a collection of smaller functional blocks which are easier to understand and test. It is the first step toward a divide-and-conquer method of testing and troubleshooting and it is time well spent. Once the task is done, the functional blocks can be considered as components, each of which receives inputs and generates outputs. Like an IC, a functional block is suspected of being bad if it has good inputs and bad outputs.

Here are some guidelines for partitioning circuits:

- Group circuits by function, making the functional blocks well-defined pieces of the UUT block diagram and as logically distinct as possible.
- If a functional block is large, subdivide it. This will improve troubleshooting efficiency.
- If failure of a circuit can cause failures to appear in many other parts of the UUT, make that circuit a functional block.
- If a circuit requires a unique test setup, make it a functional block.

An Example of Partitioning (The Demo/Trainer UUT) 3.2.1

The Demo/Trainer UUT (Figure 3-1) is an 80286-based system which includes ROM, Dynamic RAM, Parallel I/O, Video, and Serial I/O circuits. It is available from Fluke as an option and is a good example of 16-bit microcomputer systems. Contact a Fluke representative for information about this option.

The test and troubleshooting examples throughout this manual relate to the Demo/Trainer UUT. With it, you can perform the hands-on tests given in the following sections. The complete UUT nodelist, part-reference list, and schematics are shown in the appendices of the manual.

If you do not have a Demo/Trainer UUT, the examples provide enough information so that you can follow the techniques and sample programs and apply the concepts to your UUT.

(1) RS-232 CONNECTOR

(2) VIDEO CONNECTOR

(3) TEST SWITCHES (S1 THROUGH S4)

(4) STATUS LEDs

(5) KEYBOARD CONNECTOR

(6) RESET BUTTON

(7) 80286 MICROPROCESSOR

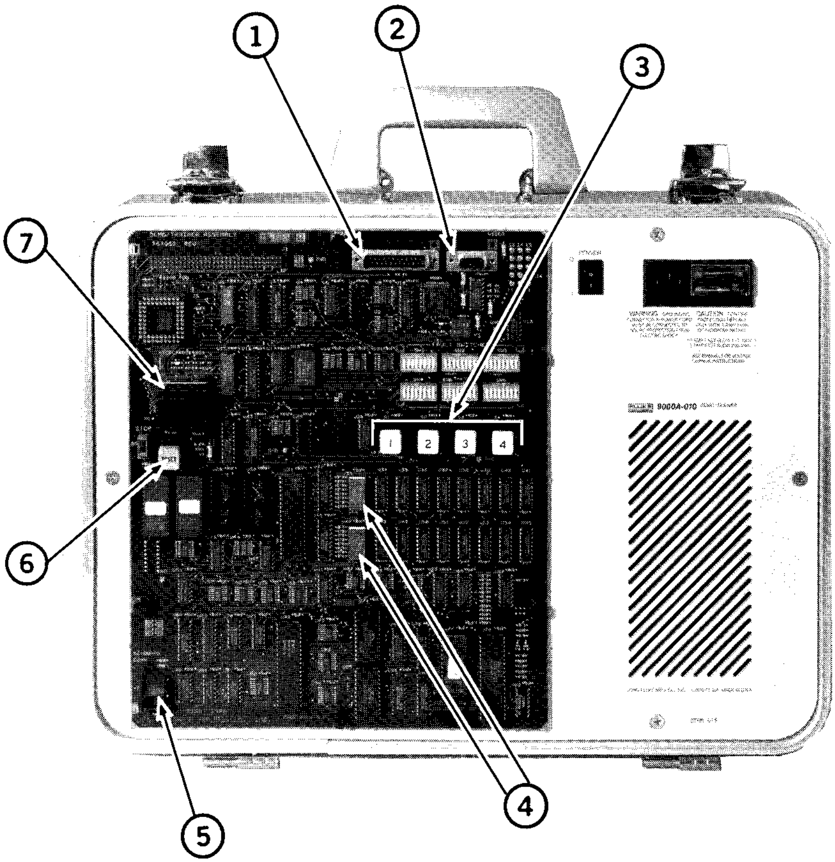


Figure 3-1: Demo/Trainer UUT

A simple block diagram of the Demo/Trainer UUT might show only five blocks: RAM, ROM, Parallel I/O, Serial I/O, and Video. While this is useful as an overview of what the system does, it is inadequate for the development of test and troubleshooting procedures. By subdividing this diagram into smaller sections, we arrive at functional blocks that can be more easily understood. Figure 3-2 shows these smaller blocks, which will be used as examples throughout this manual.

For example, the video circuitry is subdivided into three functional blocks: Video Output, Video Control, and Video RAM. This was done in anticipation that three distinct troubleshooting setups would be needed for the video circuitry. It was also done to reduce troubleshooting time by allowing functional tests to determine which portion of the video circuit has failed before GFI is invoked. Remember, troubleshooting with GFI normally begins at an output node of the failing functional block and backtraces toward good inputs to that block. Subdivision allows GFI to begin backtracing closer to the fault. For similar reasons, the dynamic RAM circuit is subdivided into RAM and Dynamic RAM Timing.

The microprocessor itself is shown in Figure 3-2 as a separate functional block for a good reason: when the pod replaces the microprocessor, it becomes a known-good functional block. All outputs from this circuit can be directly controlled by the 9100A/9105A. The pod checks for drivability on every UUT access and reports if there is a loading problem.

The Bus Buffer is partitioned separately, not for reasons of clarity, but so that it can have its own functional tests. If this circuit has a fault in it, the fault will cause most of the other functional blocks to also fail. So if the UUT fails a functional test, it is more efficient to check the Bus Buffer early in the troubleshooting process.

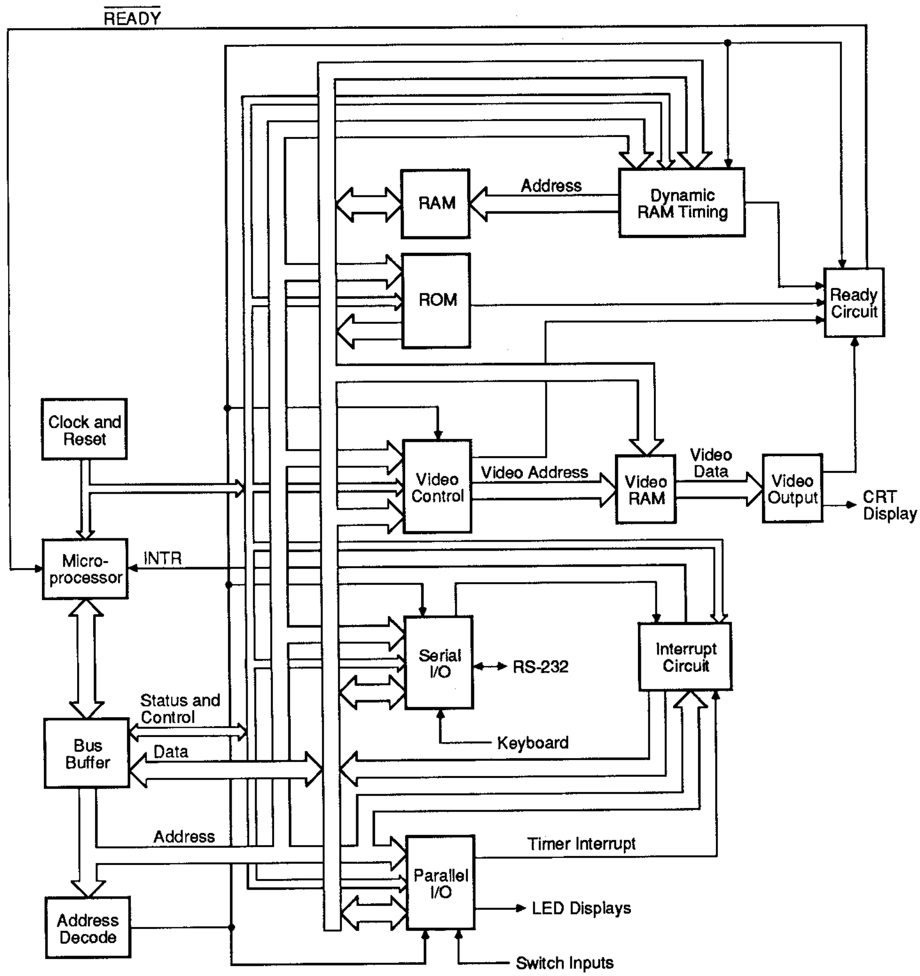


Figure 3-2: Demo/Trainer UUT Functional Blocks

The Advantage of Partitioning 3.2.2

After the partitioning is done, step back and look at the resulting detailed block diagram. Imagine that a functional test has been developed for each individual block. If a novice user has nothing but this block diagram and the collection of individual block tests, he can make a fair degree of progress toward troubleshooting and repairing a complex system.

With thoughtful partitioning, a board may be determined to be good without running all of its individual functional block tests; some functional blocks can be assumed to be good if tests for other functional blocks that depend on them are good.

Through partitioning, the large problem of testing and troubleshooting a complex system can be subdivided into smaller, more easily handled problems.

PROGRAM DEVELOPMENT SEQUENCE 3.3.

There are four levels in programming with the 9100A, as shown in Figure 3-3. Each level is a building block for the next level of programming.

The sequence shown below is the most efficient method of developing programs if you plan to develop both functional testing and GFI troubleshooting capability. This is because the functional block tests in Step 2 can often use the GFI stimulus programs developed in Step 1 to test the outputs of a functional block (See Section 3.5.1 for additional explanation). However, in other situations, you may need to use your 9100A/9105A for functional testing as soon as possible, even before troubleshooting programs can be developed. In this case you may want to do Steps 2 and 3 before doing Step 1.

The four steps of programming are:

1. *Stimulus programs* for nodes are created and *responses* from a known-good UUT are learned. (Sections 3 and 4 of this manual)

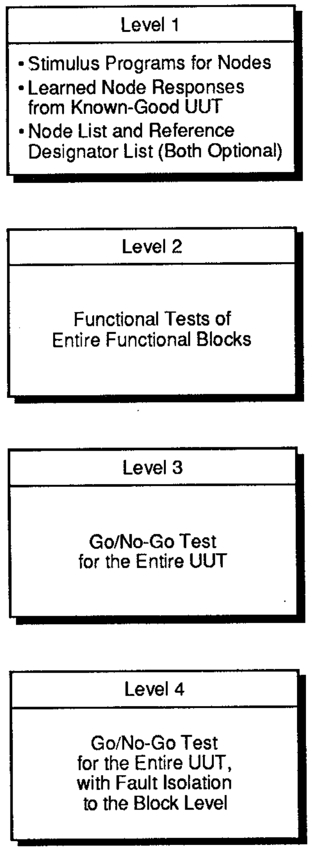


Figure 3-3: Building-Block Programming

If the node list and reference designator list are also created, this level will allow not only testing a node, but also automated backtracing from a bad node to the fault.

2. *Functional tests* of entire functional blocks are created. The *gfi test* command can use your stimulus programs and learned responses for fast, effortless functional tests of these blocks. (Sections 3 and 4 of this manual.)

3. *A UUT go/no-go test* is built from the functional tests of functional blocks. (Section 5 of this manual)

4. *Diagnostic programs* are created by adding fault handlers and *gfi hint* commands to the UUT go/no-go test. The diagnostic program traps faults and initiates tests of functional blocks that may be responsible for the fault, thereby isolating the functional block that is causing the UUT to fail. When the failing output of the block is found, then a GFI hint is generated and GFI will begin backtracing the failing circuitry. (Section 6 of this manual)

After the fourth programming level, the go/no-go test will isolate the failing functional block and then will start GFI troubleshooting (Section 7 of this manual) to backtrace to the bad node or component.

STIMULUS PROGRAMS AND LEARNED RESPONSES 3.4.

Stimulus programs and learned responses constitute the first of the four levels in programmed testing and troubleshooting, as shown in Figure 3-4.

Stimulus programs create predictable node activity so that one or more nodes can be characterized. When properly designed, these programs are usually short and simple. With the 9100A,

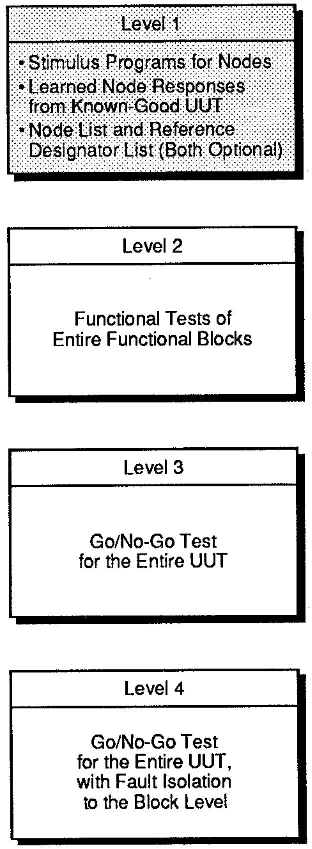


Figure 3-4: Functional Tests for Nodes (Level 1)

the most difficult task related to writing stimulus programs is understanding how the UUT operates.

Learned responses are the responses of a known-good UUT to the stimulus programs. The 9100A/9105A can store these responses from a known-good UUT for use in testing other identical UUTs.

Rules for Stimulus Programs 3.4.1.

Stimulus programs must follow these rules, to ensure that GFI troubleshooting reaches correct conclusions:

- *Measure Outputs.* Use stimulus programs to characterize signal sources (outputs) only.
- *Provide Initialization.* If a circuit ever requires initialization, place an initialization procedure in the stimulus program. The initialization must be performed before the measurement is started. The best place for initialization is near the beginning of the stimulus program.
- *A Separate Program for Each Signal Source on a Node.* Create a separate stimulus program for every signal source (output) on a node. A bidirectional line between two components should have at least two stimulus programs, one for each direction of data flow. Buses should have at least one stimulus program for every component that can output on the bus.
- *A Separate Program for Each Mode of Output.* Create a separate stimulus program for each way that an output is operated in normal UUT operation. For example, if a buffer on an address bus is stimulated by the microprocessor and also by a DMA controller, create two stimulus programs for the outputs of the buffer: one from the microprocessor, and one from the DMA controller.
- *Keep It Simple.* If a stimulus program becomes complex, find a way to split it up into more than one program. For example, consider a PIA chip connected to a data bus and a keypad that can be read through the PIA. The stimulus

program that enables the PIA data lines onto the data bus should initialize registers at the beginning of the program, then the program should read the registers in the PIA chip.

The Flow of Stimulus Across the UUT 3.4.2.

Stimulus programs are unrelated to functional blocks. Functional blocks are only defined to help with functional testing.

Stimuli generally flow from the microprocessor kernel toward the outputs of the UUT. Some stimulus programs may characterize the outputs of many components while other stimulus programs may characterize only a few outputs.

The key to efficient stimulus programs is to begin at some outputs of the microprocessor kernel that can be stimulated. Stimulate these outputs and trace through the circuit to see how many other output nodes can be characterized. Find nodes that have not been characterized, and decide what is needed to stimulate them. Then, see how many nodes are covered. Continue this process until each node is covered by at least one stimulus program.

A good way to keep track of which nodes have been covered is to use a set of colored markers. Using a separate color for each stimulus program, color in a small region around the output nodes which will be stimulated by that program (remember, stimulus programs only apply to signal sources). Even for a complex UUT, the strategy for creating stimulus programs for an entire UUT can be "mapped out" in a few hours. The time spent will promote better software organization and speed up both the writing of stimulus programs and the process of learning the responses.

Keep in mind the rules described in the Section 3.4.1, and remember that some outputs will be characterized by more than one stimulus program.

Stimulus Program Planning 3.4.3.

Stimulus programs and their matching response files are used by the 9100A/9105A Guided Fault Isolation (GFI) to backtrace through a failing circuit in a UUT to find the fault. The stimulus programs exercise a portion of the UUT circuitry in order to produce repeatable activity at circuit nodes to be measured. This activity at each node is measured on a known-good UUT and a characterization of this known-good response is stored in a response file. Each response file stores characterizations of how some circuit nodes on a good UUT perform as a result of its matching stimulus program. There is one response file for every stimulus program.

Each of the fourteen functional blocks in Section 4 includes a figure titled "Stimulus Program Planning." Figure 3-5 shows an example of such a figure.

The purpose of the stimulus program planning diagrams is to illustrate how to design the stimulus programs for a UUT. In general, you should begin the process of creating stimulus programs by identifying outputs from the microprocessor that can be exercised (such as the address bus, data bus, and control lines). Characterize all those nodes that are stimulated, then find some nodes that are not characterized and design stimulus programs to stimulate them. In general, start at the microprocessor and work outwards to the I/O devices. Continue until all nodes in the UUT are characterized.

The left-hand page of Figure 3-5 shows six blocks that represent six stimulus programs and their matching response files. Each of these stimulus program/response file pairs are used to stimulate and characterize nodes in this functional block.

The block for the *addr_out* stimulus program shows that it stimulates the outputs of the address buffers: U16, U2, and U22. As you examine each of the stimulus program planning figures in Section 4 of this manual, you will notice that the *addr_out* stimulus program stimulates nodes in many of these functional blocks. This is because the stimulus programs are not

(This page is intentionally blank.)

Stimulus Program Planning

PROGRAM: ADDR_OUT

EXERCISES ALL ADDRESS LINES FROM THE MICROPROCESSOR

MEASUREMENT AT:

U16-19,16,15,12,9,6,5,2
U2-19,16,15,12,9,6,5,2
U22-19,16,15,12,9

PROGRAM: DATA_OUT

EXERCISES ALL DATA LINES AS OUTPUTS FROM THE MICROPROCESSOR

MEASUREMENT AT:

U3-11,12,13,14,15,16,17,18
U23-11,12,13,14,15,16,17,18

PROGRAM: CTRL_OUT1

EXERCISES CONTROL LINES FROM THE MICROPROCESSOR USING POD ADDRESS SYNCHRONIZATION

MEASUREMENT AT:

U22-5,6
U57-8
U15-16
U45-8

PROGRAM: CTRL_OUT2

EXERCISES CONTROL LINES FROM THE MICROPROCESSOR USING DATA SYNCHRONIZATION

MEASUREMENT AT:

U15-17,8,9,12,11,13,5
U56-6
U45-8
U5-8

PROGRAM: CTRL_OUT3

EXERCISES CONTROL LINES FROM THE MICROPROCESSOR USING INTERRUPT ACKNOWLEDGE SYNCHRONIZATION

MEASUREMENT AT:

U15-17,5,8,9,12,11,13
U56-6
U45-8
U5-8

PROGRAM: ROM1_DATA

EXERCISES ALL DATA LINES AS INPUTS TO THE MICROPROCESSOR

MEASUREMENT AT:

U3-9,8,7,6,5,4,3,2
U23-9,8,7,6,5,4,3,2

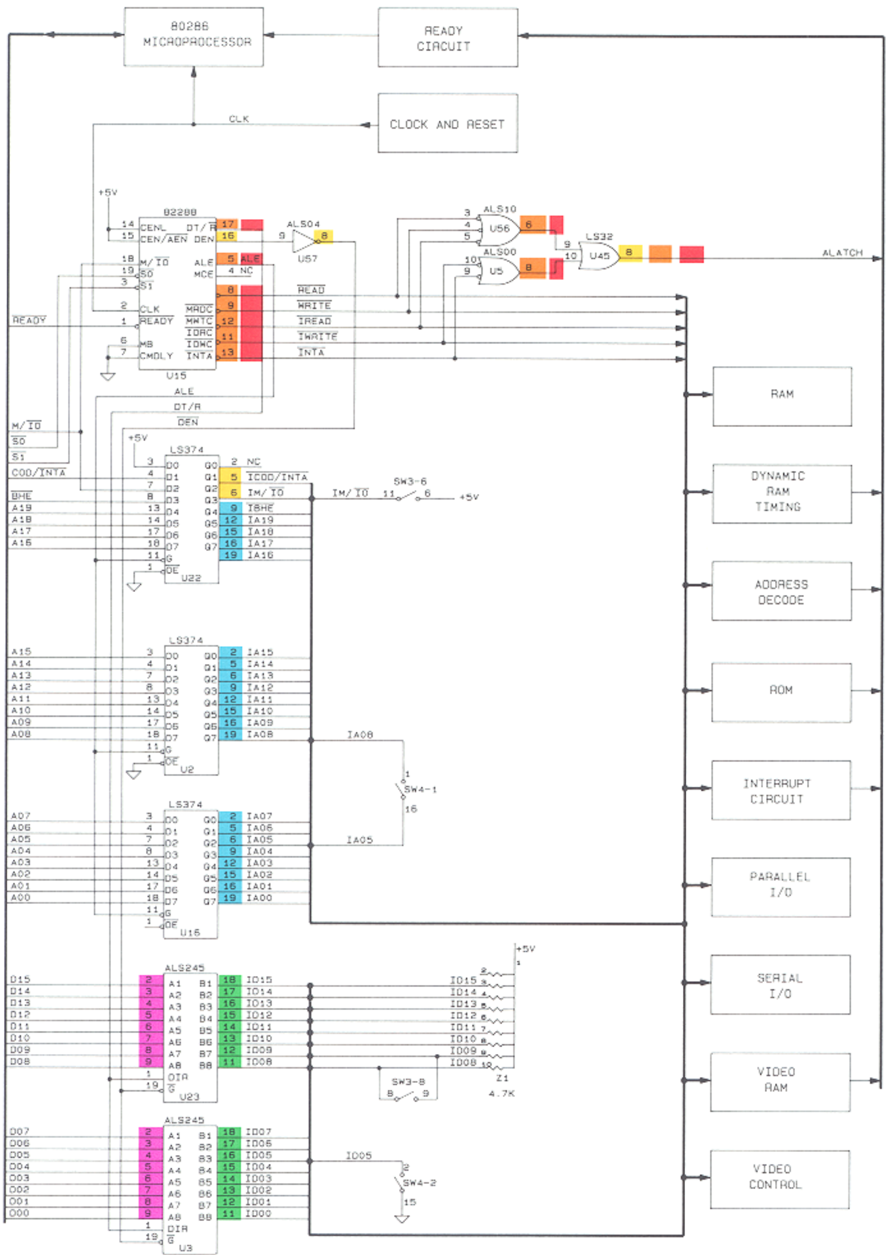


Figure 3-5: Example of Stimulus Program Planning Figure

limited by functional block boundaries and typically will stimulate nodes over several functional blocks.

Figure 3-5 shows that the *data_out* stimulus program stimulates the bidirectional data bus when the microprocessor is sending out data (a write operation). The figure also shows that the *rom1_data* stimulus program is used to stimulate the data bus buffers U3 and U23 when data is flowing into the microprocessor (a read operation).

The other three stimulus programs shown (*ctrl_out1, ctrl_out2,* and *ctrl_out3)* stimulate the control line outputs from the microprocessor and bus controller IC (an 82288 chip); *ctrl_out1* stimulates the control lines using pod data synchronization; *ctrl_out2* stimulates the control lines using pod address synchronization; *ctrl_out3* generates an interrupt acknowledge cycle and stimulates the control lines using interrupt acknowledge synchronization.

When planning the stimulus programs for your UUT, you can use colored pens to map out which outputs in your UUT will be covered by which stimulus programs. You should start with the address signals, data signals, and control signals. After that, you can plan what is required for stimulus programs for other outputs in your UUT, working from the kernel toward the I/O of the UUT.

Suggestions about Stimulus Programs 3.4.4.

The actual stimulus programs used for the Demo/Trainer UUT are listed in Section 4 of this manual. Some stimulus programs stimulate nodes in several functional blocks and other stimulus programs stimulate only a few nodes. The fact that, in Section 4, stimulus program coverage is organized by functional blocks does not imply that the stimulus programs observe functional-block boundaries. Stimulus programs do not care about functional block boundaries and usually will exercise nodes across functional-block boundaries.

Each of these stimulus programs in Section 4 follows a standard form that can be divided into five parts:

- Initialize the circuit and define the measurement device.
- Set up the stimulus and measurement devices.
- Start the measurement.
- Stimulate the circuit.
- Stop the measurement.
- Restore any conditions changed by the setup, above.

Figure 3-6 shows a simple stimulus program with each of the six parts labeled. Circuits that contain programmable components require initialization. Any circuit that needs initialization should have it provided in the stimulus program. This is necessary since there is no way to determine the order in which stimulus programs will be run when GFI or UFI troubleshooting is performed. Therefore, each stimulus program should perform any initialization the circuit needs.

Defining the Measurement Device

Most stimulus programs use the I/O modules and the probe as measurement devices. When GFI or UFI is using the I/O module as a measurement device, a message is displayed which prompts the operator to clip onto the component and to push the Ready button on the clip module. When the operator does this, the 9100A/9105A identifies the I/O module and the side (A or B) being used.

GFI or UFI can tell a stimulus program which device is being used. It is a good idea to write your stimulus programs so that the measurement device name is obtained from GFI or UFI rather than specifying the device name in the stimulus program. Getting the name from GFI or UFI has the advantage that the operator can connect a clip to either side of any of the four I/O modules. The operator can use several I/O modules, each with a

```
program data_bus

   if (gfi control) = "yes" then              ! DEFINE THE MEASUREMENT
      devname = gfi device                    ! DEVICE
   else
      devname = "/mod1"
   end if

   podsetup 'enable ~ready' "off"             ! SET UP THE MEASUREMENT
   podsetup 'report power' "off"              ! AND STIMULUS DEVICES
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   setspace space (getspace space "memory", size "word")
   reset device devname
   sync device devname, mode "pod"
   sync device "/pod", mode "data"

   arm device devname                         ! START THE MEASUREMENT

      rampdata addr 0, data 0, mask $FF       ! STIMULATE THE CIRCUIT
      rampdata addr 0, data 0, mask $FF00

   readout device devname                     ! STOP THE MEASUREMENT

   podsetup 'enable ~ready' "on"              ! RESTORE READY

end program
```

Figure 3-6: Parts of a Stimulus Program

different size of clip, and the stimulus program will still work with any of these configurations.

The stimulus program shown in Figure 3-6 uses the TL/1 *gfi control* command to determine that GFI or UFI is executing the stimulus program. If GFI or UFI is executing the program, the *gfi device* command is used to return the name of the measurement device.

Using the I/O Module as a Stimulus Device

Each I/O module can be used to overdrive a limited number of components. The same I/O module or a different I/O module may be used to measure circuit response.

For example, suppose an I/O module is used to perform a truth-table test of a 7400 NAND gate. The I/O module is clipped to the 7400. Pins 1 and 2 of the 7400 are inputs and pin 3 is the output. The same I/O module drives the inputs and measures CRC signature responses on the output. Each time the pattern is driven on the inputs, the output's CRC signature is sampled.

In this example, the same I/O module is used as the stimulus device and as the measurement device. In some cases, more than one clip is used in stimulating and measuring circuit response. The *gfi device* command returns the device name of the measurement device being used.

The stimulus program should use the *clip* command or the *assoc* command to identify the stimulus device. This command will prompt the operator to clip to the component and push the Ready button on the clip module. Using this method to identify the stimulus device creates a program that allows the operator to use any I/O module for the measurement device and any other I/O module for the stimulus device.

Two steps are necessary to drive a pattern on a set of inputs. First, a *storepatt* command is written for each input pin to be driven. If five inputs are to be driven, five *storepatt* commands are needed. After the patterns are defined by *storepatt*, a *writepatt* command is used to clock out all the defined patterns in parallel.

The I/O module has 40 lines. Clips have 14 to 40 pins. Each clip maps to the I/O module lines in a different way. The 40-pin clip is one for one (clip-pin 1 is mapped to I/O module-line 1, etc.). The other clips have a different mappings (shown in appendix B of the *Technical User's Manual).*

The TL/1 commands that involve an I/O module refer to pin numbers in three different ways. These TL/1 commands have a parameter that specifies the device name. If the device name is an I/O module name (such as "/mod1"), any pin numbers in that command will be treated as I/O module line numbers. If the device name is a clip module name (such as "/mod1A"), any pin numbers in that command will be treated as clip module pin numbers. And, if the device name is a reference designator (such as "U14"), any pin numbers in that command will be treated as component pin numbers.

If the device name is a reference designator, the component must have been clipped in response to a request from GFI, or in response to a TL/1 *clip* command prior to being used in an I/O module command.

Consider the example of a 7400 that is to have pins 1 and 2 driven by the I/O module. The reference designator for the 7400 is U3. The following TL/1 commands will perform a truth table test on one gate in the 7400:

```
dev = clip ref "U3", pins 14
storepatt device "U3", pin 1, patt "1010"
storepatt device "U3", pin 2, patt "1100"
arm device dev
    writepatt device dev
readout device dev
```

The *clip* command must be used here to define the I/O module and the I/O module side (A or B) that is clipped to U3. The two *storepatt* commands define the pattern to drive on pins 1 and 2 of U3. Because a reference designator was used as the device name (rather than a clip module name like "/mod1A") in the *storepatt* commands, any size of clip can be connected to U3. Suppose a 16-pin clip is connected to U3. The 9100A/9105A knows, from the *clip* command, that the part has 14 pins. As long as pin 1 of the 16-pin clip is placed on pin 1 of the component, the 9100A/9105A will map the pins correctly. GFI and UFI are also able to use clips larger than the component they clip over to measure the response of that component.

The *arm* and *readout* commands start and stop the measurement. Inside the measurement, the *writepatt* command sends the defined patterns to the specified pins. Because the *writepatt* command is surrounded by the *arm* and *readout* commands, a CRC signature can be gathered on the input pins or the output pins of the component, as determined by GFI.

In general, stimulus programs can be written so that any I/O module can be used either for stimulus or measurement. To do this, use the device name returned by the TL/1 *gfi device* command for measurement devices. If the stimulus program uses the I/O module as a stimulus device, use the *clip* statement and reference names for device names in the TL/1 commands (pin-number parameters) that interact with the I/O module.

FUNCTIONAL TESTS 3.5.

Functional tests of blocks are the second of the four modular levels in programming the 9100A, as shown in Figure 3-7. In this second level, tests of functional blocks are created from stimulus programs and response files.

The goal of a functional test is to evaluate the performance of the functional block and to decide whether the entire block is good (passes) or bad (fails). As shown in Figure 3-8, such a test can be divided into the following steps:

1. Initialize the circuits in the block (if necessary).
2. Stimulate the inputs to the block.
3. Measure the outputs from the block.

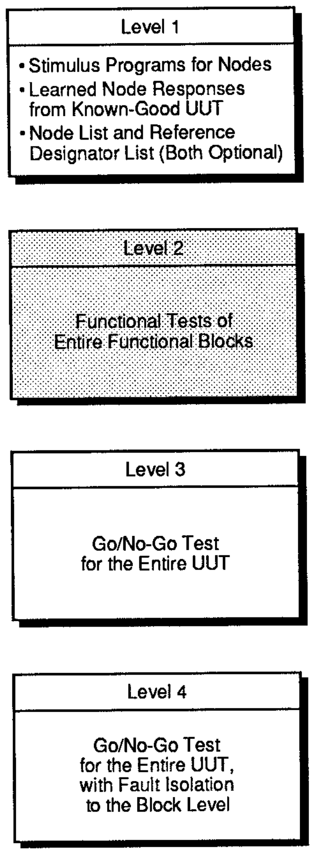


Figure 3-7: Functional Tests for Functional Blocks (Level 2)

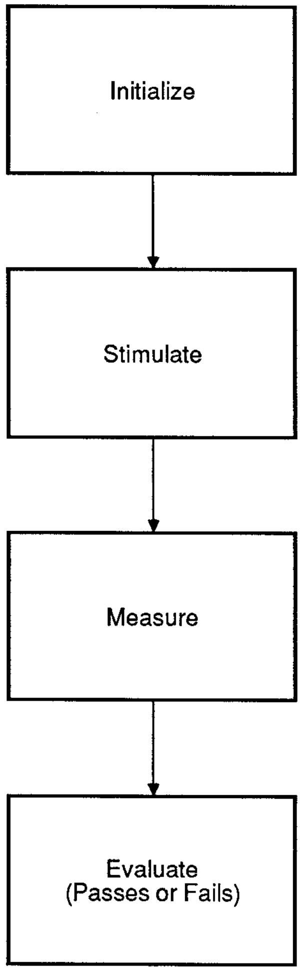


Figure 3-8: Functional Test Elements

4. Evaluate each output and decide whether the output passes or fails. If all outputs pass, the block is good, otherwise it is bad.

Programmed Functional Tests 3.5.1.

Programmed functional tests perform all four functional test steps automatically. There are three basic methods of writing functional tests for each functional block in the UUT:

- *Using the TL/1 built-in functional test commands* - Use for testing the microprocessor bus, RAM, and ROM.
- *Building on stimulus programs* - Use the *gfi test* command to build on stimulus programs and learned responses.
- *Writing TL/1 programs which are independent of GFI* - These programs must perform all four functional test steps.

Using Built-In Functional Test Commands

For some functional blocks, such as the microprocessor bus, ROM, and RAM, you should not use the *gfi test* command. Instead, these blocks can be tested with the built-in TL/1 functional test commands *testbus, testramfast, testramful, and testromfull.*

Building On Stimulus Programs

Stimulus programs and learned responses are used to decide if a node passes or fails. The TL/1 programming language has a command called *gfi test,* which performs Steps 1 through 3 of functional testing and part of Step 4 (see Figure 3-8).

The *gfi test* command tests an entire component (if the I/O module is the measurement device) and returns a passes or fails result. The command runs all stimulus programs associated with all pins on the component and compares the responses to the learned responses. It returns a "passes" result if all pins on the component are good.

Suppose the buffers of a 24-bit microprocessor address bus are tested as a functional block. If the functional test is written without the *gfi test* command, the test would perform the following operations:

1. Stimulate the address bus.

2. Capture signatures on the 24 address lines.

3. Compare captured signatures with known-good signatures (24 if/then statements).

The same functional test using the *gfi test* command would require only three *gfi test* commands. Using this command decreases the time required to write functional test programs.

Using the *gfi test* command provides an additional important advantage. When it is used, the known-good responses are automatically retrieved from the the 9100A/9105A's response files. Whenever a board is revised, the response files must be updated. If a functional test contains known-good response information built into the program, rather than stored in response files, both the response file and the functional test program must be updated if the board is revised.

You may need to develop a test quickly for just one functional block and avoid writing stimulus programs or learning responses for the entire UUT. In this case, the following procedure will help ensure that the functional test you write will later integrate well into the functional test for the entire UUT:

1. Make a plan for the stimulus programs you will need to cover the entire UUT. This usually takes several hours.

2. Write the stimulus programs needed to test the block in question.

3. Write the functional test for the block using the *gfi test* command wherever possible.

4. After the test for the block is finished, you can continue with the process of writing stimulus programs and learning responses for the rest of the blocks in the UUT.

Functional Tests That Are Independent of GFI

You can also write functional tests that do not require the use of stimulus programs and response files. If so, these tests should also contain the functional test elements shown in Figure 3-8.

Programmed Functional Test Examples 3.5.2.

The programmed functional tests for each functional block in the Demo/Trainer UUT are listed in Section 4 of this manual. The simplicity of these functional tests results from using the *gfi test* command and the built-in test functions.

It is tempting to write a functional test without first writing stimulus programs. However, a penalty is paid for this approach in two ways: it can actually take longer if stimulus programs are not created first, since the 9100A/9105A already has built-in functions to do much of the functional testing once stimulus programs are created. Second, stimulus programs will have to be written anyway before GFI troubleshooting can be used.

The sequence of steps shown in Figure 3-7 will in most cases give you the best results in the shortest time. Each increment of programming investment will result in better performance and productivity.

Keystroke Functional Tests 3.5.3.

A UUT may be tested using only 9100/9105A front panel keystrokes. Keystroke testing also involves each of the four functional test steps (initialization, stimulation, measurement, and evaluation) shown previously in Figure 3-8, but the operator performs these steps rather than having the 9100A/9105A do them with TL/1 programs. If you wish, these steps can be stored in keystroke sequences by using the SEQ key on the operator's keypad.

Each of the fourteen functional blocks in Section 4 has a "Keystroke Functional Test" figure like the example shown in Figure 3-9. The purposes of these figures are the following:

- Show the schematic diagram for that functional block.
- Show the inputs to the functional block from other functional blocks.
- Show the outputs of the functional block to other functional blocks.
- Identify the 9100A/9105A measurement and stimulus devices used to test the block and to identify where those devices are connected.
- Show the expected node response information from performing the functional test sequence for that block.

Figure 3-9 is a typical example of a Keystroke Functional Test figure that you will see for each of the functional blocks described in Section 4 of this manual. In most cases, the functional blocks to the left of the schematic are those which provide input to the functional block shown in schematic form. In most cases, any functional blocks to the right of the schematic are receiving the outputs of the functional block shown in schematic form. The arrows show the direction of the signals between the functional block boxes and the functional block shown in schematic form.

Notice the left-hand page of Figure 3-9. At the top of the page is a box labeled CONNECTION TABLE. The left column of this

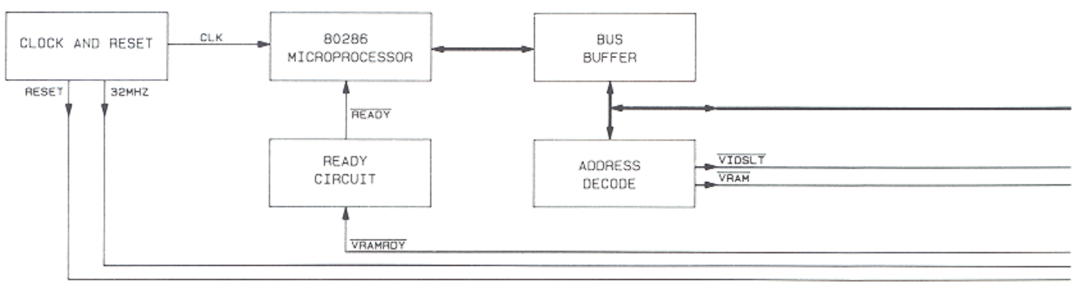
Keystroke Functional Test

CONNECTION TABLE

STIMULUS	MEASUREMENT CONTROL	MEASUREMENT
(NONE)	I/O MOD CLOCK U78-33 START U88-13 STOP U88-13 ENABLE U78-12	I/O MOD U72

RESPONSE TABLE

SIGNAL	PART PIN	I/O MOD PIN	SIGNATURE
DAD00	U72-34	34	4155
DAD01	-33	33	3F33
DAD02	-32	32	A65A
DAD03	-31	31	9024
DAD04	-30	30	DE6D
DAD05	-29	29	D6FA
DAD06	-28	28	7AC3
DAD07	-27	27	0477
DAD08	-26	26	E941
DAD09	-25	25	88B8
DAD10	-24	24	60B0
DAD11	-23	23	D869



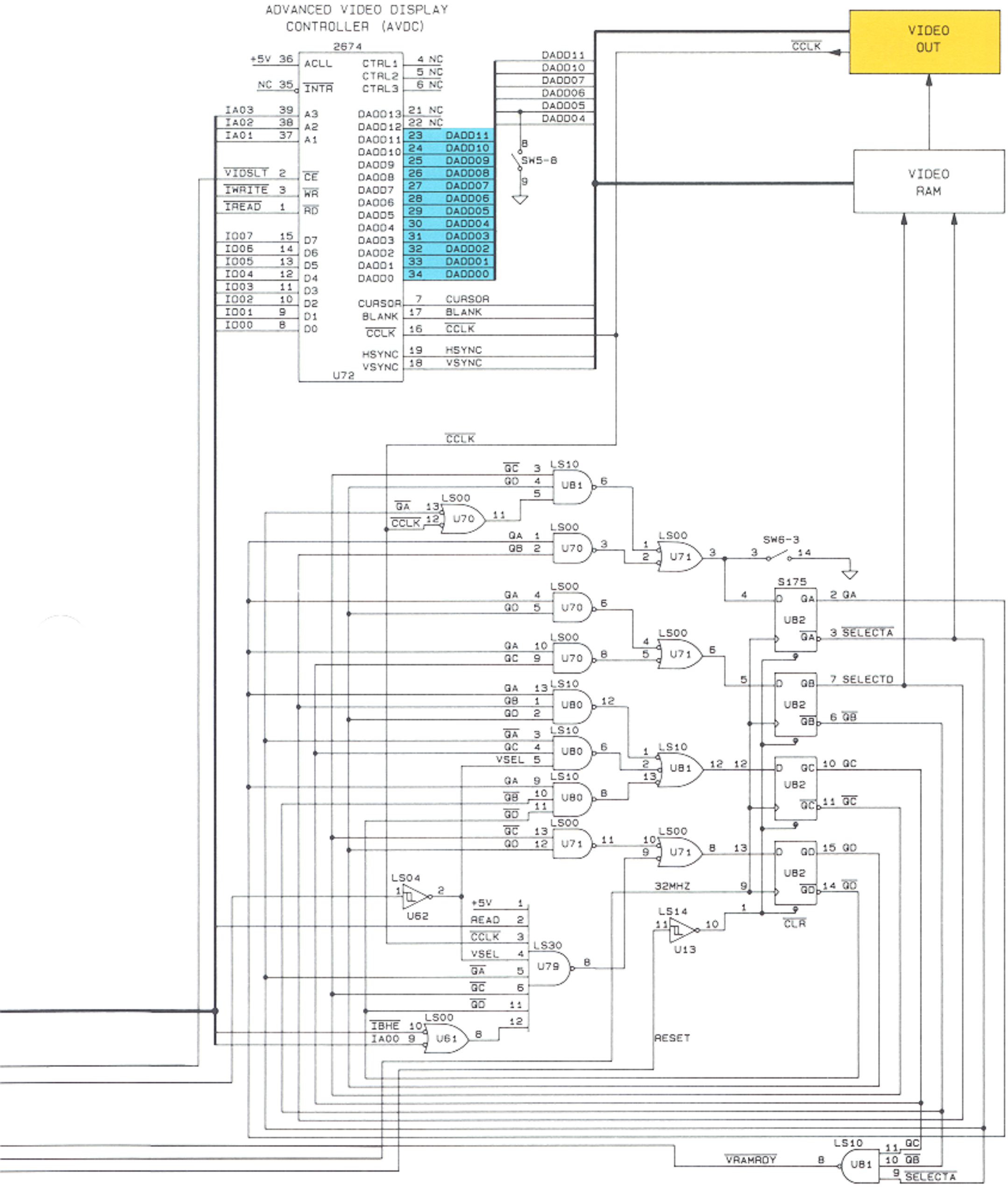


Figure 3-9: Example of Keystroke Functional Test Figure

table, labeled STIMULUS, shows what 9100A/9105A device is used to provide stimulus to the functional block shown in schematic form and where the connection is made. In the example shown in Figure 3-9, no stimulus is provided because this diagram is part of the video circuit and, once initialized, the video circuit constantly runs with no additional stimulus. In many of the keystroke functional test diagrams in Section 4, the STIMULUS column will indicate that the pod or I/O module is used.

The right column of the CONNECTION TABLE, labeled MEASUREMENT, shows which 9100A/9105A device is used to measure circuit response for the Keystroke Functional Test. The measurement device can be the probe, the pod, or an I/O module. This column also shows the components or nodes in the circuit that are to be measured.

When the I/O module is the measurement device and its external control lines are used, a third column, labeled MEASUREMENT CONTROL, shows where to connect the START, STOP, CLOCK, and ENABLE lines.

The RESPONSE TABLE shows the names of the signals to be measured, the component and pin numbers to be measured, the corresponding pin numbers used by the I/O module, and the known-good measurement value for each signal.

Section 4
Functional Block Test and Troubleshooting Examples

This section is organized into fifteen sub-sections. The first fourteen sub-sections each contain the following information:

- General discussion of a kind of functional block.
- Testing and troubleshooting approaches.
- Keystroke testing procedure for Demo/Trainer UUT.
- Functional test program listing for Demo/Trainer UUT.
- Stimulus programs and responses for troubleshooting.
- Summary of solution showing all files and programs needed to test and troubleshoot the functional block.

The last sub-section covers types of circuits not found in the Demo/Trainer UUT and is therefore organized differently than the above.

For the purpose of learning how the 9100A/9105A works, each of the fourteen functional blocks can be considered to be a self-contained portion of the UUT. The Summary of Solution page at the end of each sub-section shows all of the files required to test or troubleshoot that functional block.

Only a subset of all the functional blocks in a UUT needs testing to determine whether the UUT is good or bad. This is because testing the major functional blocks indirectly tests the other blocks as well. (See Section 5 for more details on functional testing strategy for a complete UUT). For the Demo/Trainer UUT, testing the following major functional blocks will be sufficient for a UUT go/no-go test functional test:

- Microprocessor Bus.
- ROM.
- RAM.
- Parallel I/O.
- Serial I/O.
- Video Output.

The remaining functional blocks covered in this section are useful for troubleshooting the UUT if it fails the go/no-go UUT functional test:

- Dynamic RAM Timing.
- Video Control.
- Video RAM.
- Bus Buffer.
- Address Decode.
- Clock and Reset.
- Interrupt Circuit.
- Ready Circuit.

You will find that the Dynamic RAM Timing, Video Control, and Video RAM functional blocks come from subdividing the RAM and Video blocks into smaller-size blocks.

MICROPROCESSOR BUS FUNCTIONAL BLOCK 4.1.

Test Access to the Microprocessor Bus 4.1.1.

The term "test access" refers to the point at which the pod connects to the Unit Under Test (UUT). In most cases, a UUT's microprocessor or microcontroller is replaced in its socket by the pod, but this is not always the case. For example, if the microprocessor is soldered in, the UUT can be designed to allow a bus-cycle emulation pod to access the bus through a test connector.

The test access allows the 9100/9105A pod to perform reads and writes on the microprocessor bus. The pod can selectively ignore inputs which normally would go directly to the microprocessor. Thus, any faults that would stop the microprocessor can be ignored by the pod, and testing can proceed as though the microprocessor were in a good circuit and functioning properly.

The pod uses microprocessor bus emulation as the primary means of testing and troubleshooting. It can generate stimuli to the UUT and capture the responses in conjunction with other 9100/9105A stimulus and measurement devices, thereby providing excellent troubleshooting capability for all microprocessor signals. The pod can perform basic microprocessor read and write operations, various stimulus functions built from multiple reads and writes, and built-in tests such as bus, RAM, and ROM. The pod also verifies that the microprocessor power supply is within tolerance, and that all power supply pins are connected.

A little foresight in the design of test access can make testing much easier. Here are some general guidelines to facilitate testing:

- Provide clearance around all devices. This allows access for the pod connectors (to replace the microprocessor or plug in a test socket), for a component extraction tool (if components are hard to remove, especially pin-grid array

(PGA) types), and for I/O module clips (especially if adjacent components must be clipped simultaneously).

- Provide some means to access the microprocessor bus if the microprocessor is soldered in. An additional microprocessor socket or card edge connector can provide this access. Consider providing some form of test access even though the factory or service center may use test fixturing, since this allows testing in field situations where no test fixturing is available.
- Use resistors to the power supply or ground to establish static logic levels on unused inputs instead of directly connecting inputs to power supplies or ground. This allows the 9100/9105A to drive these inputs.
- If there are microprocessor inputs that will force most of the microprocessor outputs to a high-impedance state, design the UUT so that the 9100/9105A can drive these inputs.
- If there are microprocessor outputs that cannot be placed in a high-impedence state, design the UUT so that these outputs are buffered and the buffer outputs can be turned off or overdriven by the 9100/9105A.
- Allow the UUT clock to be suppressed to permit the UUT to operate with an external clock.
- Ensure that vendors' specifications for load and timing margins are not violated and, if possible, allow for a further margin.
- Design so that all signals at a ROM chip can be latched by the I/O module with DATA synchronization.
- Pull up all lines carrying data signals to a logic 1 through resistors.

Considerations for Testing and Troubleshooting 4.1.2.

Kernel Testing

The combination of the microprocessor, ROM and RAM is collectively referred to as the kernel. The primary advantage of Fluke test and troubleshooting equipment for microprocessor-based UUTs over equipment from other vendors is its ability to troubleshoot dead kernels.

If any part of the kernel malfunctions, very little else works properly. One basic strategy is to test the kernel first, then test the other functional blocks surrounding the kernel.

The 9100/9105A has a comprehensive built-in test for the unbuffered microprocessor bus. This bus test is a series of reads and writes at different addresses while monitoring microprocessor outputs for faults. The bus test is described in detail in Section 6.2.1 of the *Technical User's Manual.* With this bus test, the 9100/9105A can determine stuck or tied lines on all outputs from the microprocessor bus. During a bus test, active interrupts or forcing signals that cause the microprocessor to malfunction will be intercepted by the pod and reported to the 9100/9105A unless they have been specifically disabled with the *podsetup* command in TL/1 or the SETUP POD command on the operator's keypad. The bus test will also report a bad power supply or an inactive clock.

Figure 4-1 summarizes the major conditions reported by the bus test. Faults such as stuck bus lines, missing clocks, and low UUT voltages must be cleared before further testing can proceed.

Finding the source of bus faults may be complicated by multiple bus-master and intervening buffers. For example, a buffer may be loading the bus because its enable line is asserted due to faulty circuitry back several logic gates from the bus. If there are several bus masters, it may be unclear where the fault is. Bus masters may be identified as **masters* in the node list. The

Signal Group	*Condition*	*Example Message*
Address Lines	stuck, tied	addr line A9 pod pin 22 stuck high
Data Lines	stuck, tied	data line D8 pod pin 37 tied data line D9 pod pin 39 tied
Control Lines	stuck, tied	control line HLDA pod pin 65 not drivable
Interrupt Lines	active	interrupt ~INTR pod pin 57 active
Forcing Lines	active	forcing signal RESET pod pin 29 active
Clock	inactive	pod timeout bad UUT clock
Power Lines	out of tolerance	bad UUT power supply

Figure 4-1: Conditions Reported by the BUS TEST

masters identification, combined with independent stimulus programs for each bus master, assist GFI in backtracing faults identified on buses.

For more information on **masters*, stimulus programs, and response files, see Section 7 of this manual and Section 5.5 of the *Programmer's Manual.*

Basic Bus Cycles

It is often useful to perform a series of reads and writes to verify proper operation of basic bus cycles. To do this, you need the address map of the UUT. You can verify bus-cycle operation with reads from and writes to RAM, ROM, or other memory-mapped VLSI devices such as PIAs, DMA controllers, SCSI controllers, and UARTs. If your UUT's microprocessor allows transfers of different data widths (byte, word, long word), transfers with these different data widths should be verified.

If reads do not return the correct data when no major bus faults are indicated by BUS test, try the built-in RAM test or ROM test. RAM test checks the ability to read and write all RAM cells specified in the address range. ROM test checks the ability to read from ROM and verifies the ROM signature. Other kernel-related functional blocks, such as Address Decode, Bus Buffers, or Ready circuitry should also be tested, as described later in Section 4 of this manual.

Synchronization Modes

When you are troubleshooting faults related to bus cycles, it is useful to synchronize the probe or I/O module to pod operations. The pod itself can be synchronized to different parts of the bus cycle that may be appropriate for a particular test. For example, a microprocessor with multiplexed address and data will output the address only during the first part of a bus cycle. To test for address faults, an I/O module or the probe can be synchronized to the address using these TL/1 *sync* commands:

```
sync device "/pod", mode "addr"
sync device "/mod1", mode "pod"
```

or from the operator's keyboard using the SYNC key:

```
SYNC I/O MOD 1 TO POD ADDR
```

With the above synchronization, the I/O module can capture address or other information in functional blocks related to the address. In a similar way, the probe or I/O module can be synchronized to data, or to other microprocessor-specific bus-cycle phases implemented by the pod.

Other Microprocessor Cycles

Other microprocessor cycles may be exercised as part of the microprocessor functional block, such as interrupts, bus exchanges, DMA transfers, or coprocessor cycles. Usually, however, implementation of these cycles involves circuitry that is complex enough to be partitioned separately. Here are a few considerations to keep in mind when testing:

- **Interrupts** are reported by the pod as "active interrupt lines". When a RUN UUT command is entered at the operator's keypad, control is returned to the microprocessor. The operator should be sure that the software needed to set interrupt priorities and handle interrupts is present so that RUN UUT operates properly. Some designs employ a watchdog timer, which asserts a non-maskable interrupt or reset unless the microprocessor performs a write within a certain period of time. When you use pod breakpoints, the watchdog timer should be disabled, the pod should be set up to ignore the watchdog timer output, or software should be written to handle the interrupt.

- **Bus Exchanges** take place when the microprocessor gives control of its bus to a requesting component. Pods allow this capability to be enabled or disabled. When enabled, the pod will grant bus requests just as the microprocessor would. The pod may appear to take an abnormally long time to perform certain tests, such as RAM test, if other components take control of the bus or if a fault condition causes bus requests. Disabling bus requests will command the pod not to grant the bus request and will cause bus requests to show up as forcing signal conditions. If the RUN UUT command is entered at the keypad, control is returned to the microprocessor and the bus request line will be re-enabled. Further troubleshooting with RUN UUT may require that the line be physically disabled.
- **DMA controllers** are integrated with some microprocessors, such as the 64180. The DMA channels operate semi-autonomously and interact with the bus exchange capability. Cautions similar to those used with bus exchanges should be used for DMA channels.
- **Dynamic RAM Refresh** capability is included on some microprocessors, such as the Z80 and the 64180. At regular intervals, a refresh cycle is performed and an address is placed on the address bus. The Z80 and 64180 have refresh signals (RFSH and REF, respectively) which indicate when refresh activity occurs. The frequency of these signals may be monitored with the probe to determine if refresh is working properly.
- **Coprocessors** work in conjunction with some 16- and 32-bit microprocessors. These coprocessors usually have a unique set of signals which control the transfer of data with the main processor. Inputs are called status lines and may be read and reported by the pod. Outputs are called control lines and may be written to check drivability or to send information to the coprocessor.

Other Input and Output

There are other types of inputs and outputs specific to each microprocessor which do not fall into the four basic classifications, address, data, control, and status. These are classified as miscellaneous and include signal types such as bit/parallel, serial, and analog I/O. Each pod treats these lines in a manner appropriate to the specific microprocessor. Refer to the particular pod manual for information on how to handle these signal types.

Microprocessor Bus Example 4.1.3.

The Demo/Trainer UUT uses an 80286 microprocessor, which has a 16-bit data bus and a 24-bit address bus. The Demo/Trainer UUT uses only the least significant 20 bits of the address bus.

The 80286 microprocessor remains in the UUT at all times, so the Demo/Trainer UUT includes a test access socket to provide access to the microprocessor bus. Most of the lines in the test access socket are directly connected to the microprocessor, although a few lines such as HOLD and HOLDA are buffered with three-state buffers. The test access switch, S5, selects either the 80286 microprocessor in the pod or the 80286 microprocessor on the UUT to control operation of the UUT.

Keystroke Functional Test 4.1.4.

Use the BUS TEST key to enter the following command:

```
TEST BUS AT ADDR 0
```

The above command is the entire procedure; the Microprocessor Bus functional block (Figure 4-2) can be tested fully with this single test.

The microprocessor bus test is built-in. It is convenient to run first because:

- It's easy.
- It's fast.
- It provides excellent diagnostic information.
- A bus fault will cause almost all other functional tests to fail, so it should be tested first.

The bus test uncovers all drive problems that may occur at the microprocessor socket. These faults will cause other tests to fail, but the diagnostics for bus faults are best with BUS TEST. If a fault is uncovered, a message will be displayed to the operator. See Appendix F in the *Technical User's Manual* for a list of fault messages.

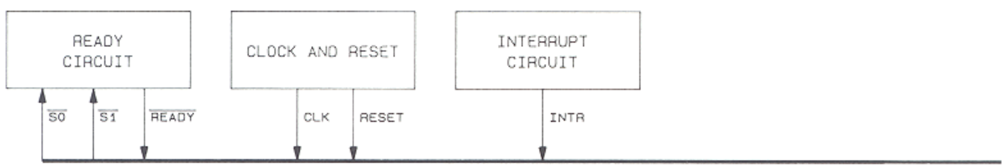
Keystroke Functional Test

CONNECTION TABLE

STIMULUS	MEASUREMENT
POD TEST ACCESS SOCKET	POD TEST ACCESS SOCKET

RESPONSE TABLE

(BUILT-IN RESPONSE MEASUREMENT)



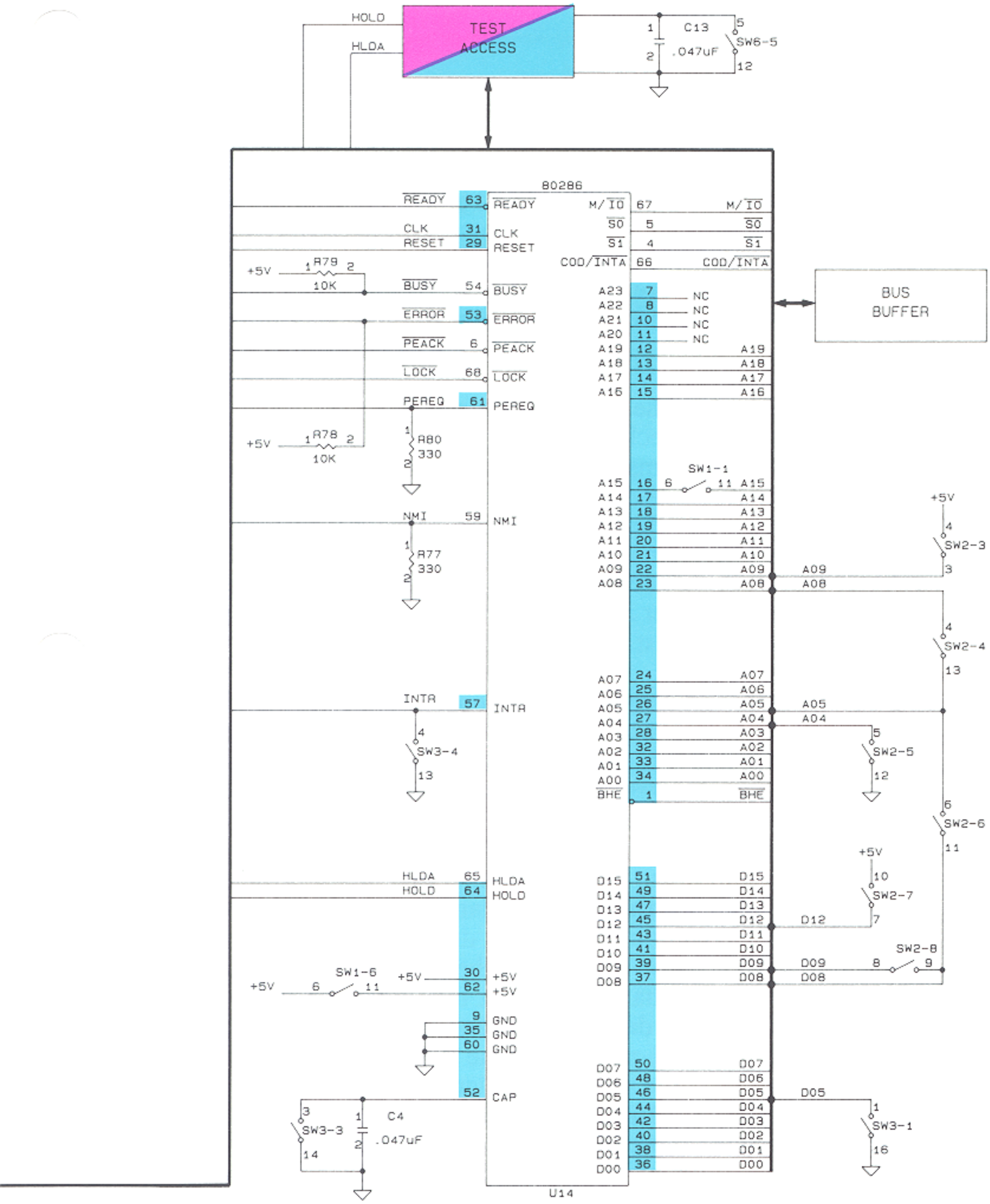


Figure 4-2: Microprocessor Bus Functional Test

Programmed Functional Test 4.1.5.

The Demo/Trainer UUT is determined to be good if functional tests for the Microprocessor Bus, ROM, RAM, Parallel I/O, Serial I/O, and Video Output functional blocks all pass. In order to make the testing as efficient as possible, the buffered bus, address decode, and ready circuitry should be exercised early in the testing. Furthermore, this testing should happen quickly, minimizing the amount of clipping of I/O modules to components.

To meet these goals, the Microprocessor Bus functional test program, *test_bus*, checks the microprocessor bus up to the buffers and also performs an access to every decoded address space (such as ROM, RAM, or Video I/O). These accesses indirectly check the Buffered Bus, Address Decode, Ready, and Interrupt Circuit functional blocks. If a ready or active interrupt problem exists, these accesses to the decoded address spaces will result in an improper ready or active interrupt condition that can be detected by the test.

The *test_bus* program also performs a check for bus contention. Bus contention occurs when a component continually outputs onto the data bus and it is usually caused by faulty enable inputs into a component. The *test_bus* program detects bus contention by reading at a spare address location, which is decoded and can be read from but has no component located at that address to output data onto the data bus. In normal operation, only high bits (logic 1s) are returned on the data bus when the spare address is read. When bus contention drives data bits low, the read at the spare address will detect the problem. In order to detect bus contention that drives data bits high, the *test_bus* program writes all-zero data to RAM and then reads the RAM. If the data read is not all-zero, either the RAM is bad or there is bus contention. To make sure the problem is bus contention, the *test_bus* program reads from two other components on the data bus that are decoded separately. The *test_bus* program uses the ROMs from bank zero and the ROMs from bank 1. If both ROM banks read zero data correctly, the problem is assumed to be a RAM problem (when bus contention occurs, most of the components on the bus will fail). When both ROM banks read zero data correctly, the *test_bus* program concludes that the problem is not bus contention and leaves further fault isolation to a later test.

If a bus contention problem is detected, a separate bus contention test program called *tst_conten* is executed (see Appendix C for a listing of this program). The *tst_conten* program tests the enable lines for each component that is connected to the data bus. All other information about good or bad data and address lines is ignored by the bus contention program.

The entire *test_bus* functional test runs quickly, but it detects most kernel faults not in the RAM or ROM components.

```
program test_bus

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the Microprocessor Bus.                             !
!                                                                        !
! This program tests the unbuffered microprocessor bus, performs an     !
! access at each decoded address of the buffered bus, and checks the    !
! data bus for bus contention (where a component outputs onto the data  !
! bus at incorrect times).   If bus contention is detected then the     !
! program TST_CONTEN is executed.  TST_CONTEN checks for incorrect      !
! enable line conditions on all the components on the buffered data bus.!
!                                                                        !
! TEST PROGRAMS CALLED:                                                  !
!    tst_conten (addr, data_bits)           Test for bus contention on  !
!                                           the data bus by checking the !
!                                           enable lines of all devices !
!                                           on the data bus.            !
!                                                                        !
! Local Constants:                                                       !
!    ZERO_AT_ROM0                           Address of zero data in ROM0 !
!    ZERO_AT_ROM1                           Address of zero data in ROM1 !
!    IO_BYTE                                I/O BYTE address specifier  !
!    MEM_WORD                               MEMORY WORD address specifier !
!                                                                        !
! Local Variables Modified:                                              !
!    x                                      value returned from a read  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  declare numeric ZERO_AT_ROM0 = $E002A !Location in ROM0 where 0 exists
  declare numeric ZERO_AT_ROM1 = $F0022 !Location in ROM1 where 0 exists
```

```
! Setup Statements

   podsetup 'enable ~ready' "on"
   podsetup 'report forcing' "on"
   IO_BYTE = getspace space "i/o", size "byte"
   MEM_WORD = getspace space "memory", size "word"

! Test the Unbuffered Microprocessor Bus.

   testbus addr 0

! Test the Extended Microprocessor Bus and Address Decoding.

   setspace (MEM_WORD)
   read addr 0                  ! RAM BANK 0
   read addr $10000             ! RAM BANK 1
   write addr $20000, data 0    ! VIDEO RAM (write only)
   read addr $30000             ! INTERRUPT POLL
   read addr $E0000             ! ROM BANK 0
   read addr $F0000             ! ROM BANK 1
   setspace (IO_BYTE)
   read addr 0                  ! VIDEO SELECT
   read addr $2000              ! RS232 SELECT
   read addr $4000              ! PIA SELECT

! Test for Bus Contention driving lines low by accessing unused address space

   setspace (MEM_WORD)
   x = read addr $50000         ! SPARE-2 ADDRESS SPACE
   if x <> $FFFF then
      execute tst_conten( $50000, cpl(x) and $FFFF)
      return
   end if


! Test for Bus Contention driving lines high by reading and writing RAM
! If failure then check for bad RAM by reading zeros from 2 other devices.

   write addr 0, data 0         ! WRITE and READ RAM addr 0
   x = read addr 0              ! If fails then check for bad RAM
   if x <> 0 then               ! by reading 0's at ROM0 and ROM1
      if (read addr ZERO_AT_ROM0) <> 0 then
         if (read addr ZERO_AT_ROM1) <> 0 then
            execute tst_conten( 0, x)
            return
         end if
      end if
   end if

end program
```

Stimulus Programs and Responses 4.1.6.

Stimulus programs are TL/1 programs that are executed by GFI for the purpose of troubleshooting faulty circuits. A stimulus program response file should be associated with each stimulus program in order to store the known-good response for each node to be stimulated by the stimulus program. In this functional block, the microprocessor is the only component and its outputs are stimulated in three groups: address lines, data lines, and control lines.

Figure 4-3 is the stimulus program planning diagram for the Microprocessor Bus functional block. It shows three stimulus programs that are used to exercise the outputs in the microprocessor functional block. These stimulus programs (and their associated response files by the same name) exercise and characterize nodes to be measured in the Microprocessor Bus functional block and in other functional blocks as well.

There are several rules for stimulus programs and response files. One is that only outputs are characterized. Another is that data must be characterized while flowing in only one direction. Therefore, the *data_out* stimulus program measures only data coming out from the microprocessor. Other stimulus programs will measure data coming in to the microprocessor.

After the stimulus program planning diagram, the stimulus programs, and the response files, there is a summary page in the form of a UUT Directory. It shows the entire set of stimulus programs, response files, and other files needed to perform testing and troubleshooting on this functional block. The summary page also shows where each of the stimulus programs and response files can be found in this manual. You will notice that each stimulus program and its associated response file (with the same name) are shown in only one location, although the pair will often be used with more than one functional block.

Stimulus Program Planning

PROGRAM: ADDR_OUT

EXERCISES ALL ADDRESS LINES FROM THE MICROPROCESSOR

MEASUREMENT AT:

U14-1
U14-34,33,32,28,27,26,25,24
U14-23,22,21,20,19,18,17,16
U14-15,14,13,12

PROGRAM: CTRL_OUT1

EXERCISES CONTROL LINES FROM THE MICROPROCESSOR USING POD ADDRESS SYNC

MEASUREMENT AT:

U14-67,5,4,66

PROGRAM: DATA_OUT

EXERCISES ALL DATA LINES AS OUTPUTS FROM THE MICROPROCESSOR

MEASUREMENT AT:

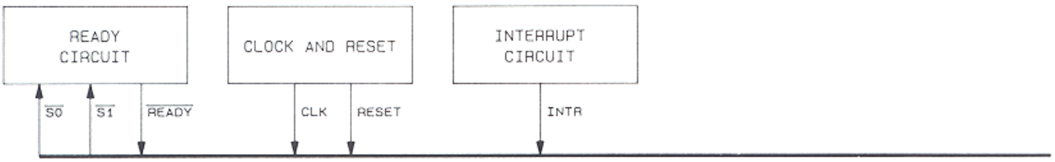
U14-36,38,40,42,44,46,48,50
U14-37,39,41,43,45,47,49,51

PROGRAM: LEVELS

MEASURES STATIC LEVELS

MEASUREMENT AT:

R77-1	C13-1
R78-2	C4-1
R79-2	U14-65
R80-1	



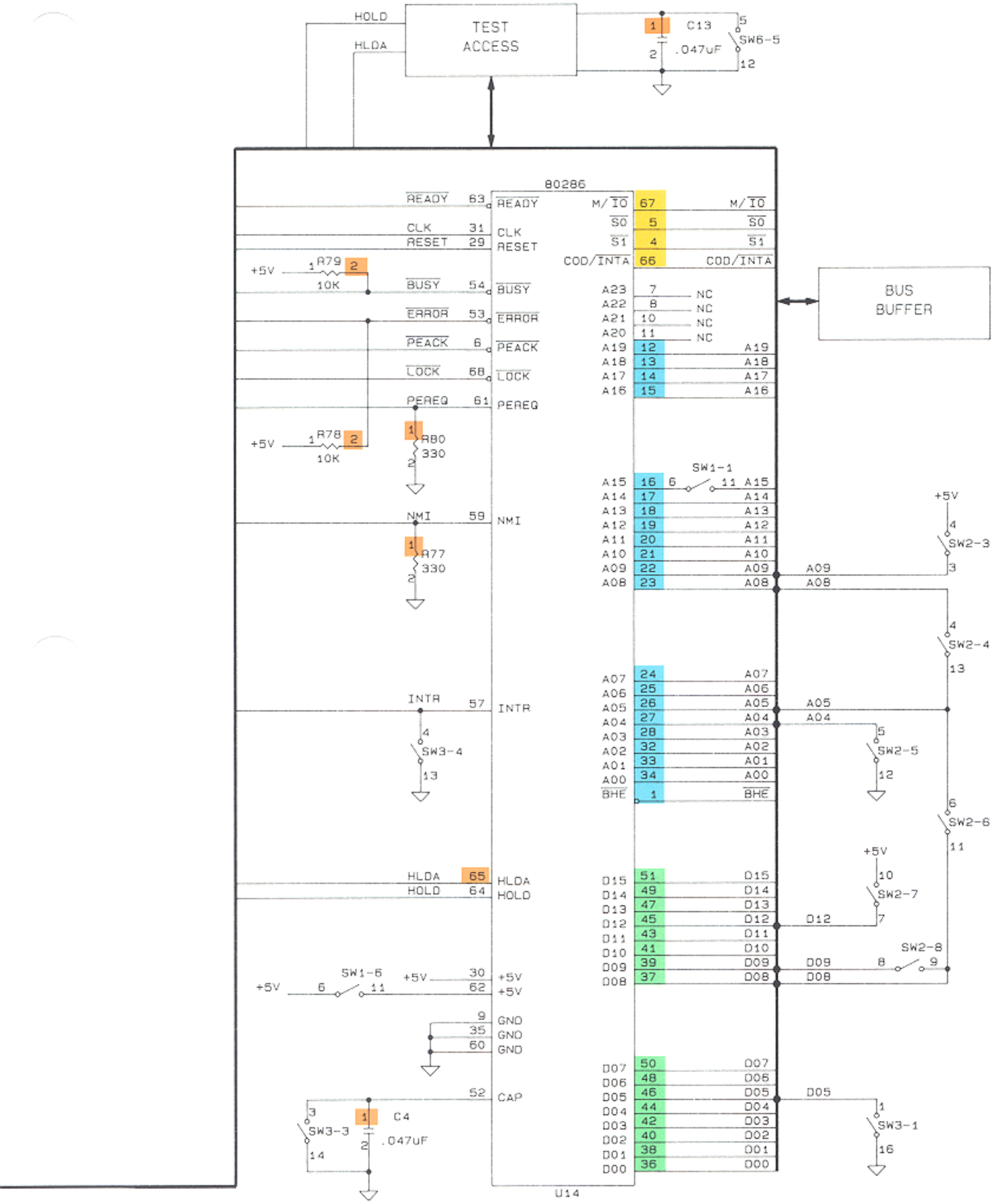


Figure 4-3: Microprocessor Bus Stimulus Program Planning

```
program addr_out

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM to wiggle all address lines from the uP.              !
!                                                                        !
! Stimulus programs and response files are used by GFI to back-trace     !
! from a failing node.  The stimulus program must create repeatable UUT  !
! activity and the response file contains the known-good responses for   !
! the outputs in the UUT that are stimulated by the stimulus program.    !
!                                                                        !
! This stimulus program is one of the programs which creates activity    !
! in the kernel area of the UUT.  These programs create activity with    !
! or without the ready circuit working properly.  Because of this, all   !
! the stimulus programs in the kernel area must disable the READY input  !
! to the pod, then perform the stimulus, then re-enable the READY input  !
! to the pod.  The 80286 microprocessor has a separate bus controller;   !
! for this reason, disabling ready and performing stimulus can get the   !
! bus controller out of synchronization with the pod.  Two fault         !
! handlers trap pod timeout conditions that indicate the bus controller  !
! is out of synchronization.  The recover() program is executed to       !
! resynchronize the bus controller and the pod.                          !
!                                                                        !
! TEST PROGRAMS CALLED:                                                  !
!    recover     ()                        The 80286 microprocessor has a!
!                                          bus controller that is totally!
!                                          separate from the pod.  In    !
!                                          some cases the pod can get out!
!                                          of sync with the bus control- !
!                                          ler.  The recover program     !
!                                          resynchronizes the pod and the!
!                                          bus controller.               !
!                                                                        !
! GRAPHICS PROGRAMS CALLED:                                              !
!    (none)                                                              !
!                                                                        !
! Local Variables Modified:                                              !
!    devname                                     Measurement device      !
!                                                                        !
! Global Variables Modified:                                             !
!    recover_times                         Reset to Zero                 !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
```

(continued on the next page)

Figure 4-4: Stimulus Program *(addr_out)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   recover_times = 0

! Let GFI determine the measurement device.

   if (gfi control) = "yes" then
       devname = gfi device
   else
       devname = "/mod1"
   end if
   print "Stimulus Program ADDR_OUT"

! Set addressing mode and setup measurement device.

   podsetup 'enable ~ready' "off"
   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   mem_byte = getspace space "memory", size "byte"
   setspace( mem_byte )
   reset device devname
   sync device devname, mode "pod"
   sync device "pod", mode "addr"

! Present stimulus to UUT.

   arm device devname              ! Start response capture.
      rampaddr addr 0, mask $1F
      rampaddr addr 0, mask $1F0
      rampaddr addr 0, mask $1F00
      rampaddr addr 0, mask $1F000
      rampdata addr $20000, data 0, mask $F0
      rampaddr addr $30000, mask $F00
      rampaddr addr $E0000, mask $1F000
   readout device devname          ! End response capture.

   podsetup 'enable ~ready' "on"
end program
```

Figure 4-4: Stimulus Program *(addr_out) - continued*

STIMULUS PROGRAM NAME: ADDR_OUT
DESCRIPTION: SIZE: 1,194 BYTES

	------------------- Response Data		-------------------			
Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
U16-19	I/O MODULE	DEB8	1	0	TRANS		
U16-16	PROBE	4A68	1	0	TRANS		
U16-16	I/O MODULE	4A68	1	0	TRANS		
U16-15	PROBE	421D	1	0	TRANS		
U16-15	I/O MODULE	421D	1	0	TRANS		
U16-12	PROBE	BFDC	1	0	TRANS		
U16-12	I/O MODULE	BFDC	1	0	TRANS		
U16-9	PROBE	113E	1	0	TRANS		
U16-9	I/O MODULE	113E	1	0	TRANS		
U16-6	I/O MODULE	8F00	1	0	TRANS		
U16-5	I/O MODULE	8300	1	0	TRANS		
U16-2	I/O MODULE	B300	1	0	TRANS		
U2-19	I/O MODULE	AED2	1	0	TRANS		
U2-16	I/O MODULE	88CD	1	0	TRANS		
U2-15	I/O MODULE	8296	1	0	TRANS		
U2-12	I/O MODULE	3B90	1	0	TRANS		
U2-9	I/O MODULE	09E8	1	0	TRANS		
U2-6	I/O MODULE	0D9C	1	0	TRANS		
U2-5	I/O MODULE	56D3	1	0	TRANS		
U2-2	I/O MODULE	9CA7	1	0	TRANS		
U14-1	PROBE	60CD	1	0	TRANS		
U14-1	I/O MODULE	60CD	1	0	TRANS		
U14-34	PROBE	DEB8	1	0	TRANS		
U14-34	I/O MODULE	DEB8	1	0	TRANS		
U14-33	PROBE	4A68	1	0	TRANS		
U14-33	I/O MODULE	4A68	1	0	TRANS		
U14-32	PROBE	421D	1	0	TRANS		
U14-32	I/O MODULE	421D	1	0	TRANS		
U14-28	PROBE	BFDC	1	0	TRANS		
U14-28	I/O MODULE	BFDC	1	0	TRANS		
U14-27	PROBE	113E	1	0	TRANS		
U14-27	I/O MODULE	113E	1	0	TRANS		
U14-26	PROBE	8F00	1	0	TRANS		
U14-26	I/O MODULE	8F00	1	0	TRANS		
U14-25	PROBE	8300	1	0	TRANS		
U14-25	I/O MODULE	8300	1	0	TRANS		
U14-24	PROBE	B300	1	0	TRANS		
U14-24	I/O MODULE	B300	1	0	TRANS		
U14-23	PROBE	AED2	1	0	TRANS		
U14-23	I/O MODULE	AED2	1	0	TRANS		
U14-22	PROBE	88CD	1	0	TRANS		
U14-22	I/O MODULE	88CD	1	0	TRANS		
U14-21	PROBE	8296	1	0	TRANS		
U14-21	I/O MODULE	8296	1	0	TRANS		

(continued on the next page)

Figure 4-5: Response File *(addr_out)*

U14-20	PROBE	3B90	1 0	TRANS
U14-20	I/O MODULE	3B90	1 0	TRANS
U14-19	PROBE	09E8	1 0	TRANS
U14-19	I/O MODULE	09E8	1 0	TRANS
U14-18	PROBE	0D9C	1 0	TRANS
U14-18	I/O MODULE	0D9C	1 0	TRANS
U14-17	PROBE	56D3	1 0	TRANS
U14-17	I/O MODULE	56D3	1 0	TRANS
U14-16	PROBE	9CA7	1 0	TRANS
U14-16	I/O MODULE	9CA7	1 0	TRANS
U14-15	PROBE	8E87	1 0	TRANS
U14-15	I/O MODULE	8E87	1 0	TRANS
U14-14	PROBE	A70C	1 0	TRANS
U14-14	I/O MODULE	A70C	1 0	TRANS
U14-13	PROBE	3951	1 0	TRANS
U14-13	I/O MODULE	3951	1 0	TRANS
U14-12	PROBE	3951	1 0	TRANS
U14-12	I/O MODULE	3951	1 0	TRANS
U22-19	I/O MODULE	8E87	1 0	TRANS
U22-16	I/O MODULE	A70C	1 0	TRANS
U22-15	I/O MODULE	3951	1 0	TRANS
U22-12	I/O MODULE	3951	1 0	TRANS
U22-9	I/O MODULE	60CD	1 0	TRANS
U57-4	I/O MODULE	8724	1 0	TRANS

Figure 4-5: Response File *(addr_out) - continued*

```
program data_out

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM for data bus buffers U3 and U23.                      !
!                                                                       !
! Stimulus programs and response files are used by GFI to backtrace     !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.   !
!                                                                       !
! This stimulus program is one of the programs which creates activity   !
! in the kernel area of the UUT.  These programs create activity with   !
! or without the ready circuit working properly.  Because of this, all  !
! the stimulus programs in the kernel area must disable the READY input !
! to the pod, then perform the stimulus, then re-enable the READY input !
! to the pod.  The 80286 microprocessor has a separate bus controller;  !
! for this reason, disabling ready and performing stimulus can get the  !
! bus controller out of synchronization with the pod.  Two fault        !
! handlers trap pod timeout conditions that indicate the bus controller !
! is out of synchronization.  The recover() program is executed to      !
! resynchronize the bus controller and the pod.                         !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    recover     ()                     The 80286 microprocessor has a!
!                                       bus controller that is totally!
!                                       separate from the pod.  In    !
!                                       some cases the pod can get out!
!                                       of sync with the bus control- !
!                                       ler.  The recover program     !
!                                       resynchronizes the pod and the!
!                                       bus controller.               !
!                                                                       !
! GRAPHICS PROGRAMS CALLED:                                             !
!    (none)                                                             !
!                                                                       !
! Local Variables Modified:                                             !
!    devname                            Measurement device              !
!                                                                       !
! Global Variables Modified:                                            !
!    recover_times                      Reset to Zero                   !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
```

(continued on the next page)

Figure 4-6: Stimulus Program *(data_out)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI determine the measurement device.

   if (gfi control) = "yes" then
      devname = gfi device
   else
      devname = "/mod1"
   end if
   print "Stimulus Program DATA_OUT"

! Set addressing mode and setup measurement device.

   podsetup 'enable ~ready' "off"
   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   setspace space (getspace space "memory", size "word")
   reset device devname
   sync device devname, mode "pod"
   sync device "/pod", mode "data"

! Present stimulus to UUT.

   arm device devname           ! Start response capture.
      rampdata addr 0, data 0, mask $FF
      rampdata addr 0, data 0, mask $FF00
   readout device devname       ! End response capture.

   podsetup 'enable ~ready' "on"
end program
```

Figure 4-6: Stimulus Program *(data_out) - continued*

```
STIMULUS PROGRAM NAME: DATA_OUT
DESCRIPTION:                                              SIZE:            982 BYTES

              ------------------- Response Data --------------------
  Node         Learned             Async Clk Counter                          Priority
Signal Src      With        SIG     LVL  LVL  Mode      Counter Range          Pin

U3-11        PROBE        AA61         1  0 TRANS
U3-11        I/O MODULE   AA61         1  0 TRANS
U3-12        PROBE        99DF         1  0 TRANS
U3-12        I/O MODULE   99DF         1  0 TRANS
U3-13        PROBE        8793         1  0 TRANS
U3-13        I/O MODULE   8793         1  0 TRANS
U3-14        PROBE        E618         1  0 TRANS
U3-14        I/O MODULE   E618         1  0 TRANS
U3-15        PROBE        8793         1  0 TRANS
U3-15        I/O MODULE   F513         1  0 TRANS
U3-16        PROBE        4FFB         1  0 TRANS
U3-16        I/O MODULE   4FFB         1  0 TRANS
U3-17        PROBE        3600         1  0 TRANS
U3-17        I/O MODULE   3600         1  0 TRANS
U3-18        PROBE        B259         1  0 TRANS
U3-18        I/O MODULE   B259         1  0 TRANS
U23-11       I/O MODULE   96EC         1  0 TRANS
U23-12       I/O MODULE   725B         1  0 TRANS
U23-13       I/O MODULE   E5ED         1  0 TRANS
U23-14       I/O MODULE   5BE0         1  0 TRANS
U23-15       I/O MODULE   7E25         1  0 TRANS
U23-16       I/O MODULE   85EA         1  0 TRANS
U23-17       I/O MODULE   77C7         1  0 TRANS
U23-18       I/O MODULE   6EBE         1  0 TRANS
U14-51       PROBE        6EBE         1  0 TRANS
U14-51       I/O MODULE   6EBE         1  0 TRANS
U14-49       PROBE        77C7         1  0 TRANS
U14-49       I/O MODULE   77C7         1  0 TRANS
U14-47       PROBE        85EA         1  0 TRANS
U14-47       I/O MODULE   85EA         1  0 TRANS
U14-45       PROBE        7E25         1  0 TRANS
U14-45       I/O MODULE   7E25         1  0 TRANS
U14-43       PROBE        5BE0         1  0 TRANS
U14-43       I/O MODULE   5BE0         1  0 TRANS
U14-41       PROBE        E5ED         1  0 TRANS
U14-41       I/O MODULE   E5ED         1  0 TRANS
U14-39       PROBE        725B         1  0 TRANS
U14-39       I/O MODULE   725B         1  0 TRANS
U14-37       PROBE        96EC         1  0 TRANS
U14-37       I/O MODULE   96EC         1  0 TRANS
U14-50       PROBE        B259         1  0 TRANS
U14-50       I/O MODULE   B259         1  0 TRANS
U14-48       PROBE        3600         1  0 TRANS
```

(continued on the next page)

Figure 4-7: Response File *(data_out)*

U14-48	I/O MODULE	3600	1	0	TRANS
U14-46	PROBE	4FFB	1	0	TRANS
U14-46	I/O MODULE	4FFB	1	0	TRANS
U14-44	PROBE	F513	1	0	TRANS
U14-44	I/O MODULE	F513	1	0	TRANS
U14-42	PROBE	E618	1	0	TRANS
U14-42	I/O MODULE	E618	1	0	TRANS
U14-40	PROBE	8793	1	0	TRANS
U14-40	I/O MODULE	8793	1	0	TRANS
U14-38	PROBE	99DF	1	0	TRANS
U14-38	I/O MODULE	99DF	1	0	TRANS
U14-36	PROBE	AA61	1	0	TRANS
U14-36	I/O MODULE	AA61	1	0	TRANS

Figure 4-7: Response File *(data_out) - continued*

```
program ctrl_out1

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM for bus controller, U15 & uP ctrl lines.             !
!                                                                       !
! Stimulus programs and response files are used by GFI to backtrace     !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.   !
!                                                                       !
! This stimulus program is one of the programs which creates activity   !
! in the kernel area of the UUT.  These programs create activity with   !
! or without the ready circuit working properly.  Because of this, all  !
! the stimulus programs in the kernel area must disable the READY input !
! to the pod, then perform the stimulus, then re-enable the READY input !
! to the pod.  The 80286 microprocessor has a separate bus controller;  !
! for this reason, disabling ready and performing stimulus can get the  !
! bus controller out of synchronization with the pod.  Two fault        !
! handlers trap pod timeout conditions that indicate the bus controller !
! is out of synchronization.  The recover() program is executed to      !
! resynchronize the bus controller and the pod.                         !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    recover     ()                        The 80286 microprocessor has a!
!                                          bus controller that is totally!
!                                          separate from the pod.  In    !
!                                          some cases the pod can get out!
!                                          of sync with the bus control- !
!                                          ler.  The recover program     !
!                                          resynchronizes the pod and the!
!                                          bus controller.               !
!                                                                       !
! GRAPHICS PROGRAMS CALLED:                                             !
!    (none)                                                             !
!                                                                       !
! Global Variables Modified:                                            !
!    recover_times                         Reset to Zero                !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
```

(continued on the next page)

Figure 4-8: Stimulus Program *(ctrl_out1)*

```
handle pod_timeout_recovered
   recover()
end handle
handle pod_timeout_no_clk
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI determine the measurement device.

   if (gfi control) = "yes" then
       devname = gfi device
   else
       devname = "/mod1"
   end if
   print "Stimulus Program CTRL_OUT1"

! Set addressing mode and setup measurement device.

   podsetup 'enable ~ready' "off"
   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   io_byte = getspace space "i/o", size "byte"
   mem_word = getspace space "memory", size "word"
   reset device devname
   sync device devname, mode "pod"
   sync device "/pod", mode "addr"
   old_cal = getoffset device devname
   setoffset device devname, offset (1000000 - 42)

! Present stimulus to UUT.

   arm device devname           ! Start response capture.
      setspace (mem_word)
      rampaddr addr $E0000, mask $1E
      rampdata addr $50000, data 0, mask $F
      setspace (io_byte)
      rampaddr addr 0, mask $5F00
      rampdata addr $2000, data 0, mask $F
   readout device devname       ! End response capture.

   setoffset device devname, offset old_cal
   podsetup 'enable ~ready' "on"
end program
```

Figure 4-8: Stimulus Program *(ctrl_out1) - continued*

STIMULUS PROGRAM NAME: CTRL_OUT1
DESCRIPTION: SIZE: 267 BYTES

Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
	Response Data						
U14-5	PROBE	5632	1 0		TRANS		
U14-5	I/O MODULE	5632	1 0		TRANS		
U14-4	PROBE	ECCF	1 0		TRANS		
U14-4	I/O MODULE	ECCF	1 0		TRANS		
U14-66	PROBE	B70D	1 0		TRANS		
U14-66	I/O MODULE	B70D	1 0		TRANS		
U14-67	PROBE	0DF0	1 0		TRANS		
U14-67	I/O MODULE	0DF0	1 0		TRANS		
U45-8	I/O MODULE	92FB	1 0		TRANS		
U15-16	I/O MODULE	2BE5	1 0		TRANS		
U57-8	I/O MODULE	9118	1 0		TRANS		
U22-5	I/O MODULE	B70D	1 0		TRANS		
U22-6	I/O MODULE	0DF0	1 0		TRANS		

Figure 4-9: Response File *(ctrl_out1)*

Summary of Complete Solution for Microprocessor Bus 4.1.7.

The entire set of programs and files needed to test and GFI troubleshoot the Microprocessor Bus functional block is shown below. The format below is similar to a 9100A/9105A UUT directory (you could consider the functional block to be a small UUT), but in addition shows the use of each program and the location in this manual for each file.

UUT DIRECTORY
(Complete File Set for Microprocessor Bus)

Programs (PROGRAM):

TEST_BUS	Functional test	Section 4.1.5
ADDR_OUT	Stimulus Program	Figure 4-4
DATA_OUT	Stimulus Program	Figure 4-6
CTRL_OUT1	Stimulus Program	Figure 4-8
LEVELS	Stimulus Program	Figure 4-92

Stimulus Program Responses (RESPONSE):

ADDR_OUT	Figure 4-5
DATA_OUT	Figure 4-7
CTRL_OUT1	Figure 4-9
LEVELS	Figure 4-93

Node List (NODE):

NODELIST	Appendix B

Text Files (TEXT):

Reference Designator List (REF):

REFLIST	Appendix A

Compiled Database (DATABASE):

GFIDATA	Compiled by the 9100A

(This page is intentionally blank.)

ROM FUNCTIONAL BLOCK 4.2.

Introduction to ROM 4.2.1.

The typical ROM block consists of the ROMs, an address path from the microprocessor to the ROMs, a data path from the ROMs to the microprocessor, and a ROM-select scheme. There are often hardware buffers separating the address and data paths from the microprocessor and ROMs; your UUT may or may not include these buffers. A simplified diagram of a typical ROM functional block is shown in Figure 4-10.

Figure 4-10 shows the microprocessor's Read/Write strobe as an input to the ROM-select circuitry. Many UUTs use the Read/Write strobe to make sure the ROM is selected only during Read cycles. This prevents potential data-bus contention between the ROM and the microprocessor during erroneous Write cycles to the ROM's address space.

Considerations for Testing and Troubleshooting 4.2.2.

Testing ROM

To test ROM thoroughly, every data bit read from the ROM (*i.e.*, every cell in the ROM) must be verified. Of course, you could compare the contents of every location with known-good contents, but this would be slow and would require that the 9100A/9105A store the known-good contents of all ROM chips. In practice, it is easier and faster to read every ROM address, compress the data into a CRC signature, and compare this signature with the signature from a known-good UUT.

The 9100A/9105A's built-in ROM test performs the operation described above. The test is first used to capture the signature

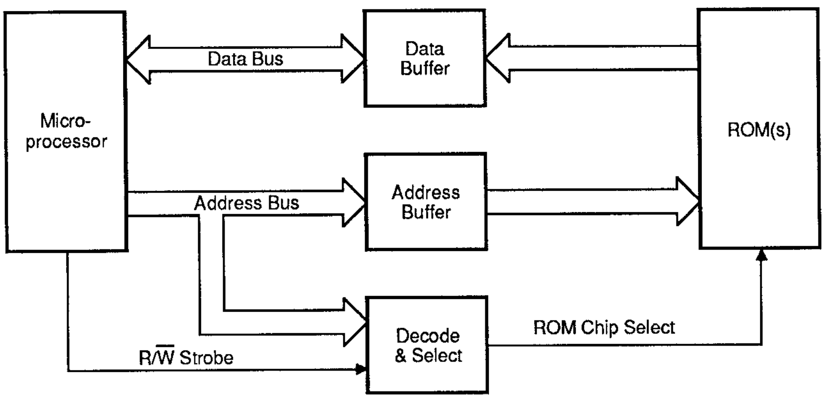


Figure 4-10: Typical ROM Block

response of a known-good UUT. Then, the test can be performed on a suspect UUT.

Refer to Section 6.2.3 of the *Technical User's Manual* for more information about the built-in ROM test.

ROM-Test Diagnostic Messages and Troubleshooting Techniques

If the built-in ROM test finds a fault, one of several diagnostic messages will be displayed. Figure 4-11 summarizes the types of conditions reported, with example messages. Here are some details about the various types of messages:

Incorrect Signature

This means that the ROM test could not identify the data or address lines at fault. It may indicate that the ROM chip itself is bad or that the wrong ROM chip is inserted. However, it could also indicate faulty ROM-select circuitry, especially if the circuitry allows ROM to be selected over only part of the proper address range. This type of fault would allow the test to read enough addresses to generate a signature, albeit an incorrect one. Here are some troubleshooting tips for this situation:

- Check that the correct ROM chip is plugged in.
- Perform the test on a known-good UUT with an I/O module clipped over the ROM chip. Write down the signatures of the individual lines from the I/O module.
- Perform the test on the suspect UUT, again with the I/O module clipped over the ROM chip.
- Compare the signatures for the individual lines. Trace any faulty inputs back toward the microprocessor, giving priority to tracing faults in chip-select lines and then in address lines.

Signal Group	*Fault*	*Example Message*
ROM Chip	bad data cells	read incorrect sig XXXX expected YYYY
ROM-Select Lines	open or stuck	read incorrect sig XXXX expected YYYY
		all ROM data bits stuck low
		all ROM data bits stuck high
Data Lines	open or stuck	data line <name> stuck high
		data line <name>stuck low
	tied	data line <name> tied to data line <name>
Address Lines	open or stuck	address line <name>stuck
	tied	address line <name>tied to address line <name>
Undetermined Fault		read incorrect sig XXXX expected YYYY

Figure 4-11: Conditions Reported by ROM Test

All Data Bits Stuck High or Low

This means that the ROM test found all ones or all zeroes on every data line throughout the test. Most probably, it means that the ROM chip is not being properly selected, that the ROM chip is missing (or unprogrammed), or that an intervening bus buffer is faulty.

To troubleshoot these faults, first check that the ROM chip is present and that it is the right part. If so, you can then trace the ROM-select path back to the microprocessor. Use a 9100A/9105A read operation on the address at which the failure occurred as a stimulus for the probe or I/O module. If the ROM-select path is good, verify that the address and data buffers are good.

Data or Address Line Stuck High, Stuck Low, or Tied

When an individual address or data line is at fault, use the probe to trace from the ROM socket back to the microprocessor and compare each node response with the known-good response.

If the faulty line is an address line, synchronize the probe to address and stimulate the line with the STIM key using the TOGGLE ADDR command on the operator's keypad. Use the LOOP key while probing to verify both low and high levels at each point on the address line until the fault is isolated.

If the faulty line is a data line, synchronize the probe to data, run a ROM test and press the LOOP key to repeat the ROM test while probing. Again, look for both low and high levels until the fault is isolated.

Additional Considerations

Here are some additional suggestions to consider when testing and troubleshooting ROM:

- **Multiple ROM Chips:** If you have more than one ROM chip on your UUT, test each chip separately. This will speed the troubleshooting process if a fault is found.

  If there is more than one ROM chip on the same data bus (or, in systems wider than 8 bits, on the same portion of the data bus) be careful that an erroneously enabled output buffer of one ROM is not corrupting the test results for another ROM. For example, consider an 8-bit microprocessor system with two ROM chips, A and B, in which chip A's output-enable input pin is tied low (a fault). Chip A will pass its ROM test, because the data in the ROM can still be read with the output-enable line tied low. ROM chip B, however, will fail its test with an incorrect-signature fault, even though there are no faults directly associated with chip B. When chip B is read by the test, the fault on chip A causes both ROMs to contend for the data bus, resulting in an incorrect signature. See the microprocessor bus functional block for suggestions on how to check for bus contention.

- **Unprogrammed ROM:** Be sure that the ROM being tested has been programmed. An unprogrammed ROM may result in an "all ROM data bits stuck high" or an "all ROM data bits stuck low" message during a ROM test.

- **Data Tied to Address:** If a ROM test results in a bad signature, it is a good idea to make sure that a data line is not tied to an address line. You can do so by clipping an I/O module to the ROM chip that produced the incorrect signature.

  If address line or data line failures are identified by a ROM test but not by a BUS test, the fault is on the ROM side of the address or data buffers.

- **Proper Sync Mode:** Generally, the data sync mode should be used to trace back faults in the ROM-select path, even though the ROM-select signal may be created from address lines. This is because the ROM-select signal should normally be asserted at the time the microprocessor reads in data from the ROM. This is also normally the situation for probing the address signals at the ROM socket.

ROM Example 4.2.3.

The operating system code for the Demo/Trainer UUT is stored in four 32K x 8 EPROMs, U27, U28, U29, and U30, shown in Figure 4-3. Since a 16-bit system is used, ROM is organized as 64K x 16 bits. The ROM0 bank covers the even addresses E0000 through EFFFE and is contained in U29 and U30. The ROM1 bank covers the even addresses F0000 through FFFFE and is contained in U27 and U28. Both banks can only be accessed in 16-bit mode. IA01 is connected to A0 on the ROMs, and the least significant address bit, IA00, is not connected to ROM. IA00 is always low in word accesses. A20-A23 are not used. At reset, 80286 code execution begins at the reset address (FFFFF0). ROM accesses do not require wait states.

Keystroke Functional Test 4.2.4.

Use the ROM TEST key to enter the following commands, and compare the measured signature with the response table in Figure 4-12.

```
GET SIG ROM REF U29 ADDR E0000 UPTO EFFFE ...
... DATA MASK FF ADDR STEP 2
... (ADDR OPTION: MEMORY WORD)
```

The measured signature (shown on the operator's display) should be 8E6E.

```
GET SIG ROM REF U30 ADDR E0000 UPTO EFFFE ...
... DATA FF00 ADDRSTEP 2
... (ADDR OPTION: MEMORY WORD)
```

The measured signature (shown on the operator's display) should be F387.

```
GET SIG ROM REF U27 ADDR F0000 UPTO FFFFE ...
... DATA FF ADDRSTEP 2
... (ADDR OPTION: MEMORY WORD)
```

The measured signature (shown on the operator's display) should be F387.

```
GET SIG ROM REF U28 ADDR F0000 UPTO FFFFE ...
... DATA FF00 ADDRSTEP 2
... (ADDR OPTION: MEMORY WORD)
```

The measured signature (shown on the operator's display) should be 8E6E.

(This page is intentionally blank.)

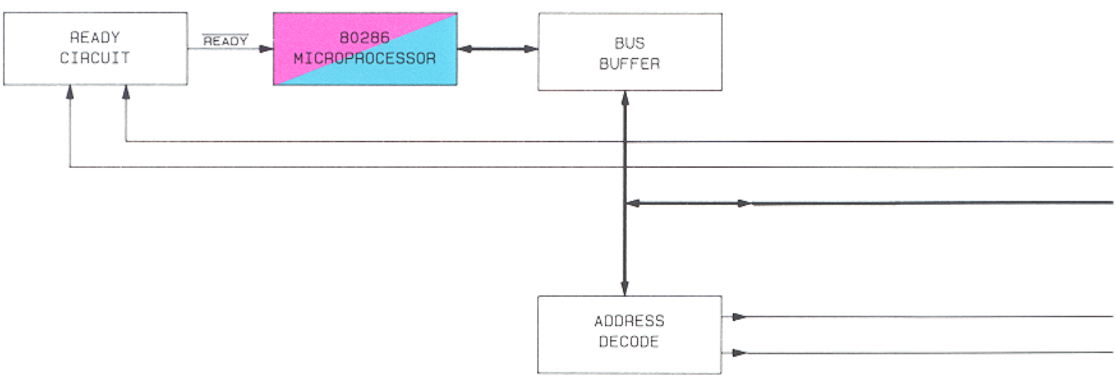
Keystroke Functional Test

CONNECTION TABLE

STIMULUS	MEASUREMENT
POD TEST ACCESS SOCKET	POD TEST ACCESS SOCKET

RESPONSE

ROM CHIP	ROM SIGNATURE
U29	8E6E
U30	F387
U27	F387
U28	8E6E



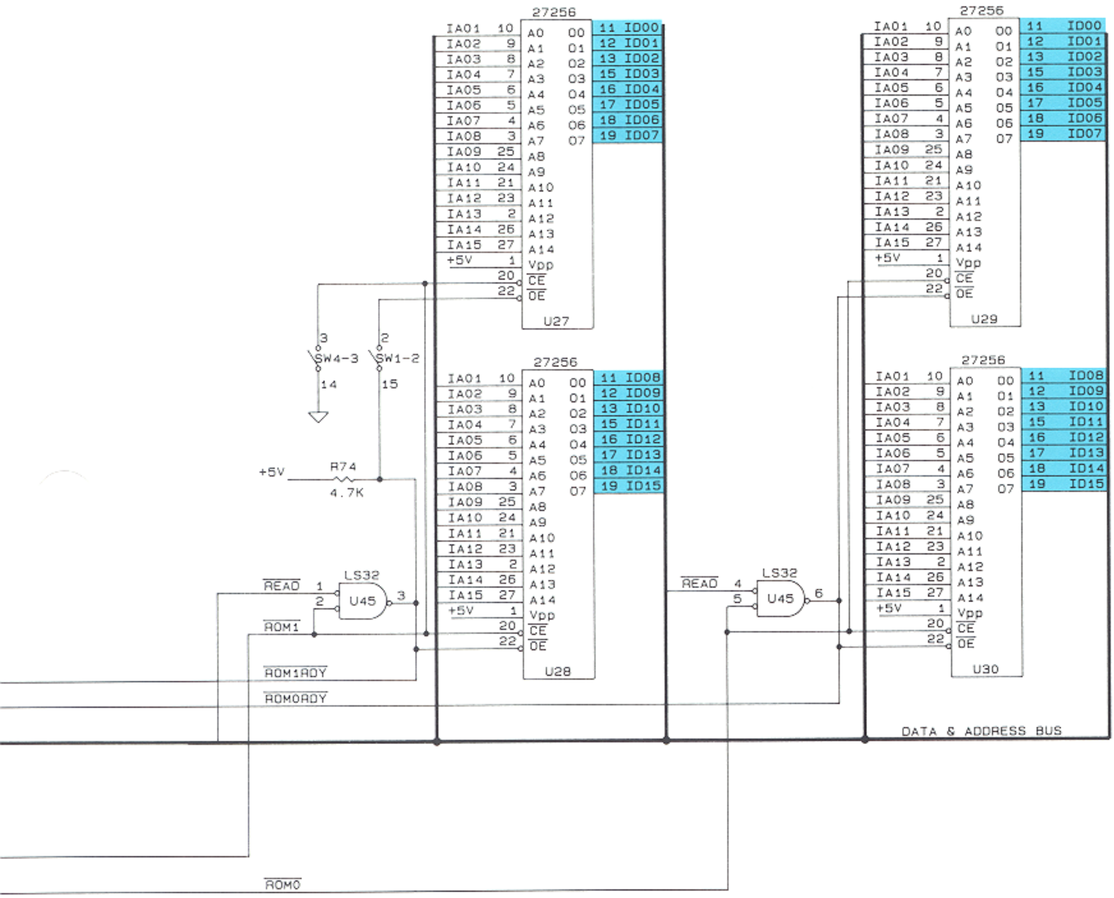


Figure 4-12: ROM Functional Test

Programmed Functional Test 4.2.5.

The *test_rom* program is the programmed functional test for the ROM functional block. It uses the *testromful* command to test the ROMs. This command will generate one of seven built-in fault conditions if *testromful* fails. The *test_rom* program then handles all seven built-in fault conditions and categorizes them into one of two new fault conditions called *rom_comp* for a component failure or *rom_address* for an address failure. The seven built-in *testramfull* faults are redirected as follows:

New Fault Condition	*Built-in Fault Condition*
rom_comp	rom_sig_incorrect rom_data_high_tied_all rom_data_low_tied_all rom_data_fault rom_data_data_tied
rom_address	rom_addr_addr_tied rom_addr_fault

The new fault condition *rom_comp* uses the *gfi test* command to clip the I/O module onto the ROMs and to test all inputs and outputs of a ROM. If a failure is detected, the test passes control to GFI. GFI backtraces to find the circuit problem that is causing the failure.

The new fault condition *rom_address* checks the address bus, and if a failure is detected control is passed to GFI. GFI then backtraces to the circuit problem which is causing the failure.

```
program test_rom

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the ROM functional block.                          !
!                                                                       !
! This program tests the ROM functional block of the Demo/Trainer.  The !
! TL/1 testromfull command is used to test the ROMs.  If the ROMs are   !
! found to be faulty, then one of seven built-in fault conditions is    !
! generated.                                                            !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Setup.

   podsetup 'enable ~ready' "on"
   podsetup 'report forcing' "on"
   setspace space (getspace space "memory", size "word")

! Main part of Test.

   testromfull addr $F0000, upto $FFFFE, addrstep 2, sig $156F
   testromfull addr $E0000, upto $EFFFE, addrstep 2, sig $B61E

end program
```

Stimulus Programs and Responses 4.2.6.

Figure 4-13 is the stimulus program planning diagram for the ROM functional block. The outputs in the ROM functional block are the outputs of U45 and the outputs of the ROM chips onto the data bus.

The stimulus programs to exercise these outputs are *rom0_data* (which reads data from U29 and U30), *rom1_data* (which reads data from U27 and U28), and *decode* (which accesses each decoded address space in the Demo/Trainer UUT).

One of the rules for stimulus programs is that when dealing with a data bus, every component that is decoded separately to output onto the data bus must have a separate stimulus program to read data from that component. For this reason, two stimulus programs are required: *rom0_data* and *rom1_data*.

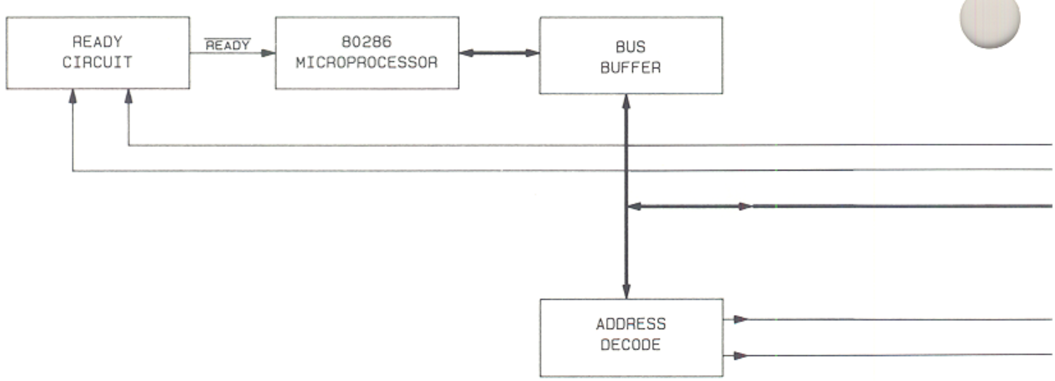
(This page is intentionally blank.)

Stimulus Program Planning

PROGRAM: ROM0_DATA
READS FIRST 2K OF DATA FROM ROMs U29 AND U30
MEASUREMENT AT: U29-11,12,13,15,16,17,18,19 U30-11,12,13,15,16,17,18,19

PROGRAM: DECODE
PERFORMS AN ACCESS FOR EACH DECODED BLOCK
MEASUREMENT AT: U45-3,6

PROGRAM: ROM1_DATA
READS FIRST 2K OF DATA FROM ROMs U27 AND U28
MEASUREMENT AT: U27-11,12,13,15,16,17,18,19 U28-11,12,13,15,16,17,18,19



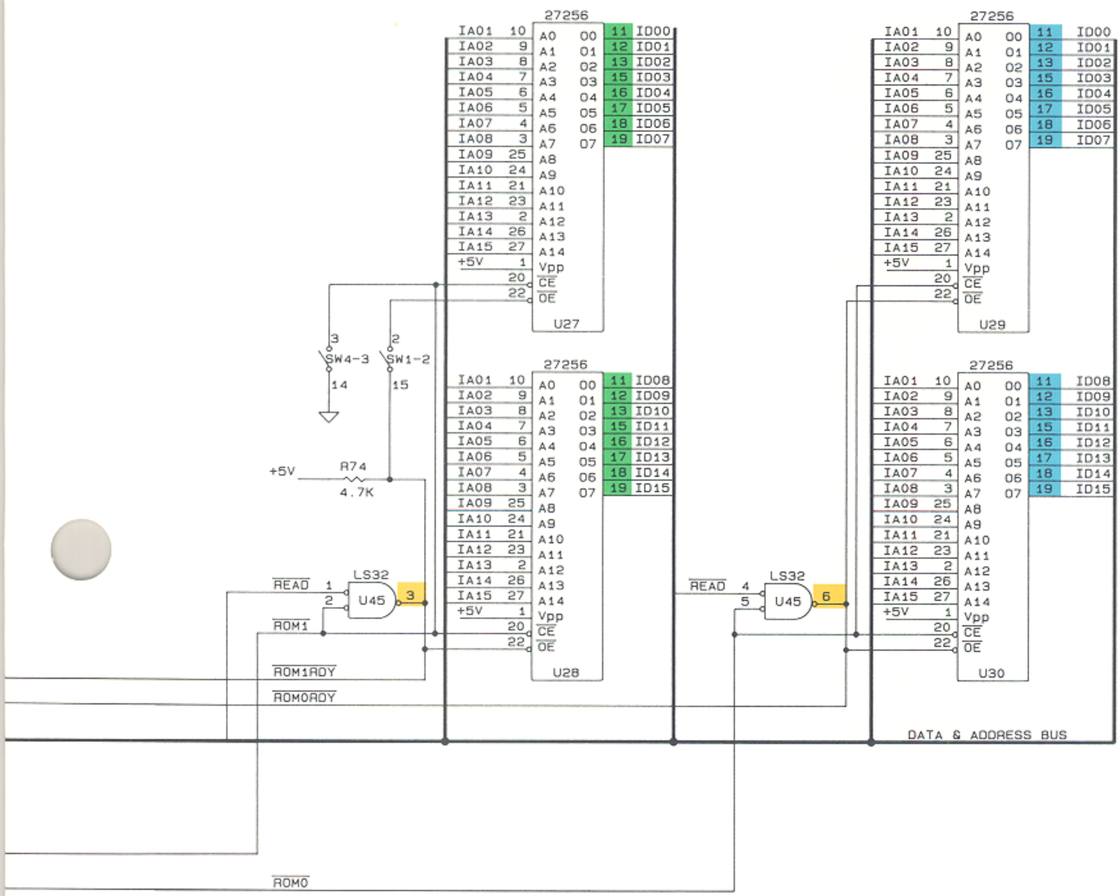


Figure 4-13: ROM Stimulus Program Planning

```
program rom0_data

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM to exercise data out of ROMs U29 and U30.             !
!                                                                        !
! Stimulus programs and response files are used by GFI to backtrace      !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.   !
!                                                                        !
! This stimulus program is one of the programs which creates activity   !
! in the kernel area of the UUT.  These programs create activity with   !
! or without the ready circuit working properly.  Because of this, all  !
! the stimulus programs in the kernel area must disable the READY input !
! to the pod, then perform the stimulus, then re-enable the READY input !
! to the pod.  The 80286 microprocessor has a separate bus controller;  !
! for this reason, disabling ready and performing stimulus can get the  !
! bus controller out of synchronization with the pod.  Two fault        !
! handlers trap pod timeout conditions that indicate the bus controller !
! is out of synchronization.  The recover() program is executed to      !
! resynchronize the bus controller and the pod.                          !
!                                                                        !
! TEST PROGRAMS CALLED:                                                  !
!    recover     ()                        The 80286 microprocessor has a!
!                                          bus controller that is totally!
!                                          separate from the pod.  In    !
!                                          some cases the pod can get out!
!                                          of sync with the bus control- !
!                                          ler.  The recover program     !
!                                          resynchronizes the pod and the!
!                                          bus controller.               !
!                                                                        !
! GRAPHICS PROGRAMS CALLED:                                              !
!    (none)                                                              !
!                                                                        !
! Global Variables Modified:                                             !
!    recover_times                         Reset to Zero                 !
!    devname                               Measurement device            !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
```

(continued on the next page)

Figure 4-14: Stimulus Program *(rom0_data)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                         !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI user select which I/O module to use

  if (gfi control) = "yes" then
    devname = gfi device
  else
    devname = "/mod1"
  end if
   print "Stimulus Program ROM0_DATA"

! Set desired measurement modes

  setspace space (getspace space "memory", size "word")
  reset device devname
  sync device devname, mode "pod"
  sync device "/pod", mode "data"

! Present stimulus to the UUT

  arm device devname                   ! Start response capture.
     rampaddr addr $E0000, mask $1FE
  readout device devname               ! End response capture

end rom0_data
```

Figure 4-14: Stimulus Program *(rom0_data) - continued*

STIMULUS PROGRAM NAME: ROM0_DATA
DESCRIPTION: SIZE: 454 BYTES

	------------------- Response Data	--------------------					
Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
U29-11	PROBE	45DD		1 0	TRANS		
U29-11	I/O MODULE	45DD		1 0	TRANS		
U29-12	PROBE	CF83		1 0	TRANS		
U29-12	I/O MODULE	CF83		1 0	TRANS		
U29-13	PROBE	BD79		1 0	TRANS		
U29-13	I/O MODULE	BD79		1 0	TRANS		
U29-15	PROBE	8A76		1 0	TRANS		
U29-15	I/O MODULE	8A76		1 0	TRANS		
U29-16	PROBE	66F3		1 0	TRANS		
U29-16	I/O MODULE	66F3		1 0	TRANS		
U29-17	PROBE	FAB5		1 0	TRANS		
U29-17	I/O MODULE	FAB5		1 0	TRANS		
U29-18	PROBE	534E		1 0	TRANS		
U29-18	I/O MODULE	534E		1 0	TRANS		
U29-19	PROBE	8D0A		1 0	TRANS		
U29-19	I/O MODULE	8D0A		1 0	TRANS		
U30-11	I/O MODULE	73E9		1 0	TRANS		
U30-12	I/O MODULE	AC84		1 0	TRANS		
U30-13	I/O MODULE	50BB		1 0	TRANS		
U30-15	I/O MODULE	5B3B		1 0	TRANS		
U30-16	I/O MODULE	06EF		1 0	TRANS		
U30-17	I/O MODULE	00A0		1 0	TRANS		
U30-18	I/O MODULE	6BF0		1 0	TRANS		
U30-19	I/O MODULE	52EE		1 0	TRANS		

Figure 4-15: Response File *(rom0_data)*

```
program rom1_data

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM to exercise data out of ROMs U29 and U30.              !
!                                                                         !
! Stimulus programs and response files are used by GFI to back-trace      !
! from a failing node.  The stimulus program must create repeatable UUT   !
! activity and the response file contains the known-good responses for    !
! the outputs in the UUT that are stimulated by the stimulus program.     !
!                                                                         !
! This stimulus program is one of the programs which creates activity     !
! in the kernel area of the UUT.  These programs create activity with     !
! or without the ready circuit working properly.  Because of this, all    !
! the stimulus programs in the kernel area must disable the READY input   !
! to the pod, then perform the stimulus, then re-enable the READY input   !
! to the pod.  The 80286 microprocessor has a separate bus controller;    !
! for this reason, disabling ready and performing stimulus can get the    !
! bus controller out of synchronization with the pod.  Two fault          !
! handlers trap pod timeout conditions that indicate the bus controller   !
! is out of synchronization.  The recover() program is executed to        !
! resynchronize the bus controller and the pod.                           !
!                                                                         !
! TEST PROGRAMS CALLED:                                                   !
!    recover     ()                        The 80286 microprocessor has a!
!                                          bus controller that is totaly !
!                                          separate from the pod.  In    !
!                                          some cases the pod can get out!
!                                          of sync with the bus control- !
!                                          ler.  The recover program     !
!                                          resynchronizes the pod and the!
!                                          bus controller.               !
!                                                                         !
! GRAPHICS PROGRAMS CALLED:                                               !
!    (none)                                                               !
!                                                                         !
! Global Variables Modified:                                              !
!    recover_times                         Reset to Zero                  !
!    devname                               Measurement device             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
```

(continued on the next page)

Figure 4-16: Stimulus Program *(rom1_data)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  FAULT HANDLERS:                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Main part of STIMULUS PROGRAM                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI user select which I/O module to use

   if (gfi control) = "yes" then
      devname = gfi device
   else
      devname = "/mod1"
   end if
   print "Stimulus Program ROM1_DATA"

! Set desired measurement modes

   setspace space (getspace space "memory", size "word")
   reset device devname
   sync device devname, mode "pod"
   sync device "/pod", mode "data"

! Present stimulus to the UUT

   arm device devname                  ! Start response capture.
      rampaddr addr $F0000, mask $1FE
   readout device devname              ! End response capture

end program
```

Figure 4-16: Stimulus Program *(rom1_data) - continued*

STIMULUS PROGRAM NAME: ROM1_DATA
DESCRIPTION: SIZE: 982 BYTES

Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
U27-11	PROBE	73E9	1	0	TRANS		
U27-11	I/O MODULE	73E9	1	0	TRANS		
U27-12	PROBE	AC84	1	0	TRANS		
U27-12	I/O MODULE	AC84	1	0	TRANS		
U27-13	PROBE	50BB	1	0	TRANS		
U27-13	I/O MODULE	50BB	1	0	TRANS		
U27-15	PROBE	5B3B	1	0	TRANS		
U27-15	I/O MODULE	5B3B	1	0	TRANS		
U27-16	PROBE	06EF	1	0	TRANS		
U27-16	I/O MODULE	06EF	1	0	TRANS		
U27-17	PROBE	00A0	1	0	TRANS		
U27-17	I/O MODULE	00A0	1	0	TRANS		
U27-18	PROBE	6BF0	1	0	TRANS		
U27-18	I/O MODULE	6BF0	1	0	TRANS		
U27-19	PROBE	52EE	1	0	TRANS		
U27-19	I/O MODULE	52EE	1	0	TRANS		
U28-11	I/O MODULE	45DD	1	0	TRANS		
U28-12	I/O MODULE	CF83	1	0	TRANS		
U28-13	I/O MODULE	BD79	1	0	TRANS		
U28-15	I/O MODULE	8A76	1	0	TRANS		
U28-16	I/O MODULE	66F3	1	0	TRANS		
U28-17	I/O MODULE	FAB5	1	0	TRANS		
U28-18	I/O MODULE	534E	1	0	TRANS		
U28-19	I/O MODULE	8D0A	1	0	TRANS		
U23-2	PROBE	52EE	1	0	TRANS		
U23-2	I/O MODULE	52EE	1	0	TRANS		

(continued on the next page)

Figure 4-17: Response File *(rom1_data)*

U23-3	PROBE	6BF0	1	0	TRANS
U23-3	I/O MODULE	6BF0	1	0	TRANS
U23-4	PROBE	00A0	1	0	TRANS
U23-4	I/O MODULE	00A0	1	0	TRANS
U23-5	PROBE	06EF	1	0	TRANS
U23-5	I/O MODULE	06EF	1	0	TRANS
U23-6	PROBE	5B3B	1	0	TRANS
U23-6	I/O MODULE	5B3B	1	0	TRANS
U23-7	PROBE	50BB	1	0	TRANS
U23-7	I/O MODULE	50BB	1	0	TRANS
U23-8	PROBE	AC84	1	0	TRANS
U23-8	I/O MODULE	AC84	1	0	TRANS
U23-9	PROBE	73E9	1	0	TRANS
U23-9	I/O MODULE	73E9	1	0	TRANS
U3-2	PROBE	8D0A	1	0	TRANS
U3-2	I/O MODULE	8D0A	1	0	TRANS
U3-3	PROBE	534E	1	0	TRANS
U3-3	I/O MODULE	534E	1	0	TRANS
U3-4	PROBE	FAB5	1	0	TRANS
U3-4	I/O MODULE	FAB5	1	0	TRANS
U3-5	PROBE	66F3	1	0	TRANS
U3-5	I/O MODULE	66F3	1	0	TRANS
U3-6	PROBE	8A76	1	0	TRANS
U3-6	I/O MODULE	8A76	1	0	TRANS
U3-7	PROBE	BD79	1	0	TRANS
U3-7	I/O MODULE	BD79	1	0	TRANS
U3-8	PROBE	CF83	1	0	TRANS
U3-8	I/O MODULE	CF83	1	0	TRANS
U3-9	PROBE	45DD	1	0	TRANS
U3-9	I/O MODULE	45DD	1	0	TRANS

Figure 4-17: Response File *(rom1_data) - continued*

Summary of Complete Solution for ROM 4.2.7.

The entire set of programs and files needed to test and GFI troubleshoot the ROM functional block is shown below. The format below is similar to a 9100A/9105A UUT directory (you could consider the functional block to be a small UUT), but in addition shows the use of each program and the location in this manual for each file.

UUT DIRECTORY
(Complete File Set for ROM)

Programs (PROGRAM):

TEST_ROM	Functional Test	Section 4.2.5
ROM0_DATA	Stimulus Program	Figure 4-14
ROM1_DATA	Stimulus Program	Figure 4-16
DECODE	Stimulus Program	Figure 4-108

Stimulus Program Responses (RESPONSE):

ROM0_DATA	Figure 4-15
ROM1_DATA	Figure 4-17
DECODE	Figure 4-109

Node List (NODE):

NODELIST	Appendix B

Text Files (TEXT):

Reference Designator List (REF):

REFLIST	Appendix A

Compiled Database (DATABASE):

GFIDATA	Compiled by the 9100A

(This page is intentionally blank.)

RAM FUNCTIONAL BLOCK 4.3.

Introduction to RAM 4.3.1.

The typical RAM block consists of the RAM chips, an address path from the microprocessor to the RAMs, a bidirectional data path between the microprocessor and the RAMs, and RAM-select circuitry. There are often hardware buffers between the microprocessor and the RAM chips.

There are two basic types of RAM: static and dynamic. Static RAM chips are faster and require no refresh circuitry. They are also more expensive and take more room for a given memory size. Dynamic RAM chips use a capacitor for charge storage and therefore must be periodically refreshed to maintain data storage. However, dynamic RAM chips provide more memory for a given size chip.

A simplified diagram of a typical RAM functional block is shown in Figure 4-18.

Considerations for Testing and Troubleshooting 4.3.2.

Speed and accuracy are the most critical factors in RAM testing, and RAM tests are typically a compromise between these two factors. To further complicate the issue, different hardware configurations bring with them different failure mechanisms which may require specialized testing.

The built-in RAM tests offers a number of choices to better match the test to the testing needs. While the RAM FULL, RAM FAST and pod-dependent RAM QUICK tests directly address the speed and accuracy compromise, they are different from each other.

Section 5 of the *Technical User's Manual* describes the various RAM tests in detail.

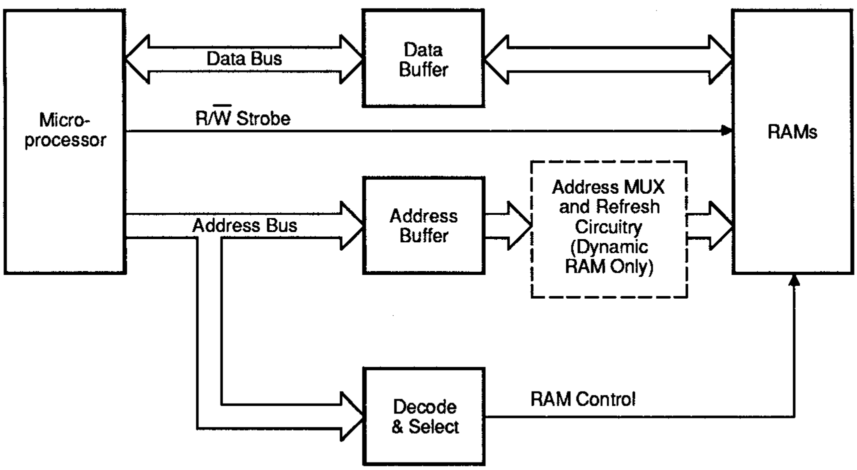


Figure 4-18: Typical RAM Block

Many types of faults can occur in RAM functional blocks. Address lines or data lines can be stuck or tied to other lines. Individual memory cells can be stuck low or high, or cells can be aliased (they respond to more than one address). Transition faults can exist (where a cell can change from one state to another, but not back again). Coupling faults can cause the contents of one cell to be disturbed when the contents of another cell is changed. If this coupling depends on the contents of several neighboring cells, the fault is called a pattern sensitive fault. Chip select address decoding logic can be faulty. Row or column decoders might not select when they should or they might select when they shouldn't. In dynamic memory, refresh logic can fail, causing cells to lose their contents.

Although failure mechanisms are different between dynamic and static RAM, both types of RAM may be functionally tested with exactly the same built-in RAM tests; only the delay parameter is of unique concern for dynamic RAM. The delay parameter provides a means of testing the refresh circuitry by specifying the number of milliseconds to wait for refresh-related faults to occur.

The first step in troubleshooting RAM is to run a built-in functional test. Besides confirming a RAM fault, the functional test often provides excellent clues for where to begin fault isolation. Figure 4-19 illustrates typical fault information provided by the RAM tests.

In general, the following procedure will work for troubleshooting any RAM faults discovered by the 9100A/9105A:

1. Create a combination of reads and writes to confirm the failure.
2. Synchronize the probe as needed.
3. Perform looping reads and writes while tracing with synchronized probe.

Fault Condition	*TEST RAM FAST*	*TEST RAM FULL* coupling enabled	*TEST RAM FULL* coupling disabled
Stuck cells	always found	always found	always found
Aliased cells	"	"	"
Stuck address lines	"	"	"
Stuck data lines	"	"	"
Shorted address lines	"	"	"
Multiple selection decoder	may be found	always found	always found
Dynamic coupling	"	"	"
Shorted data lines	may be found	always found	may be found
Aliasing between bits in same word	"	"	"
Pattern-sensitive faults	not found	not found	not found
Refresh problems	always found, if delay is sufficiently long and standby reads do not mask the problem.		

Figure 4-19: RAM Test Failure Information

RAM Example 4.3.3.

The Demo/Trainer UUT, Figure 4-20, uses 128K bytes of dynamic RAM, organized as 64k x 16 bits, and composed of sixteen 1-bit wide 4164 chips.

Keystroke Functional Test 4.3.4.

Use the RAM TEST key to enter the following command:

```
TEST RAM FAST ADDR 0 UPTO 1FFFE DATA MASK ...
... FFFF ADDR STEP 2 DELAY 250 SEED 0
... (ADDR OPTION: MEMORY WORD)
```

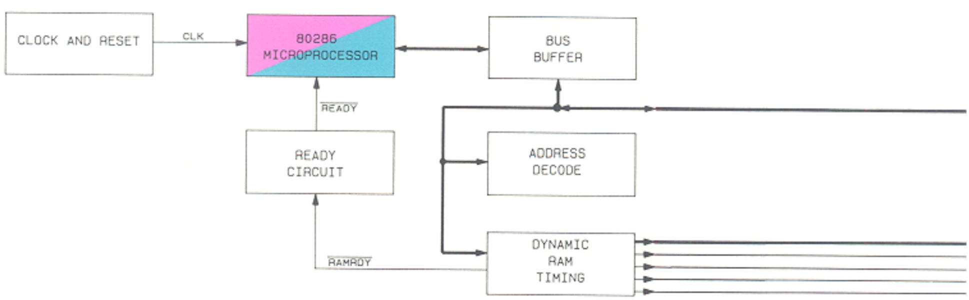
Keystroke Functional Test

CONNECTION TABLE

STIMULUS	MEASUREMENT
POD TEST ACCESS SOCKET	POD TEST ACCESS SOCKET

RESPONSE

(BUILT-IN RESPONSE MEASUREMENT)



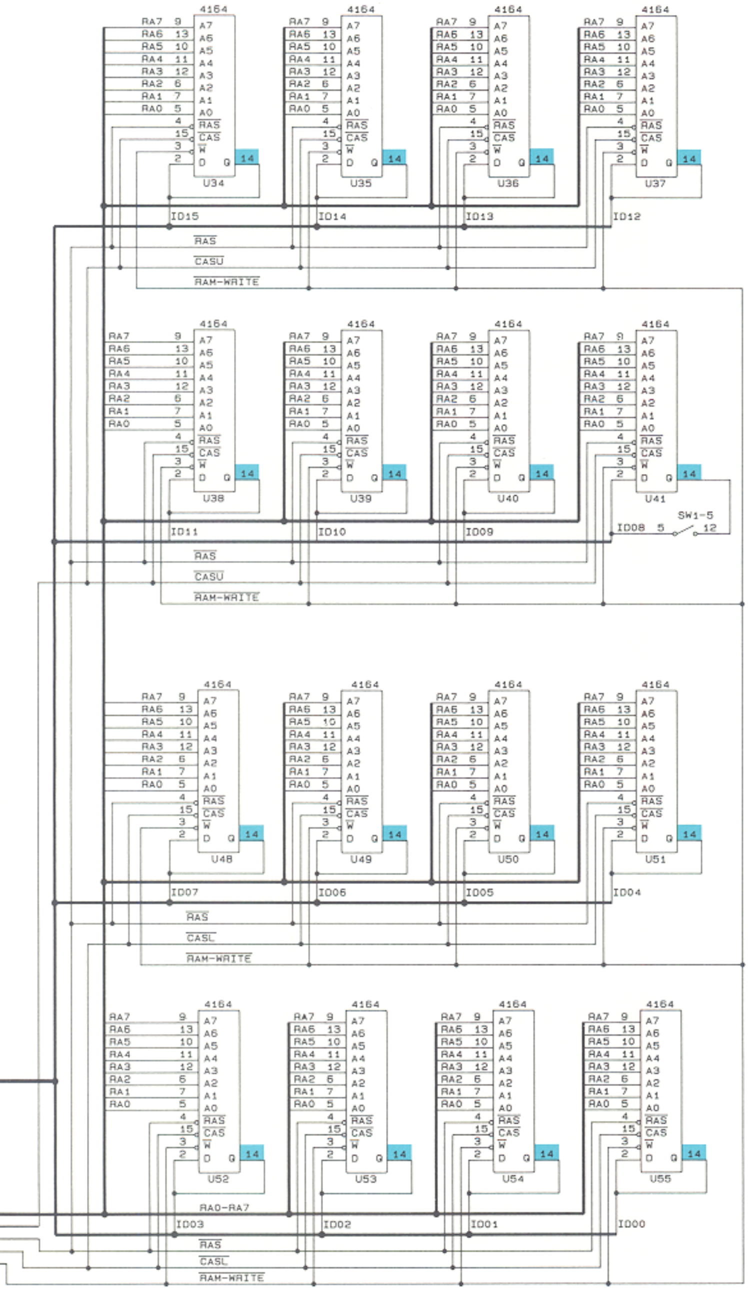


Figure 4-20: RAM Functional Test

Programmed Functional Test 4.3.5.

The *test_ram* program is the programmed functional test of the Dynamic RAM functional block. This program uses the *testramfast* command to test the RAM. This command will generate one of eleven different fault conditions if the *testramfast* fails. All eleven fault condition handlers pick up some parameters and redirect the fault condition to a new fault condition called *ram_component.* The fault condition handler for the *ram_component* fault condition accepts a parameter called *data_bits* that indicates which data bit positions are faulty.

The *ram_component* fault condition handler first checks the Ready circuit to make sure that a ready fault condition is not causing RAM failures. If the Ready circuit is good, one of the failing RAMs (as indicated by the *data_bits* parameter) is checked using the *gfi test* command. If a failure is found, GFI takes control and backtraces to the circuit fault causing the failure.

If the RAM component is good, the *ram_component* fault condition handler uses the *gfi test* command to check the data bus at the bus buffers. If a failure is detected, GFI begins backtracing from the bus buffers.

```
program test_ram

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the RAM functional block.                         !
!                                                                      !
! This program tests the RAM functional block of the Demo/Trainer.  The !
! TL/1 testramfast command is used to test the RAMs.  If the RAMs are  !
! found to be faulty, then one of twelve built-in fault conditions is  !
! generated.                                                           !
!                                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Setup

   podsetup 'enable ~ready' "on"
   podsetup 'report forcing' "on"
   setspace space (getspace space "memory", size "word")

! Main part of test

   testramfast addr 0, upto $1FFFE, delay 250, seed 1

end program
```

Stimulus Programs and Responses 4.3.6.

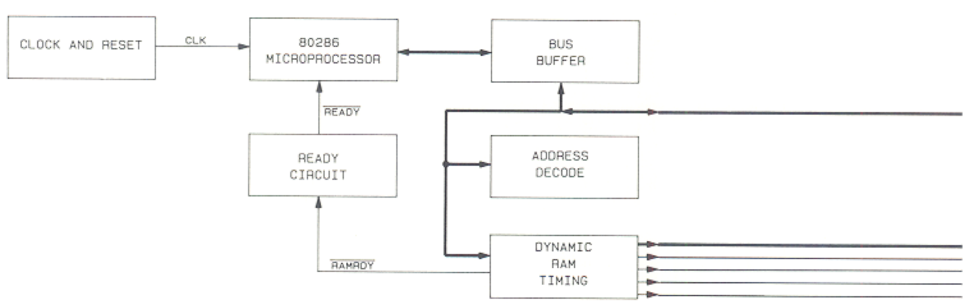
Figure 4-21 is the stimulus program planning diagram for the RAM functional block. There is one stimulus program and a matching response file for RAM. The stimulus program *ram_data* outputs data from RAM onto the data bus.

One rule for a stimulus program is that data should flow in only one direction during the measurement portion of the stimulus program. Although *ram_data* executes *ram_fill* in order to fill RAM with known data, *ram_fill* is executed before the measurement is started in the *ram_data* stimulus program. Therefore data will flow only in one direction during the measurement portion of *ram_data.*

Stimulus Program Planning

PROGRAM: RAM_DATA			
EXECUTES RAM_FILL AND READS FROM THE FIRST 512 LOCATIONS OF RAM			
MEASUREMENT AT:			
U55-14	U51-14	U41-14	U37-14
U54-14	U50-14	U40-14	U36-14
U53-14	U49-14	U39-14	U35-14
U52-14	U48-14	U38-14	U34-14

INITIALIZATION PROGRAM: RAM_FILL
INITIALIZES RAM BY FILLING THE FIRST 512 LOCATIONS OF RAM WITH STANDARD DATA
MEASUREMENT AT:
(NONE)



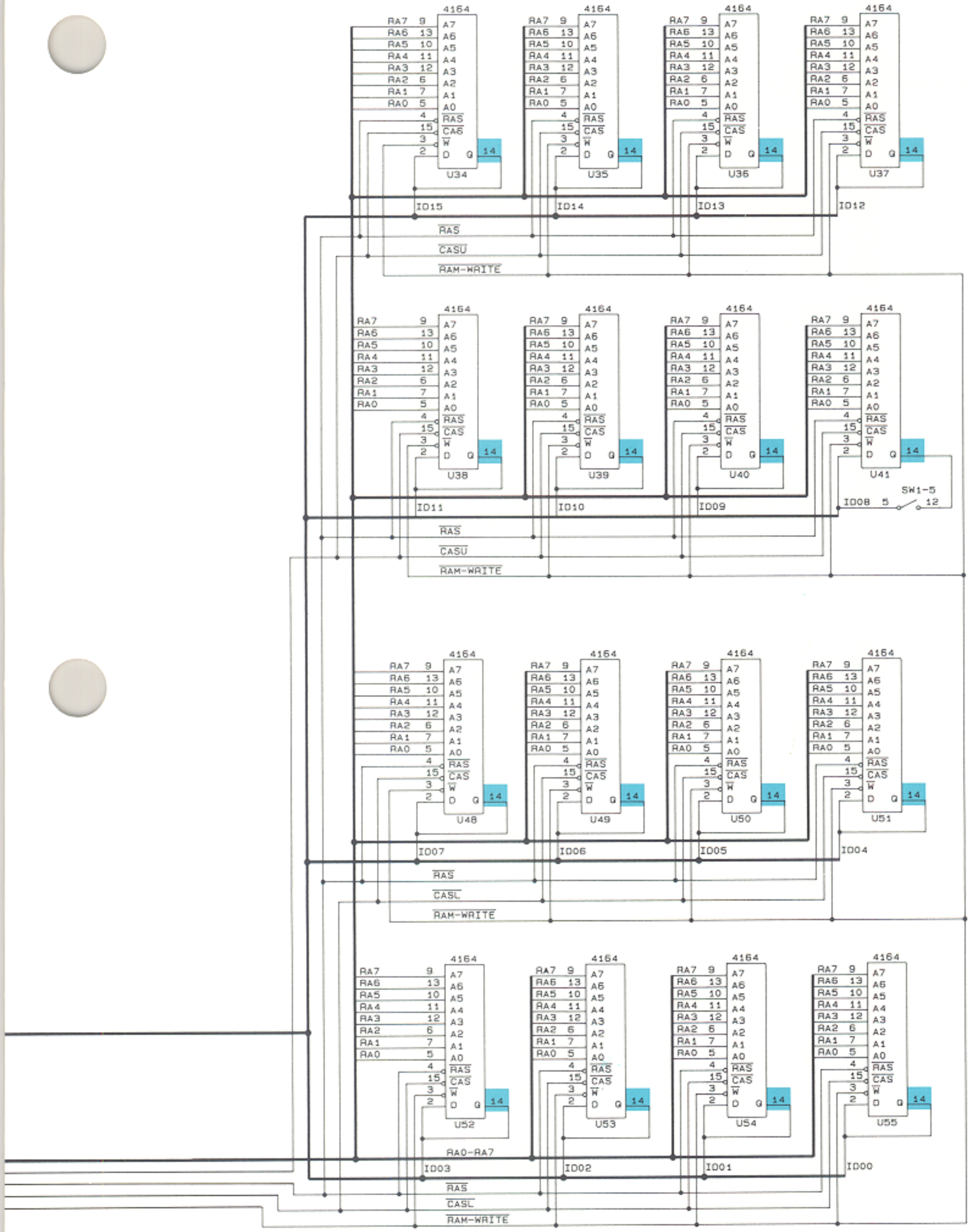


Figure 4-21: RAM Stimulus Program Planning

```
program ram_data

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM to exercise data out of the dynamic RAM.            !
!                                                                      !
! Stimulus programs and response files are used by GFI to backtrace    !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.   !
!                                                                      !
! This stimulus program is one of the programs which creates activity  !
! in the kernel area of the UUT.  These programs create activity with  !
! or without the ready circuit working properly.  Because of this, all  !
! the stimulus programs in the kernel area must disable the READY input !
! to the pod, then perform the stimulus, then re-enable the READY input !
! to the pod.  The 80286 microprocessor has a separate bus controller;  !
! for this reason, disabling ready and performing stimulus can get the  !
! bus controller out of synchronization with the pod.  Two fault        !
! handlers trap pod timeout conditions that indicate the bus controller !
! is out of synchronization.  The recover() program is executed to      !
! resynchronize the bus controller and the pod.                        !
!                                                                      !
! TEST PROGRAMS CALLED:                                                !
!    dram_fill1  ()                        Initialize data in the RAM   !
!                                                                      !
!    recover     ()                        The 80286 microprocessor has a!
!                                          bus controller that is totaly !
!                                          separate from the pod.  In    !
!                                          some cases the pod can get out!
!                                          of sync with the bus control- !
!                                          ler.  The recover program     !
!                                          resynchronizes the pod and the!
!                                          bus controller.               !
!                                                                      !
! GRAPHICS PROGRAMS CALLED:                                            !
!    (none)                                                            !
!                                                                      !
! Local Variables Modified:                                            !
!    devname                               Measurement device          !
!                                                                      !
! Global Variables Modified:                                           !
!    recover_times                         Reset to Zero               !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
```

(continued on the next page)

Figure 4-22: Stimulus Program *(ram_data)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI user select which I/O module to use

   if (gfi control) = "yes" then
     devname = gfi device
   else
     devname = "/mod1"
   end if
   print "Stimulus Program RAM_DATA"

! Set desired measurement modes

   reset device devname
   execute ram_fill()
   setspace space (getspace space "memory", size "word")
   sync device devname, mode "pod"
   sync device "/pod", mode "data"

! Present stimulus:  Read data out of RAM

   arm device devname              ! Start response capture.
      rampaddr addr 0, mask $1FE
   readout device devname          ! End response capture

end program
```

Figure 4-22: Stimulus Program *(ram_data) - continued*

STIMULUS PROGRAM: RAM_DATA
DESCRIPTION: SIZE: 454 BYTES

		--- Response Data ---					
Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
U34-14	I/O MODULE	95A1		1 0	TRANS		
U35-14	I/O MODULE	6F97		1 0	TRANS		
U36-14	I/O MODULE	7744		1 0	TRANS		
U37-14	I/O MODULE	5AE5		1 0	TRANS		
U38-14	I/O MODULE	A54D		1 0	TRANS		
U39-14	I/O MODULE	797B		1 0	TRANS		
U40-14	I/O MODULE	A5F7		1 0	TRANS		
U41-14	I/O MODULE	3BEF		1 0	TRANS		
U48-14	PROBE	C0A6		1 0	TRANS		
U48-14	I/O MODULE	C0A6		1 0	TRANS		
U49-14	PROBE	1338		1 0	TRANS		
U49-14	I/O MODULE	1338		1 0	TRANS		
U50-14	PROBE	66F9		1 0	TRANS		
U50-14	I/O MODULE	66F9		1 0	TRANS		
U51-14	PROBE	6CF8		1 0	TRANS		
U51-14	I/O MODULE	6CF8		1 0	TRANS		
U52-14	PROBE	BE05		1 0	TRANS		
U52-14	I/O MODULE	BE05		1 0	TRANS		
U53-14	PROBE	3C7C		1 0	TRANS		
U53-14	I/O MODULE	3C7C		1 0	TRANS		
U54-14	PROBE	70F3		1 0	TRANS		
U54-14	I/O MODULE	70F3		1 0	TRANS		
U55-14	PROBE	DACC		1 0	TRANS		
U55-14	I/O MODULE	DACC		1 0	TRANS		

Figure 4-23: Response File *(ram_data)*

```
program ram_fill

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! INITIALIZATION PROGRAM fills Dynamic RAM with a pattern of data.
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    (none)                                                             !
!                                                                       !
! GRAPHICS PROGRAMS CALLED:                                             !
!    (none)                                                             !
!                                                                       !
! Text Files Accessed:                                                  !
!    dram_filll                                                         !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   setspace space (getspace space "memory", size "word")
   writeblock file "dram_filll", format "motorola"

end program
```

Figure 4-24: Inititalization Program *(ram_fill)*

Summary of Complete Solution for RAM 4.3.7.

The entire set of programs and files needed to test and GFI troubleshoot the RAM functional block is shown below. The format below is similar to a 9100A/9105A UUT directory (you could consider the functional block to be a small UUT), but in addition shows the use of each program and the location in this manual for each file.

UUT DIRECTORY
(Complete File Set for RAM)

Programs (PROGRAM):

TEST_RAM	Functional Test	Section 4.3.5
RAM_DATA	Stimulus Program	Figure 4-22
RAM_FILL	Initialization Program	Figure 4-24

Stimulus Program Responses (RESPONSE):

RAM_DATA		Figure 4-23

Node List (NODE):

NODELIST		Appendix B

Text Files (TEXT):

DRAM_FILL1	Initialization Data File	

Reference Designator List (REF):

REFLIST		Appendix A

Compiled Database (DATABASE):

GFIDATA		Compiled by the 9100A

DYNAMIC RAM TIMING FUNCTIONAL BLOCK 4.4.

Introduction to Dynamic RAM Timing Circuits 4.4.1.

Unlike static RAM, dynamic RAM chips use a capacitor for charge storage and therefore must be periodically refreshed to maintain the data in memory. Refreshing does not require that data be re-written at memory locations; it requires only that every row be accessed within a certain time period (typically at least every 2 milliseconds). This is sufficient to restore the charge on the memory cells.

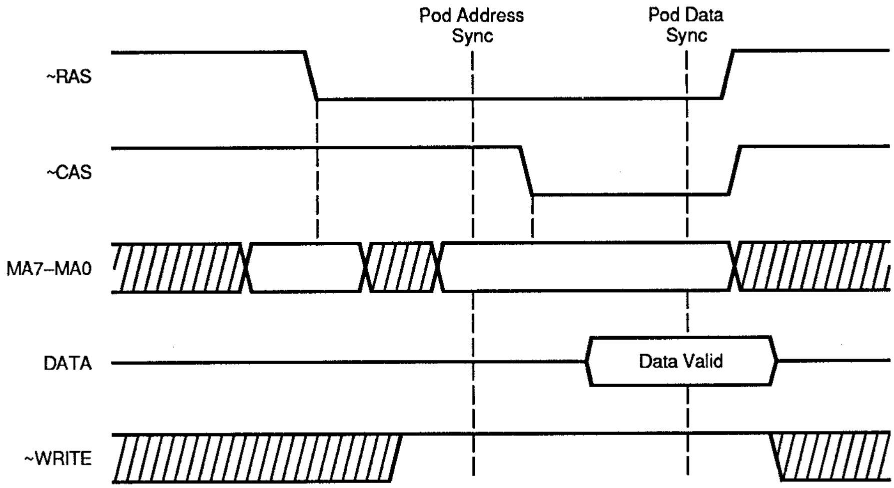
In addition, dynamic RAM uses multiplexed address signals. The row address is clocked into the internal decoder of the dynamic RAM chip with the falling edge of the Row Address Strobe (RAS), and the column address is clocked with the falling edge of the Column Address Strobe (CAS). Multiplexed addressing decreases the pin count and package size, but it also makes dynamic RAM more difficult to test and troubleshoot than static RAM.

Considerations for Testing and Troubleshooting 4.4.2.

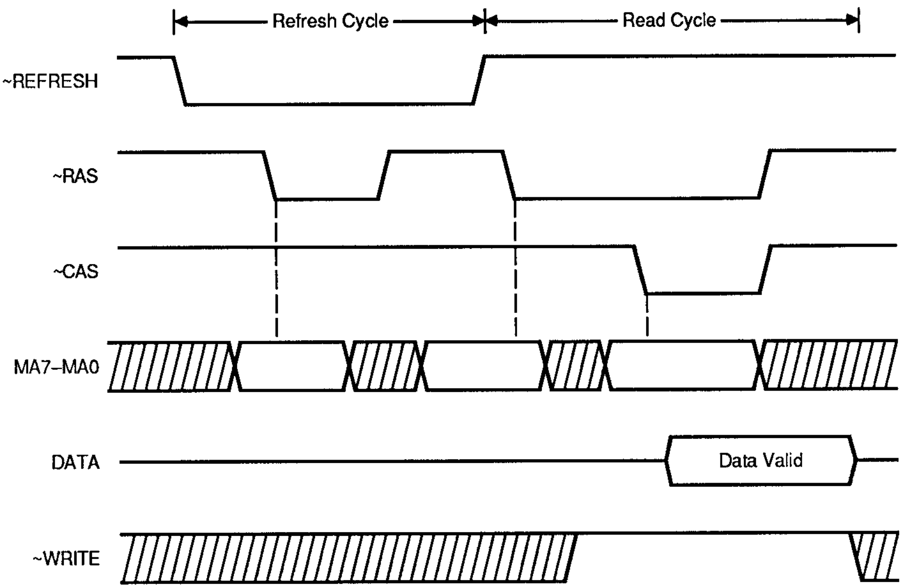
The thought process used to test and troubleshoot dynamic RAM is very similar to that used for static RAM, but the actual measurements for dynamic RAM are more difficult because of row and column strobing for multiplexed addresses, because of refreshing, and because there are more failure mechanisms.

Consider, for example, a dynamic RAM with 64K memory locations addressed by eight address inputs (MA7-MA0). A multiplexer allows the 16 address lines to be brought to the eight RAM address lines, using RAS to strobe the row address and CAS to strobe the column address. Typical timing for a read cycle of such a system is shown in Figure 4-25.

With static RAM, the microprocessor's address lines can be tested by making measurements using the probe or an I/O



Dynamic RAM Read Cycle, Without Refresh



Dynamic RAM Read Cycle, With RAS-Only Refresh

Figure 4-25: Dynamic RAM Read Cycles

module synchronized to the pod address while performing looping reads or writes. With dynamic RAM, however, the RAM's address inputs are multiplexed between row and column addresses. It is important to be able to separate row addressing from column addressing. To test dynamic RAM addressing requires the ability to control the timing of the clock strobe for a measurement. The 9100/9105A has this capability; under program control, it can adjust the timing for when the probe or I/O module actually clocks data. Using the *getoffset* and *setoffset* commands, you can create a program to measure the address line activity on the RAM chips at the RAS strobe (or at the CAS strobe). Typically, it makes sense to have two separate programs: one to measure activity for RAS address timing and one to measure activity for CAS address timing.

For the top example of Figure 4-25, the TL/1 *setoffset* and *getoffset* commands are used to adjust the sync timing from Pod Data Sync (or Pod Address Sync) to the RAS and CAS positions. One program could be used to measure at RAS time and another to measure at CAS time. The I/O module or probe used to measure the RAS and CAS address activity would be synchronized to Pod Data Sync or Pod Address Sync. However, the *setoffset* command provides an offset from Pod Data Sync or Pod Address Sync that determines when the clocking for measurements actually occurs.

For some designs, more than one RAS cycle can occur during a read or write cycle. The bottom half of Figure 4-25 shows typical timing for such a situation. RAS goes low first for a refresh and then again later for the read. In this case, it is not sufficient to clock measurements on address lines with RAS alone. If you want to examine the row address signals on the address lines, you could use the Refresh signal to qualify clocking for the appropriate address information.

Measuring the RAS and CAS Lines

An easy check for RAS and CAS lines is to look for activity on the lines. With the probe or I/O module synchronized to the FREERUN clock, an asynchronous level history for RAS should always show high and low levels and never an invalid level. An asynchronous level history for CAS will be the same as RAS if it is being accessed at the time. When the RAM is not being accessed, CAS may be similarly active or it may remain high, depending on the UUT.

Although the absence of the proper levels described above will indicate some types of faults, these simple checks cannot determine if the lines are definitely good. Subtle timing problems are common with some dynamic RAM designs.

To analyze the exact timing of RAS and CAS, use the 9100A/9105A to generate the appropriate sync signal and to display the UUT waveforms on an oscilloscope:

1. Use the SYNC key on the operator's keypad to select the Pod Address Synchronization mode:

```
   SYNC TO POD ADDR
or SYNC I/O MOD <number> TO POD ADDR
```

2. Use the READ key on the operator's keypad to enter the following command:

```
   READ FAST FOREVER ADDR <ram address>
```

3. Synchronize an oscilloscope to the TRIGGER OUTPUT sync output on the rear panel of the 9100A/9105A.

4. Study the oscilloscope waveforms at the dynamic RAM chips.

Once the timing of RAS and CAS (as well as other dynamic RAM signals) is understood from the above procedure, two options are available. The first is to troubleshoot directly with a synchronized oscilloscope, and the second is to write a TL/1 program to automate the procedure.

Determining If Refresh Signals Are Working

Typical dynamic RAM must access every row address for cell refresh at least every 2 milliseconds. The ability of the 9100A/9105A to measure frequency min-max is the simplest tool for troubleshooting the circuitry that implements this refresh. No matter how the refresh circuitry is designed, the refresh signals (refresh address, RAS, and related timing signals) are on a regular schedule of one full cycle in less than 2 milliseconds. For a first-cut characterization of these signals, try measuring frequency min-max.

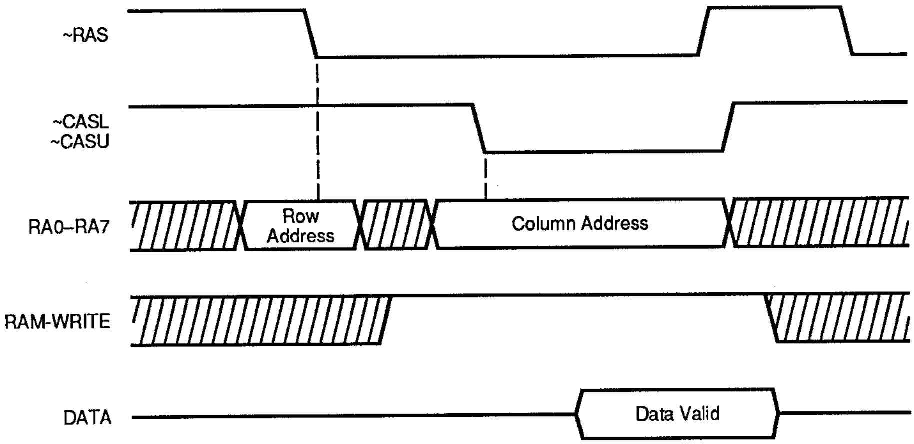
For a more precise characterization of the refresh signals, use the external synchronization capabilities (start, stop, clock) of the 9100A/9105A. Characterize all related signals during the start/stop interval of one refresh cycle, and then characterize the signals used for start/stop/clock with frequency min-max.

Dynamic RAM Timing Circuit Example 4.4.3.

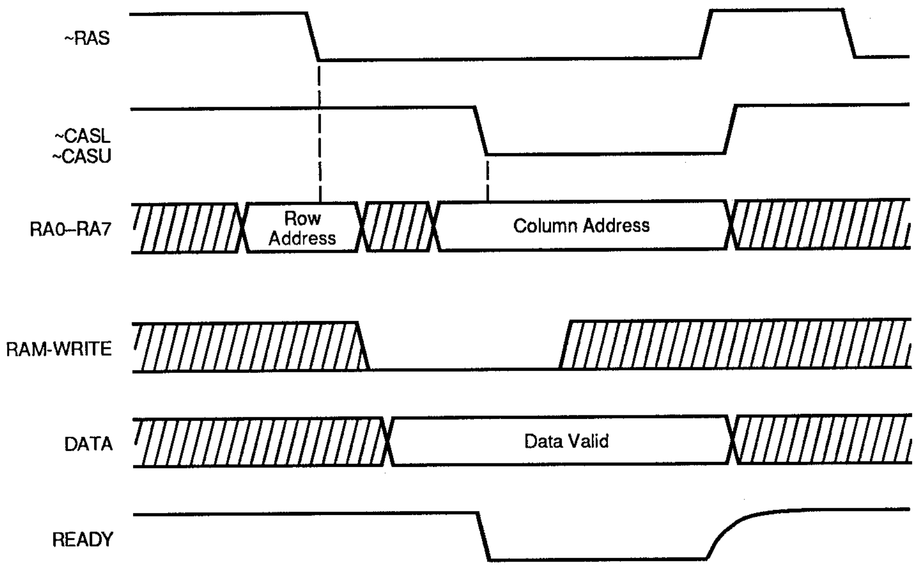
A diagram of read/write timing for the Demo/Trainer UUT's RAM timing circuit is shown in Figure 4-26. The circuit schematic is shown in Figure 4-28.

Accessing

To select RAM, U65 and U66 multiplex 16 address lines into eight lines. The multiplexed address is then latched into the RAM chips by two externally applied clock pulses. The first, the negative-going edge of the row-address strobe (~RAS), latches the eight row-address bits. The second, the negative-going edge of the column-address strobe (~CAS), latches the eight column-address bits. Timing for RAS and CAS is determined by delay line U60. CAS is a delayed RAS signal; it goes low 55 nsec after RAS goes low.



Read Cycle



Write Cycle

Figure 4-26: Dynamic RAM Read/Write Timing

The 80286 can access upper and lower bytes separately, or together as a word. RAM is organized as 128K bytes, addressed from 00000 to 1FFFE. Access is accomplished by gating ~CASL and ~CASU (U58D). IA00 (internal buffered address bit zero) selects DO-D7 and ~IBHE (Internal Buffered Bus High Enable) selects D8-D15. The low byte is accessed when IA00 is low. The high byte is accessed when IBHE- is low. The entire word is accessed when both IA00 and ~IBHE are low. The 80286 determines the type of access based on the instruction being executed.

Refreshing

RAM Refresh timing is illustrated in Figure 4-27.

To maintain data, each of the 128 RAS addresses must be refreshed (or read) every 2 msec. The Demo/Trainer UUT uses the RAS-only refresh method for this purpose. A RAS-only refresh cycle asserts only the RAS line to strobe in the refresh address.

A single Demo/Trainer UUT row refresh occurs every 15 μsec. A complete refresh entails 128 row refreshes, requiring about 1.9 msec.

The RFRQ (Refresh Request) signal both marks the need for a refresh cycle and increments the refresh address counter U67. U42 and U43 are used to divide PCLK (4 MHz) by 16 to produce RFRQ.

RAM refresh and RAM access are mutually exclusive. U61D insures that a refresh cannot occur if a RAM access is in progress. Conversely, if a refresh is in progress and the processor asks for a RAM access, U58B prevents Ready from being returned, causing the addition of a wait state. The processor is thus put on hold until the refresh is completed.

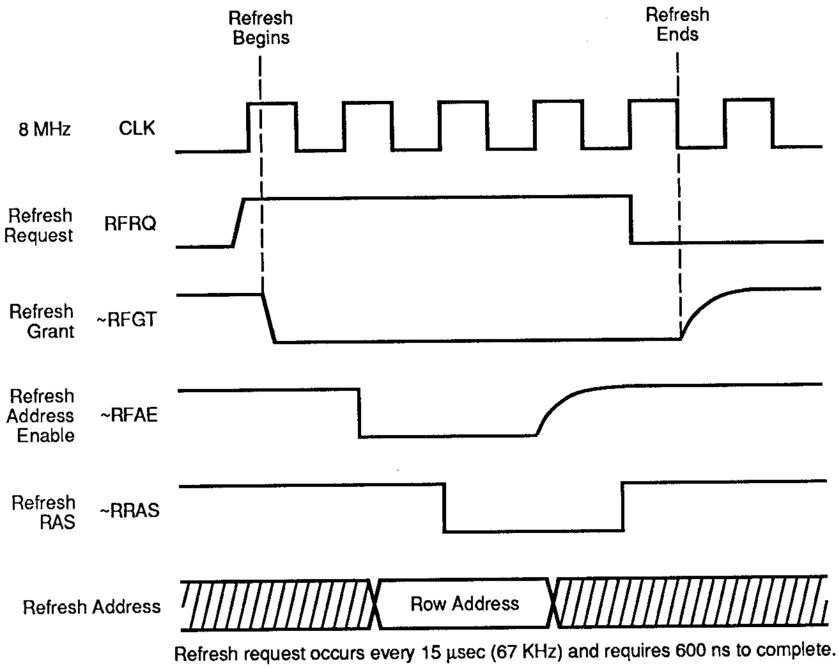


Figure 4-27: RAM Refresh Timing

RAM refresh is performed as follows:

1. If ~RAM is high (no RAM access in progress) and refresh is being requested, U61D outputs RFGT (Refresh Grant).

2. RFGT high enables the U44A/U44B state machine. This circuit times the output of Refresh Address Enable (RFAE) to U67. After the proper refresh address setup time, it also enables Refresh RAS (RRAS) to strobe in the refresh address.

3. After the refresh address is strobed in, RFGT goes low, allowing the processor access to the RAM.

Keystroke Functional Test 4.4.4.

1. Use a 16-pin clip module on side A of I/O module 1 to check CAS addresses and select line. Use the the EXEC and I/O MOD keys with the commands below for each of the following parts: U65, U66 and U26. The correct measurement for each pin is shown in the table below.

```
EXECUTE UUT DEMO PROGRAM CAS_STIM
SHOW I/O MOD 1 PIN <see table> CAPTURED ...
... RESPONSES
```

NOTE

The SHOW command requires a clip module pin number rather than a part pin number. This requires you to translate part pin numbers to clip module pin numbers (see Appendix B of the Technical User's Manual). For your convenience, this translation has been done for you in this example, and the results are shown in the "I/O MOD PIN" column of the response table in the next figure.

SIGNAL	*PART/PIN*	*I/O PIN*	*SIGNATURE*
RA0	U65-4	4	0140
RA1	-7	7	02AF
RA2	-9	13	0150
RA3	-12	16	03A9
RA4	U66-4	4	00D3
RA5	-7	7	022A
RA6	-9	13	0151
RA7	-12	16	0263
RAM-WRITE	U26-8	14	0352

2. Use a 16-pin clip module on side A of I/O module 1 to check RAS addresses. Use the the EXEC and I/O MOD keys with the commands below for each of the following parts: U65 and U66. The correct measurement for each pin is shown in the table below.

```
EXECUTE UUT DEMO PROGRAM RAS_STIM
SHOW I/O MOD 1 PIN <see table> CAPTURED ...
... RESPONSES
```

SIGNAL	*PART/PIN*	*I/O PIN*	*SIGNATURE*
RA0	U65-4	4	02BF
RA1	-7	7	0154
RA2	-9	13	022A
RA3	-12	16	01D1
RA4	U66-4	4	022A
RA5	-7	7	0150
RA6	-9	13	022B
RA7	-12	16	0114

3. The next step is measuring refresh signals that are active with no stimulus. Use a 16-pin clip module on side A of I/O module 1 to test refresh signals on RA0-RA7. Connect the external control lines as follows:

Start to U67-9
Stop to U67-9
Clock to U63-8

Use the the SYNC and I/O MOD keys with the commands below to measure refresh signals. The correct measurement for each pin is shown in the table below.

```
SYNC I/O MOD 1 TO EXT ENABLE ALWAYS ...
... CLOCK ↓ START ↑ STOP ↑
ARM I/O MOD 1 FOR CAPTURE USING SYNC
SHOW I/O MOD 1 PIN <see table> CAPTURED ...
... RESPONSES
```

SIGNAL	*PART/PIN*	*I/O PIN*	*SIGNATURE*
RA0	U65-4	4	968C
RA1	-7	7	AFC1
RA2	-9	13	4A2C
RA3	-12	16	25AF
RA4	U66-4	4	ACDE
RA5	-7	7	122D
RA6	-9	13	EEA6
RA7	-12	16	68F8

4. Use a 14-pin clip module on side A of I/O module 1 to check the select logic. Use the the EXEC and I/O MOD keys with the commands below. The correct measurement for each pin is shown in the response table in Figure 4-28.

```
EXECUTE UUT DEMO PROGRAM RAMSELECT1
SHOW I/O MOD 1 PIN 14 CAPTURED RESPONSES
```

5. Use a 14-pin clip module on side A of I/O module 1 to check the select logic. Use the the EXEC and I/O MOD keys with the commands below. The correct measurement for each pin is shown in the response table in Figure 4-28.

```
EXECUTE UUT DEMO PROGRAM RAMSELECT2
SHOW I/O MOD 1 PIN 14 CAPTURED RESPONSES
SHOW I/O MOD 1 PIN 17 CAPTURED RESPONSES
```

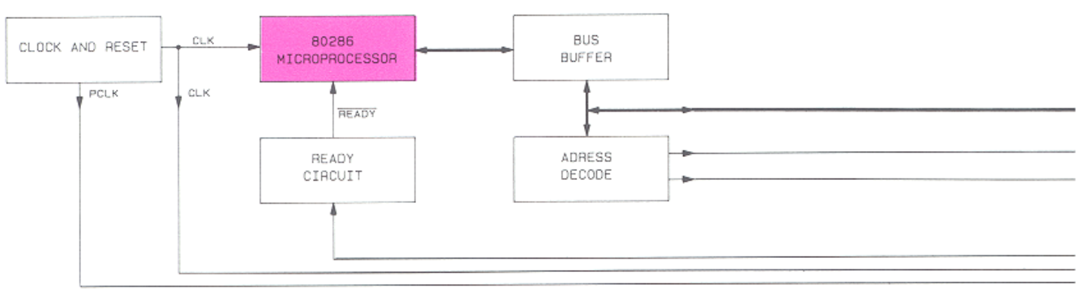
Keystroke Functional Test

CONNECTION TABLE

STIMULUS	MEASUREMENT
POD TEST ACCESS SOCKET	I/O MOD U65 U26 U66 U63 U58

RESPONSE TABLE

SIGNAL	PART PIN	I/O MOD PIN	SIGNATURE
RA0	U65-4	4	SEE TEXT
RA1	-7	7	
RA2	-9	13	
RA3	-12	16	
RA4	U66-4	4	SEE TEXT
RA5	-7	7	
RA6	-9	13	
RA7	-12	16	
RAM-WRITE	U26-8	14	
RASS	U63-8	14	01B6
CASU	U58-8	14	B6FD
CASL	-11	17	B603



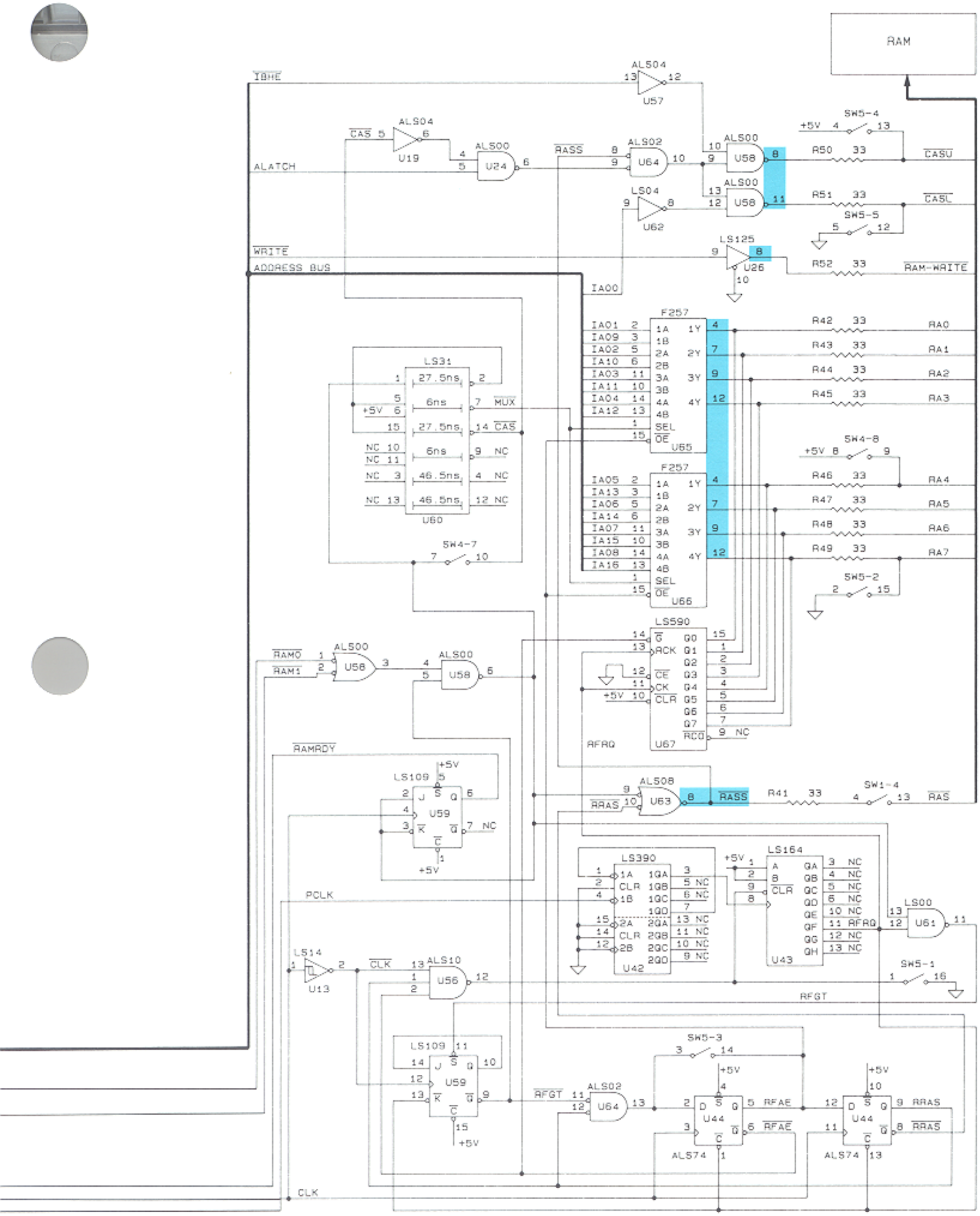


Figure 4-28: Dynamic RAM Timing Functional Test

Programmed Functional Test 4.4.5.

The *tst_refrsh* program is the programmed functional test for the Dynamic RAM Timing functional block. This program checks the outputs at U65, U58, U63 and U25 using the *gfi test* command. If the *gfi test* command fails, the *abort_test* program is executed and GFI troubleshooting begins. (See the Bus Buffer functional block for a discussion of the *abort_test* program).

```
program tst_refrsh

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the DYNAMIC RAM REFRESH functional block.          !
!                                                                       !
! This program tests the DYNAMIC RAM REFRESH functional block of the    !
! Demo/Trainer. The gfi test command and I/O module are used to perform !
! the test.                                                             !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    abort_test (ref-pin)                 If gfi has an accusation      !
!                                         display the accusation else   !
!                                         create a gfi hint for the     !
!                                         ref-pin and terminate the test!
!                                         program (GFI begins trouble-  !
!                                         shooting).                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   print "\nlTESTING RAM TIMING & REFRESH Circuit"

   podsetup 'enable ~ready' "on"

   if gfi test "U65-1" fails then abort_test("U65-1")
   if gfi test "U66-1" fails then abort_test("U66-1")

   print "RAM TIMING & REFRESH TEST PASSES"
end program
```

Stimulus Programs and Responses 4.4.6.

Figure 4-29 is the stimulus program planning diagram for the Dynamic RAM Timing functional block. The *ras_stim* and *cas_stim* stimulus programs both perform read and write accesses to various addresses in RAM. However, the *getoffset* and *setoffset* commands are used to adjust the timing when the data is measured, so that *cas_stim* measures data when CAS addresses are valid and *ras_stim* measure data when RAS addresses are valid.

The *ramselect1* and *ramselect2* programs provide stimulus for measurement of a number of logic outputs. The *refsh_addr, refsh_time,* and *refsh_u56* programs provide stimulus for measurement at various ICs that perform the RAM refresh function. The *frequency* program measures frequency at a number of nodes.

Stimulus Program Planning

PROGRAM: CAS_STIM

EXERCISES THE CAS ADDRESS

MEASUREMENT AT:

U65-4,7,9,12
U66-4,7,9,12
U26-8

PROGRAM: RAS_STIM

EXERCISES THE RAS ADDRESS

MEASUREMENT AT:

U65-4,7,9,12
U66-4,7,9,12

PROGRAM: RAMSELECT1

EXERCISES THE RAM SELECT LOGIC

MEASUREMENT AT:

U19-6
U24-6
U64-10
U60-2,7,14
U58-3,6
U59-6,10,9
U63-8
U61-11

PROGRAM: RAMSELECT2

EXERCISES THE RAM SELECT LOGIC

MEASUREMENT AT:

U57-12
U62-8
U58-8,11

PROGRAM: REFSH_ADDR

MEASURES THE REFRESH ADDRESS SEQUENCE

MEASUREMENT AT:

U67-15,1,2,3,4,5,6,7

PROGRAM: FREQUENCY

MEASURES FREQUENCY

MEASUREMENT AT:

U13-2
U56-12
U42-3,7
U43-11

PROGRAM: REFSH_TIME

MEASURES REFRESH TIMING

MEASUREMENT AT:

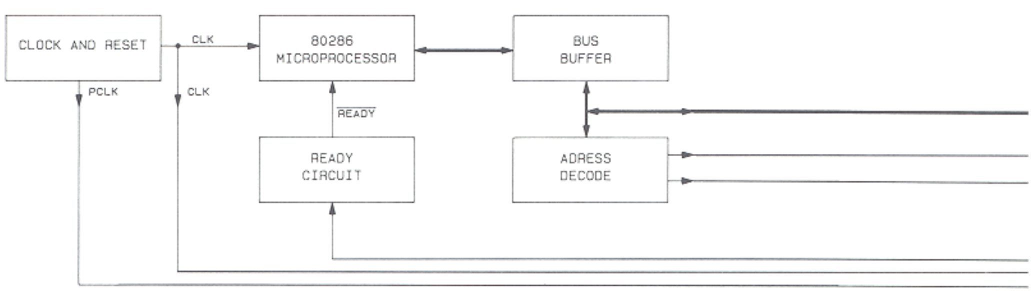
U61-11
U59-10,9
U64-13
U44-5,6,9,8
U43-11

PROGRAM: REFSH_U56

MEASURES REFRESH TIMING FOR U56

MEASUREMENT AT:

U56-12



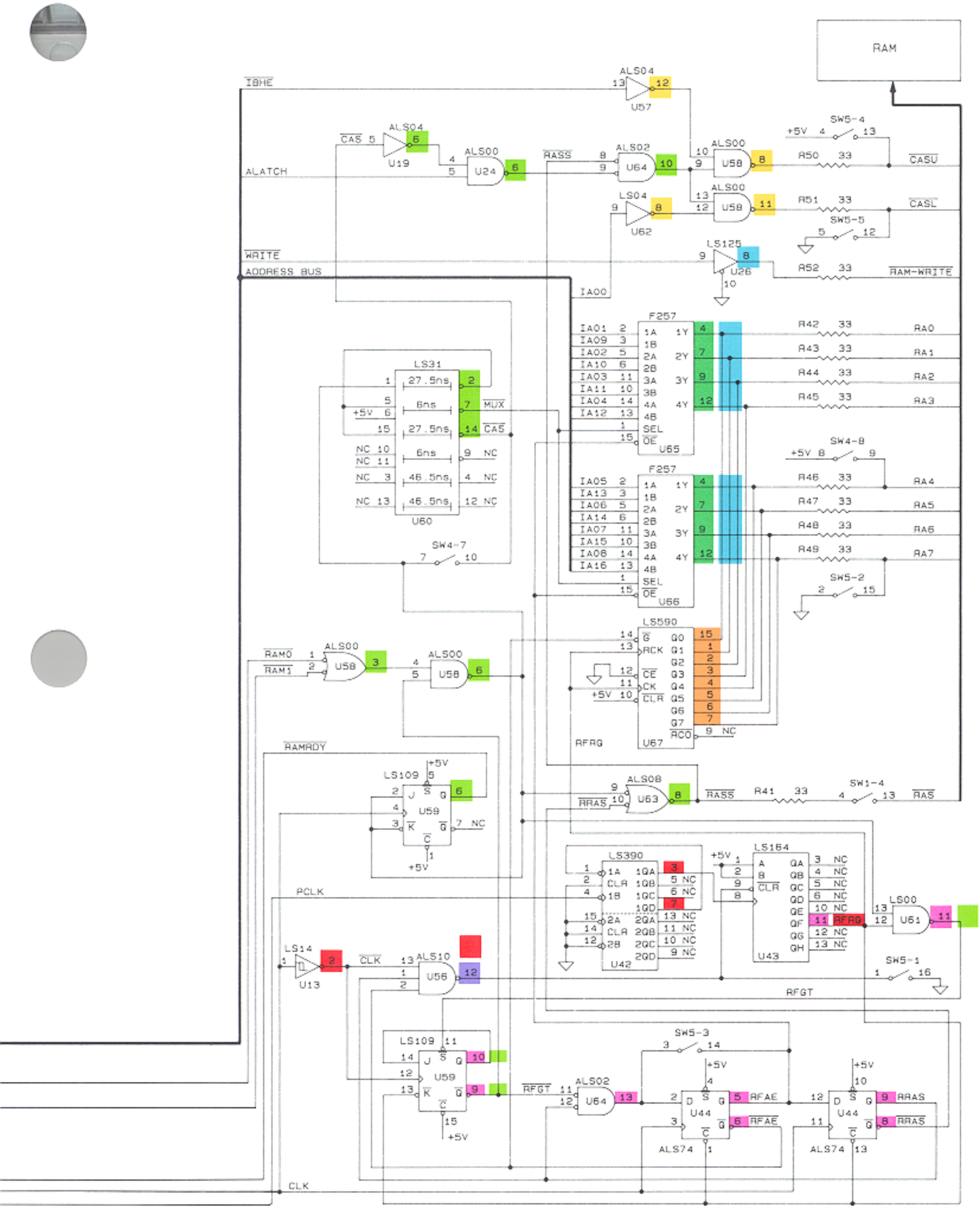


Figure 4-29: Dynamic RAM Timing Stimulus Program Planning

```
program cas_stim

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM characterizes CAS address lines.                      !
!                                                                        !
! Stimulus programs and response files are used by GFI to back-trace     !
! from a failing node.  The stimulus program must create repeatable UUT  !
! activity and the response file contains the known-good responses for   !
! the outputs in the UUT that are stimulated by the stimulus program.    !
!                                                                        !
! This stimulus program is one of the programs which creates activity    !
! in the RAM area of the UUT.  This stimulus program uses the setoffset  !
! and getoffset commands to adjust the timing to CAS address valid.      !
!                                                                        !
! TEST PROGRAMS CALLED:                                                  !
!                                                                        !
! GRAPHICS PROGRAMS CALLED:                                              !
!    (none)                                                              !
!                                                                        !
! Local Variables Modified:                                              !
!    devname                            Measurement device               !
!    bias                               Offset value to use              !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare numeric bias = 999957

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                        !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Let GFI determine the measurement device.

   if (gfi control) = "yes" then
      devname = gfi device
   else
      devname = "/mod1"
   end if
   print "Stimulus Program CAS_STIM"
```

(continued on the next page)

Figure 4-30: Stimulus Program *(cas_stim)*

```
! Set addressing mode and setup measurement device.

   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   setspace space (getspace space "memory", size "word")
   reset device devname
   sync device devname, mode "pod"
   sync device "/pod", mode "data"

! Store calibration offset, set new offset
! Display warning message if setting new offset fails

   cal_offset = getoffset device devname
   if (setoffset device devname, offset bias) = 0 then
      fault 'setoffset returned a bad status, fatal error'
   end if

! Present stimulus to UUT.

   arm device devname
      read addr $AB54         ! This addr gives complmentary CAS address
      read addr $1549A
      write addr $1234, data $4320
      read addr $55AA
      write addr $AB54, data $AAAA
      read addr $156A8
      write addr $AA54, data $55AA
      read addr $1AD50
      write addr $1FFFE, data $FFFE
      read addr $2AD4
   readout device devname

! Restore calibration offset

   setoffset device "/mod1", offset cal_offset
end cas_stim
```

Figure 4-30: Stimulus Program *(cas_stim) - continued*

STIMULUS PROGRAM NAME: CAS_STIM
DESCRIPTION: SIZE: 199 BYTES

Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
U65-4	I/O MODULE	0140	1 0		TRANS		
U65-7	I/O MODULE	02AF	1 0		TRANS		
U65-9	I/O MODULE	0150	1 0		TRANS		
U65-12	I/O MODULE	03A9	1 0		TRANS		
U66-7	I/O MODULE	022A	1 0		TRANS		
U66-9	I/O MODULE	0151	1 0		TRANS		
U66-12	I/O MODULE	0263	1 0		TRANS		
u66-4	I/O MODULE	00D3	1 0		TRANS		
u26-8	I/O MODULE	0352	1 0		TRANS		

Figure 4-31: Response File *(cas_stim)*

```
program ras_stim

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM characterizes RAS address lines.                        !
!                                                                          !
! Stimulus programs and response files are used by GFI to backtrace      !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.   !
!                                                                          !
! TEST PROGRAMS CALLED:                                                    !
!    (none)                                                                !
!                                                                          !
! GRAPHICS PROGRAMS CALLED:                                                !
!    (none)                                                                !
!                                                                          !
! Local Variables Modified:                                                !
!    devname                               Measurement device              !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                        !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    declare numeric bias = 999964

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                          !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Let GFI determine the measurement device.

    if (gfi control) = "yes" then
        devname = gfi device
    else
        devname = "/mod1"
    end if
    print "Stimulus Program RAS_STIM"

! Set addressing mode and setup measurement device.

    setspace space (getspace space "memory", size "word")
    reset device devname
    sync device devname, mode "pod"
    sync device "/pod", mode "addr"
```

(continued on the next page)

Figure 4-32: Stimulus Program *(ras_stim)*

```
! Store calibration offset, set new offset
! Display warning message if setting new offset fails

    cal_offset = getoffset device devname
    if (setoffset device devname, offset bias) = 0 then
       fault 'setoffset returned a bad status, fatal error'
    end if

! Present stimulus to UUT.

   arm device devname
      read addr $AB54          ! This addr gives complementary CAS address
      read addr $1549A
      write addr $1234, data $4320
      read addr $55AA
      write addr $AB54, data $AAAA
      read addr $156A8
      write addr $AA54, data $55AA
      read addr $1AD50
      write addr $1FFFE, data $FFFE
      read addr $2AD4
   readout device devname

! Restore the calibrated offset value.

   setoffset device devname, offset cal_offset

end ras_stim
```

Figure 4-32: Stimulus Program *(ras_stim) - continued*

STIMULUS PROGRAM NAME: RAS_STIM
DESCRIPTION: SIZE: 182 BYTES

Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
	--- Response Data ---						
U65-4	I/O MODULE	02BF	1 0		TRANS		
U65-7	I/O MODULE	0154	1 0		TRANS		
U65-9	I/O MODULE	022A	1 0		TRANS		
U65-12	I/O MODULE	01D1	1 0		TRANS		
U66-4	I/O MODULE	022A	1 0		TRANS		
U66-7	I/O MODULE	0150	1 0		TRANS		
U66-9	I/O MODULE	022B	1 0		TRANS		
U66-12	I/O MODULE	0114	1 0		TRANS		

Figure 4-33: Response File *(ras_stim)*

```
program ramselect1

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM to wiggle RAM select circuitry.                       !
!                                                                        !
! Stimulus programs and response files are used by GFI to backtrace      !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.    !
!                                                                        !
! Ramselect1 is used to stimulate the RAM select circuitry after the     !
! decoders.  The stimulus is a combination of reads that will ensure     !
! the decoder and related circuitry is working properly.                 !
!                                                                        !
! TEST PROGRAMS CALLED:                                                  !
!    recover     ()                     The 80286 microprocessor has a!
!                                       bus controller that is totally!
!                                       separate from the pod.  In    !
!                                       some cases the pod can get out!
!                                       of sync with the bus control- !
!                                       ler.  The recover program     !
!                                       resynchronizes the pod and the!
!                                       bus controller.               !
!                                                                        !
! GRAPHICS PROGRAMS CALLED:                                              !
!    (none)                                                              !
!                                                                        !
! Global Variables Modified:                                             !
!    recover_times                      Reset to Zero                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  FAULT HANDLERS:                                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
handle pod_timeout_enabled_line
  recover()
end handle

handle pod_timeout_recovered
  recover()
end handle

! Let GFI determine the measurement device.

    if (gfi control) = "yes" then
        devname = gfi device
    else
        devname = "/mod1"
    end if
    print "Stimulus Program RAMSELECT1"
```

(continued on the next page)

Figure 4-34: Stimulus Program *(ramselect1)*

```
! Set addressing mode and setup measurement device.

    setspace space (getspace space "memory", size "word")
    reset device devname
    sync device devname, mode "pod"
    sync device "/pod", mode "data"

! Present stimulus to UUT.

    arm device devname
       read addr $1A5A4
       read addr $F0000
       read addr $F0000
       read addr $5A5A
       read addr $F0000
       read addr $F0000
       write addr $7BDE, data $1234
       read addr $F0000
       write addr $15A5A, data $9876
       read addr $F0000
   readout device devname

end program
```

Figure 4-34: Stimulus Program *(ramselect1) - continued*