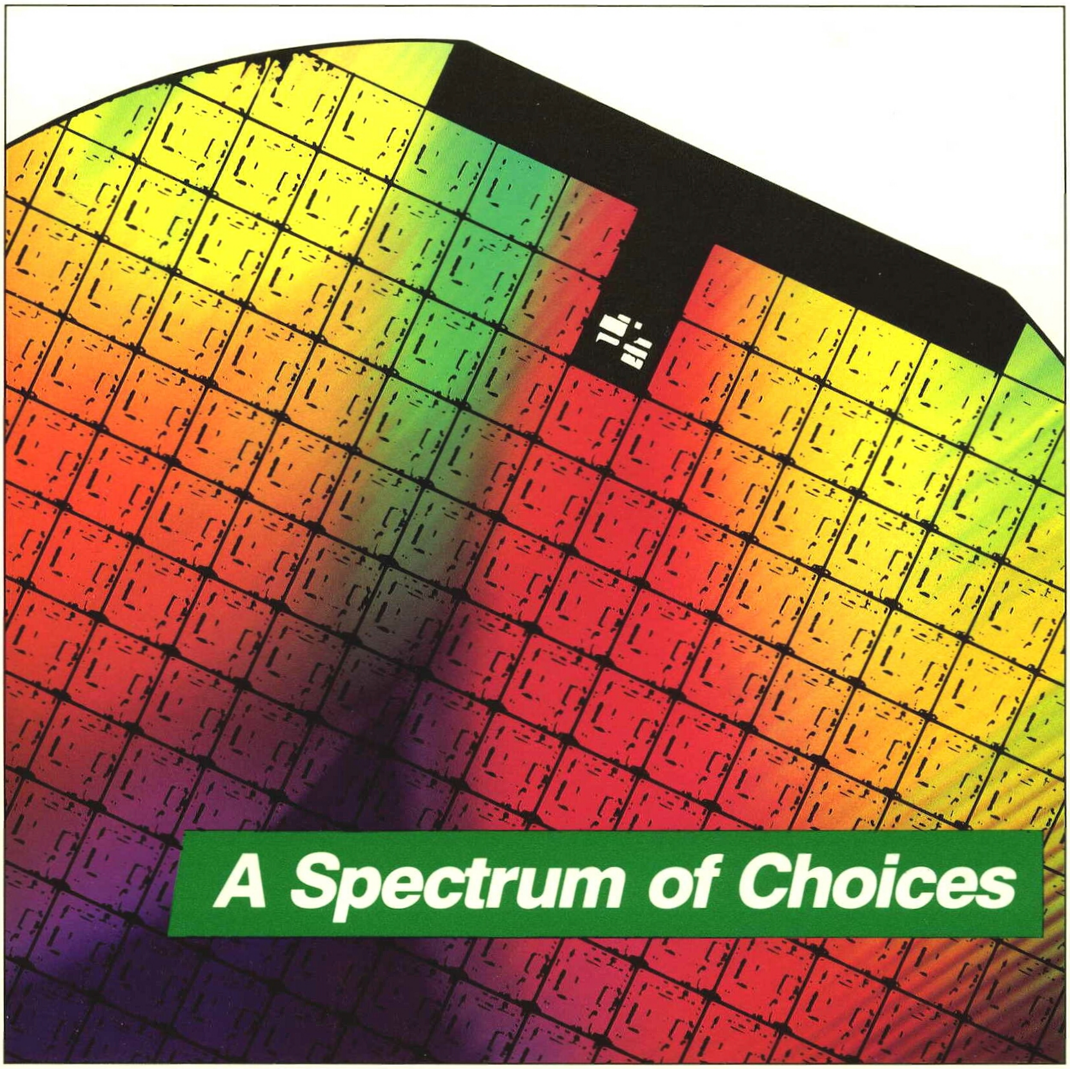
Am29PL100
Field Programmable Controllers

Handbook

Advanced
Micro
Devices



Advanced Micro Devices

Am29PL100 Field Programmable Controller Family

Handbook

901 Thompson Place, P.O. Box 3453, Sunnyvale, California 94088
(408) 732-2400 TWX: 910-339-9280 TELEX: 34-6306

TABLE OF CONTENTS

APPENDICES

CHAPTER 1

FIELD PROGRAMMABLE CONTROLLER OVERVIEW

1.1 DESIGN CHOICES

Sequential state machine design is normally approached using one of two general methods: the traditional random logic and flip-flop approach, or microprogramming. Until recently, traditional methods were used for state machines with relatively few states, e.g. dynamic memory controllers, while microprogramming was used for applications with many states, e.g. CPUs. The area in between was handled with a hodge-podge of techniques ranging from ad-hoc use of counters and shift registers to PROM-based sequencers. Now, Advanced Micro Devices has introduced the Am29CPL100 Field Programmable Controller (FPC) family to allow cost-effective application of microprogramming techniques to fairly small state machines.

Traditional design methodology generally uses state diagrams to define machine behavior, followed by derivation of appropriate J-K flip-flop excitation equations. This approach typically results in very high-speed state machine implementations which are highly optimized for a particular task. Unfortunately, this technique is at best tedious and can be essentially unusable for large state machines.

The microprogramming approach to state machine design consists of storing machine cycle control sequences in memory locations. These instructions are fetched and executed sequentially. Microprogramming is similar to assembly language programming of other processors with subroutines, loops, and structured programming constructs.

1.2 Am29CPL100 ARCHITECTURE OVERVIEW

The Advanced Micro Devices Am29CPL100 is a family of compatible single-chip Field Programmable Controller (FPC) devices. It combines, in one chip, address sequencer logic, program memory, and a powerful instruction set that supports a repertoire of jumps, multiple branches, and subroutine calls. These instructions can be executed conditionally depending on external condition tests. A Serial Shadow Register (SSR) helps designers diagnose system troubles at the individual IC component level. A pipeline register permits fetching the next instruction at the same time that the current instruction is being executed. This Chapter provides a general description of the FPC. For a detailed description, refer to the data sheets in Appendix E.

The first member of the Am29PL100 Family, the Am29PL141, was the first device to combine the elements of an intelligent microcode controller in a single device. From this base, the second generation in the family, the Am29PL142, added more program memory, registered inputs, and a stack. The next generation in the Am29PL100 family, such as the Am29PL142 and Am29CPL144, incorporates low-power, high-performance (25-30 MHz) and CMOS technologies with devices that are 100% compatible with their bipolar equivalents. Further still, new CMOS Am29PL100 devices, like the Am29CPL144, incorporate larger program memory and more features.

The Am29CPL100 architecture consists of four major blocks:

- Address sequencer control logic
- Branch control/condition code select logic
- Instruction decode logic
- Program memory with a pipeline register and SSR

1.2.1 Address Sequencer

As shown in Figure 1-1, FPC control sequences, stored in the on-chip programmable memory, are fetched under control of the address sequencer and clocked into the pipeline register. Figure 1-2 shows a detailed block diagram of one device in the Am29PL100 family: the Am29PL142.

In the Am29PL142, the address sequencer inputs consist of eight external inputs and a portion of the currently held contents in the pipeline register. These bits are wrapped around internally in the chip. (The remaining bits go off chip to control other components in the system.) The test field in the general instruction format tells the sequencer which input to test. The result of this test determines whether the sequencer will process the next instruction in program memory or fetch an instruction from an address specified in the data field from the stack or from an external address specified by the test inputs.

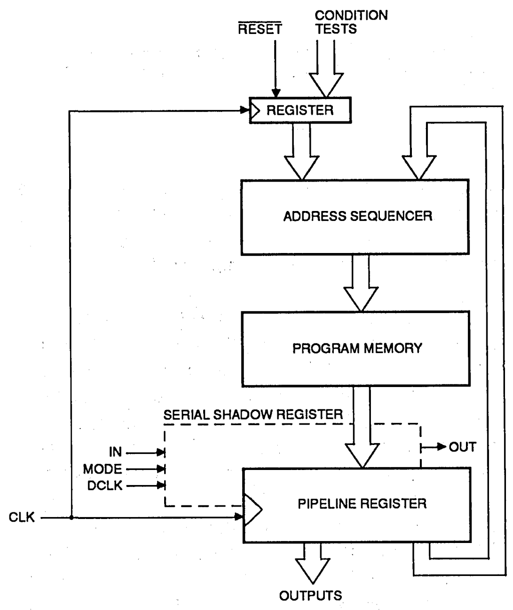


Figure 1-1. Am29CPL100 Block Diagram

Within the address sequencer control logic, a four-to-one program counter multiplexer supplies the next state address (refer to Figure 1-2). This next state address can be one of the following:

- Current address, PC (for repeat or hold instructions)
- The next sequential address from program memory, PC+1 (for sequential and continue instructions)
- The stack (for nesting and repeat loops)
- Output of the GO-TO branch control logic

The Program Counter contains the address of the current state, PC (the current instruction being executed). Allowing the address multiplexer to select the current state as the next state enables execution of loops and wait-until-condition-true type instructions. The FPC can thus simply wait until a particular event becomes true. This function of intelligent state machines is needed to interface with various microprocessors and peripherals. The incremented Program Counter address, PC+1, is the normal next address when no jumps, branches, or subroutine calls are active.

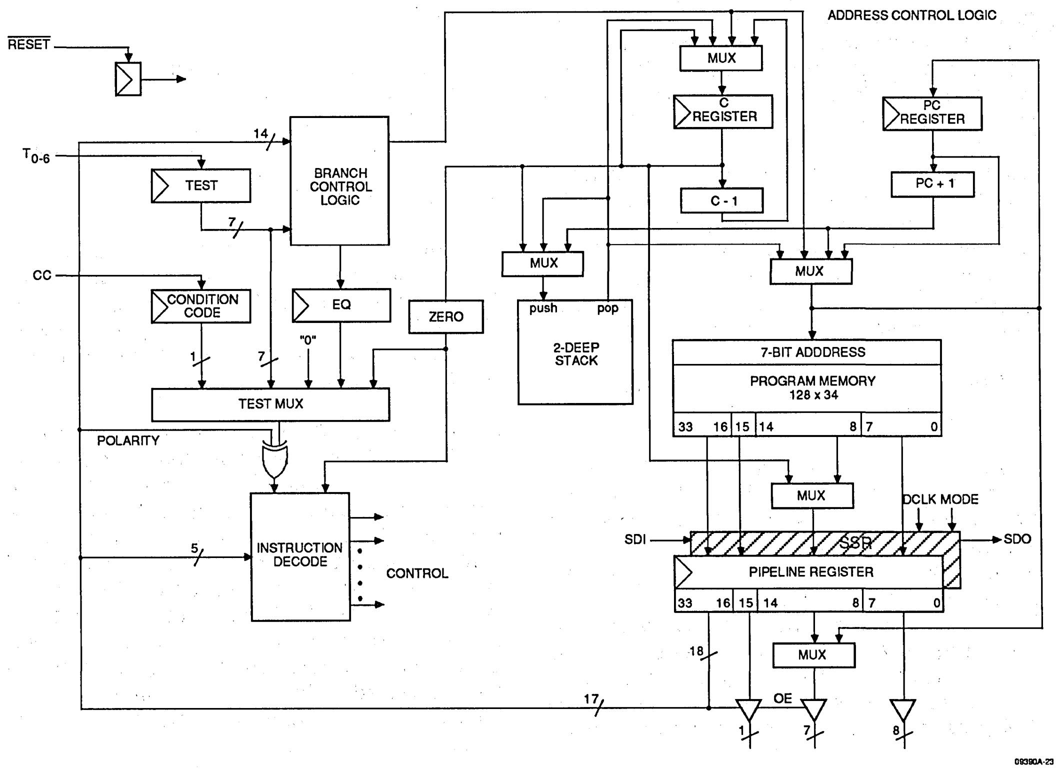


Figure 1-2. Am29PL142 Detailed Block Diagram

The address sequencer logic also incorporates a stack. This output from the stack either loads the counter register multiplexer (discussed below) or program counter multiplexer. The input to the stack is selected by a three-to-one multiplexer, which selects either PC+1 (the return address from subroutine calls), the current top of stack (TOS) contents (for looping on TOS), or the counter register contents (for intermediate storing of counter values).

The counter block is used for timing. It has a counter register (CREG) and a four-to-one multiplexer as the source for the CREG. To perform iterative loops, the controller first loads CREG with the value of the number of iterations required. Every iteration of the loop decrements the count. When the count reaches zero, iterations stop. The zero condition is detected by the zero detect logic on the chip.

The $\overline{\text{RESET}}$ input initializes the FPC, setting the program counter multiplexer to all ones. An additional operating mode allows use of serial shadow register (SSR) diagnostic techniques.

1.2.2 Branch Control/Condition Code Select

The branch control logic provides the address for multiple branching and for conditional statements such as IF-THEN-ELSE. The condition code select logic selects the condition to be tested, which the user can specify for each microprogram instruction. This allows monitoring of both external and internal events. The user-defined microcode can set the polarity control to test on either true or false conditions without the need for external hardware inverters.

The branch control logic implements program jumps using either the data field from the instruction format or the external inputs. Individual inputs can be masked (i.e. set to zero) so that only the desired input can affect the address sequencer's operation. Use of external inputs by the branch control logic supports multiway branching. These external inputs can also be used to preload the counter register.

A flexible instruction set provides powerful conditional branch, multibranching, subroutine, and loop structures. These instructions, explained in the data sheets, fall into six categories:

- Program Branch Instructions (e.g. GOTOPL, GOTOTM, FORK)
- Subroutine Branch Instructions (e.g. CALPL, CALTM, RET)
- Stack Instructions (e.g. PSH, PSHPL, POP)
- Looping Instructions (e.g. LPPL, LPTM, LPSTK)
- Load Counter Instructions (e.g. LDPL, LDTM)
- Miscellaneous Instructions (e.g. DEC, DECPL, DECTM)

1.2.3 Instruction Decode Logic

The instruction decoder decodes the microinstruction, including the opcode field, the polarity bit, the test field, and the data field. The test field specifies the condition code input that will be tested to determine if a branch is to be taken. For conditional branches, if the condition is true (or false if the polarity is set to 1), a branch is taken to the branch address specified in the data field or externally by the test inputs. The output enable bit in the microcode enables the outputs of the FPC.

1.2.4 Program Memory and Pipeline Register

Conceptually, each memory location can be thought of as defining a particular state of the state machine, with each address corresponding to the number of this state. The external test inputs and internal test, the EQ condition (used to determine if the external inputs have a value equal to the constant field in the microcode), are included to allow conditional state transitions. Typical microcode consists of testing one of the test inputs and branching if the condition tested is true.

The program memory stores the program of microinstructions specifying state transitions. Each microinstruction specifies the state of each of the outputs used to control peripherals and other devices. The remaining fields in the microinstruction have been described above. The program memory is programmed using commercially available logic programmers.

The pipeline register associated with the memory contains the microinstruction currently being executed. It allows concurrent execution of the current microinstruction and fetching of the next instruction. Its upper bits form the state sequencing and internal control logic. The low order 16 bits are used as general purpose, user definable control outputs. Of these user controlled bits, the upper eight bits can be three-stated by output enable bit (OE) in the microinstruction. If more than 16 output control bits are needed, Am29PL100 devices can be cascaded quite simply.

The FPC operates in two modes: normal and diagnostic. In the normal mode, a microinstruction is executed for every clock cycle. When the FPC is programmed to use the diagnostics feature, the Serial Shadow Register (SSR) is activated. This provides a simple, straightforward method of in-system testing to isolate problems to the individual IC level.

SSR diagnostics simplify device and system-level diagnostics. To test a chip, an instruction is shifted serially into the SSR and then loaded in parallel into the pipeline. As a result, the instruction is executed and its results are transferred back from the pipeline into the SSR. From there, it may be shifted out for diagnosis.

1.3 INSTRUCTION FORMAT

This section discusses the microinstruction format using the Am29PL142 as an example. For more detailed information on other Am29CPL100 devices, refer to the appropriate data sheets.

Each microinstruction is partitioned into fields. There are two microinstruction formats: the general microinstruction format and the compare microinstruction format. The low order 16 bits in each format contain 16 user-controlled output signals that appear on FPC outputs P[15:0].

In general microinstruction format, the upper 18 bits are assigned as follows:

Bits	Description
16-22	Data (a conditional branch address, test input mask, or counter value)
23-26	Test (specifies which one of eight input signals to use for the condition code)
27	Polarity (specifies whether to test input for true or false)
28-32	Opcode (identifies microinstruction to execute)
33	Output Enable (when set to 0, it 3-states output lines P[15:8])

In the compare microinstruction format, the upper 16 bits are assigned as follows:

Bits	Description
16-22	Data (a 6-bit mask for masking the T[5:0} inputs)
23-29	Constant (specifies a 6-bit constant for comparison with T*M for the condition code)
30-32	Opcode (identifies microinstruction to execute)
32	Output Enable (when set to 0, it 3-states output lines P[15:8])

1.4 Am29CPL100 SOFTWARE SUPPORT

Designing complex state machines and intelligent controllers requires good software support. The Am29CPL100 family is supported by assemblers and simulators. The assembler generates a JEDEC fuse map from source code programs written using the high-level Am29CPL100 instruction set. This JEDEC fuse map is used by commercially available logic programmers to program Am29CPL100 devices. The simulator is used to perform logic simulation of Am29CPL100 programs.

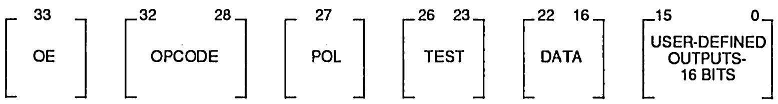
1.4.1 PL14X Assembler

The PL14X Assembler converts design specifications written in a symbolic language into a JEDEC fuse map which can be used by other modules such as the simulator and commercially available logic programmers.

The assembler allows data to be defined as bytes or words, permits forward label references, and allows assignment of values to bits in binary, octal, decimal, and hexadecimal format.

High level language constructs, such as IF-THEN-ELSE and WHILE, are directly supported by the assembler providing program structure and clear documentation for the designer.

The assembler is described in detail in the PL14X assembler documentation.



WHERE:

OE	= SYNCHRONOUS OUTPUT ENABLE FOR P[15:8].
OPCODE	= A 5-BIT OPCODE FIELD FOR SELECTING ONE OF THE 27 SINGLE DATA-FIELD INSTRUCTIONS.
POL	= A 1-BIT TEST CONDITION POLARITY SELECT. 0 = TEST FOR TRUE (HIGH) CONDITION. 1 = TEST FOR FALSE (LOW) CONDITION.
TEST	= A 4-BIT TEST CONDITION SELECT

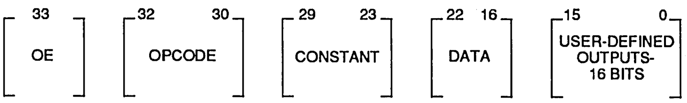
TEST [3:0]	CONDITION INPUT UNDER TEST
0000	T[0]
0001	T[1]
0010	T[2]
0011	T[3]
0100	T[4]
0101	T[5]
0110	T[6]
0111	CC
1000	EQ
1001	CREG ZERO
1010-1111	UNCOND [0]

THE POLARITY BIT POL IN AN INSTRUCTION ALLOWS THE USER TO TEST FOR A PASS/TRUE OR FAIL/FALSE CONDITION AS SHOWN IN TABLE 2. AN UNCONDITIONAL TRUE IS SET BY SELECTING UNCOND AND POL = 1.

DATA = A 7-BIT CONDITIONAL BRANCH ADDRESS, TEST INPUT MASK, OR COUNTER VALUE FIELD DESIGNATED AS PL IN INSTRUCTION MNEMNONICS.

Figure 1-3. Am29PL142 General Instruction Format

The special two data field comparison instruction is shown below:



WHERE:

OE	= SYNCHRONOUS OUTPUT ENABLE FOR P[15:8].
OPCODE	= COMPARE INSTRUCTION (BINARY 100).
CONSTANT	= A 7-BIT CONSTANT FOR EQUAL COMPARISON WITH T*M.
DATA	= A 7-BIT MASK FIELD FOR MASKING THE INCOMING T[6:0] INPUTS.

Figure 1-4. Am29PL142 Comparison Instruction Format

1.4.2 PL14X Simulator

Device simulation is based on a test vector file, generated from the test vectors specified by the designer. The PL14X simulator uses the JEDEC fuse map file (generated by the PL14X assembler) and the test vector file as its inputs. The simulator generates computed output signals that are compared with expected output values as specified in the test vector file. A printout of the output shows the difference if any.

The simulator also provides an interactive mode allowing the designer to interactively preload or change any or all of the internal registers of simulated Am29CPL100 devices. Single-step and breakpoints provide further control. For details, refer to the PL14X Simulator documentation.

1.5 AN OVERVIEW OF THIS TECHNICAL HANDBOOK

Chapter 2 is an Am29PL100 tutorial.

Chapter 3 presents reprints of articles written about the Am29PL141. The first article is an overview discussion of the Am29PL141, its architecture and applications. The second article discusses a VME bus controller designed using the Am29PL141.

Chapter 4 provides a simple example of an Am29PL141 application. It is a coffee machine controller. This example shows not only the hardware but also the microprogram required.

Chapter 5 describes the realistic use of an Am29PL141 as an interface for the DEC PDP-11 Unibus. The complexity of Unibus handshaking is such that microprogramming is a reasonable design technique, but use of a separate sequencer, control memory, and pipeline register is not economical. Since the FPC contains a sequencer, memory, and pipeline, an interface for the DEC PDP-11 Unibus can be readily designed using the Am29PL141 FPC. It fits this class of problem rather well. The PDP-11 was chosen for this example because it has a well documented protocol which is familiar to many engineers.

Chapter 6 describes the use of an Am29PL141 as a controller for the DEC Q-Bus. The problem addressed is to design an interface between the Q-Bus and a generic device to allow the following operations:

- DATI/DATO with device as slave
- Device interrupt request (single level)
- Device direct memory access request
- DATI/DATO with device as master

The control logic is implemented using the Am29PL141 FPC. Its microprogram implements a state machine to control both device and Q-Bus handshaking.

Chapter 7 describes the use of the Am29PL141 as a dual port memory arbitrator in a Starlan system. The Am29PL141 controls the DMA transfers to and from the relatively slow speed communication lines freeing the CPU to perform other tasks.

Chapter 8 describes the use of an IBM/PC to run diagnostic tests on a device containing a Serial Shadow Register (SSR). The Am29PL141 controls the flow of data to and from the SSR.

Chapter 9 describes an Am29PL141-QIC-02 and SCSI interface. This interface links tape drives with a CPU. It permits the backup of large hard disk drives on quarter-inch magnetic tape.

Chapter 10 describes a high speed DMA controller using the Am29PL141.

The appendixes include the JEDEC Standard Number 3; the QIC-02 and SCSI timing diagrams; References; Glossary; Am29PL100 data sheets; and an index.

CHAPTER 2

AM29PL100 TUTORIAL

Written by Art Goldstein

The purpose of this tutorial is to present background information on the Am29PL100 family of parts to enable designers to use these devices in their designs.

The Am29PL100 are high-speed logic devices that are programmable by instructions rather than Boolean equations. From this point of view, the Am29PL100 family represents a new trend and will, over time, enjoy wide acceptance because of the simplicity of its approach.

Historically, the Am29PL100 family evolved out of bit-slice design technology. At the heart of these designs is the concept of microprogramming that was first proposed in the early 1950s by M.V. Wilkes as a technique for simplifying the design of a computer-control unit.

Nominally, the design of the control unit seems to have little to do with the bulk of sequential logic design. In fact, however, the control unit can be regarded as a general model for a broad range of logic design problems. Because of this, microprogramming represents, in essence, a logic design technique that offers many advantages over designs based strictly on Boolean equations and state diagrams.

Microprogramming differs very little from normal assembly-language programming. Both have similar features in terms of instruction set format and program structures. The difference is that microprogramming supports the design of hardware while an assembly language program is more often associated with the manipulation of abstract data.

In this age of the microprocessor, it is well to keep in mind that the Intel 4004, generally considered the first commercially available microprocessor, was used more as a replacement for random logic than as a general-purpose processor. Indeed, the same can be said of 8-bit processors. It has been only with the advent of the 16- and 32-bit processors that the microprocessor has flowered into a general-purpose computing device used to supplant minicomputer and even mainframe machines.

Because of this history, the concept that software techniques can be used to implement hardware should come as no surprise. Microprogramming, because of its association with bit-slice design, might seem mysterious to the uninitiated, but it is indeed both simple and straightforward, as this tutorial will show.

This tutorial consists of two sections. Section 2.1 is devoted to microprogramming and presents an overview of the subject as it relates specifically to the Am29PL100 family. In addition, Section 2.1 includes the design of a computer control unit that not only sheds light on how microprogramming works but also serves as the foundation for the design of the Am29PL100 family itself.

Section 2.2 presents five Am29PL100-based designs. Within the course of this section, the Am29PL100 microinstructions are explained and used to implement a wide spectrum of hardware structures.

2.1 INTRODUCTION TO MICROPROGRAMMING

This section introduces the major ideas of microprogramming by working through the design of a microprogrammed control unit for a very simple computer system. In this way, we not only obtain an understanding of microprogramming, but also learn about the structure of the Am29PL100 field programmable controller and why it came about. Later family members, such as the Am29CPL142 and Am29CPL144 are very close architecturally to the Am29PL141, and the concepts discussed will apply to the family in general.

The reason for focusing on the design of a computer control unit is twofold. First, microprogramming evolved specifically as a means of simplifying the design of this part of the computer. Second, the control unit serves as a useful model for many complex digital networks.

The point to bear in mind is that microprogramming is a general logic design technique and is a useful alternative to Boolean equations and state diagrams in many circumstances. As its name implies, microprogramming involves software techniques. While some knowledge of machine language programming is useful, we do not assume any particular background, but instead develop the pertinent techniques as the need arises.

2.1.1 Simple Computer System

Figure 2-1 shows a block diagram of a simple computer system capable of processing the 16 instructions listed in Figure 2-2. The computer system consists of these major components:

Dual-ported register file (RF) — The register file contains eight 16-bit registers. The contents of any two registers are simultaneously available at the YA and YB output ports by supplying the appropriate addresses to Port-A and Port-B address lines and holding the WRRF line high. Data is written into a register by specifying the register address on the Port A address lines and pulsing the $\overline{\text{WRRF}}$ line low.

Arithmetic logic unit (ALU) — The ALU performs the eight operations listed in Table 2-1. Each operation requires two 16-bit operands with the result appearing on a 16-bit output port. For arithmetic operations, three status lines indicate whether an overflow occurred or a result was greater than or equal to zero.

Memory subsystem — The memory subsystem consists of a 4K-word by 16-bit memory array, a memory address register (MAR), an input data register (IDR) and an output data register (ODR). The address in the MAR comes either from the program counter (see the description of register R7 below for further details) or from an address specified in an instruction. The signal READ determines whether data is written to or read from the memory.

Control unit (CU) — The control unit activates a sequence of control signals to implement the instructions fetched from memory. (See Section 2.3 for a detailed description of each control signal.)

As shown in Figure 2-2, nearly half the instructions operate upon the contents of registers in the register file. These eight registers, designated R0 through R7, are referenced by the 3-bit source or destination field contained within the instructions. (See Section 2.4 for a detailed description of each instruction.)

Three of the registers have special meanings that the programmer must take into consideration:

R7 — This register is the program counter. It is incremented during the execution phase of every instruction and is used as the source for the MAR unless the instruction causes a transfer of control.

R6 — This register is the stack pointer. The CALL SUBROUTINE instruction uses this register to store the return address on the stack. The RETURN instruction uses this register to restore the program counter with the return address.

R0 — This register is implied as the source and destination register in the STORE and LOAD instructions, respectively. The BRANCHN and BRANCHZ instructions test this register to determine if the branch is taken.

2.1.2 Design of Control Unit

The control unit directs the activity of the elements of this system. It responds to instructions fetched from memory and activates a sequence of control signals to implement the instructions. To show how this works, let's examine the sequence of events that takes place in fetching, decoding and executing the ADD instruction shown in Figure 2-2.

In our example, we want to add the contents of

OPERATION	ALU CODE	FUNCTION PERFORMED*
SUB	000	A - B
ADD	001	A + B
INC	010	A + 1
DEC	011	A - 1
AND	100	A • B
OR	101	A + B
SHFTL	110	SHIFT A LEFT ONE POSITION
SHFTR	111	SHIFT A RIGHT ONE POSITION

* A REFERS TO OPERAND ON INPUT PORT A
B REFERS TO OPERAND ON INPUT PORT B

RESULT OF OPERATION APPEARS ON OUTPUT PORT.

Table 2-1. ALU Operations

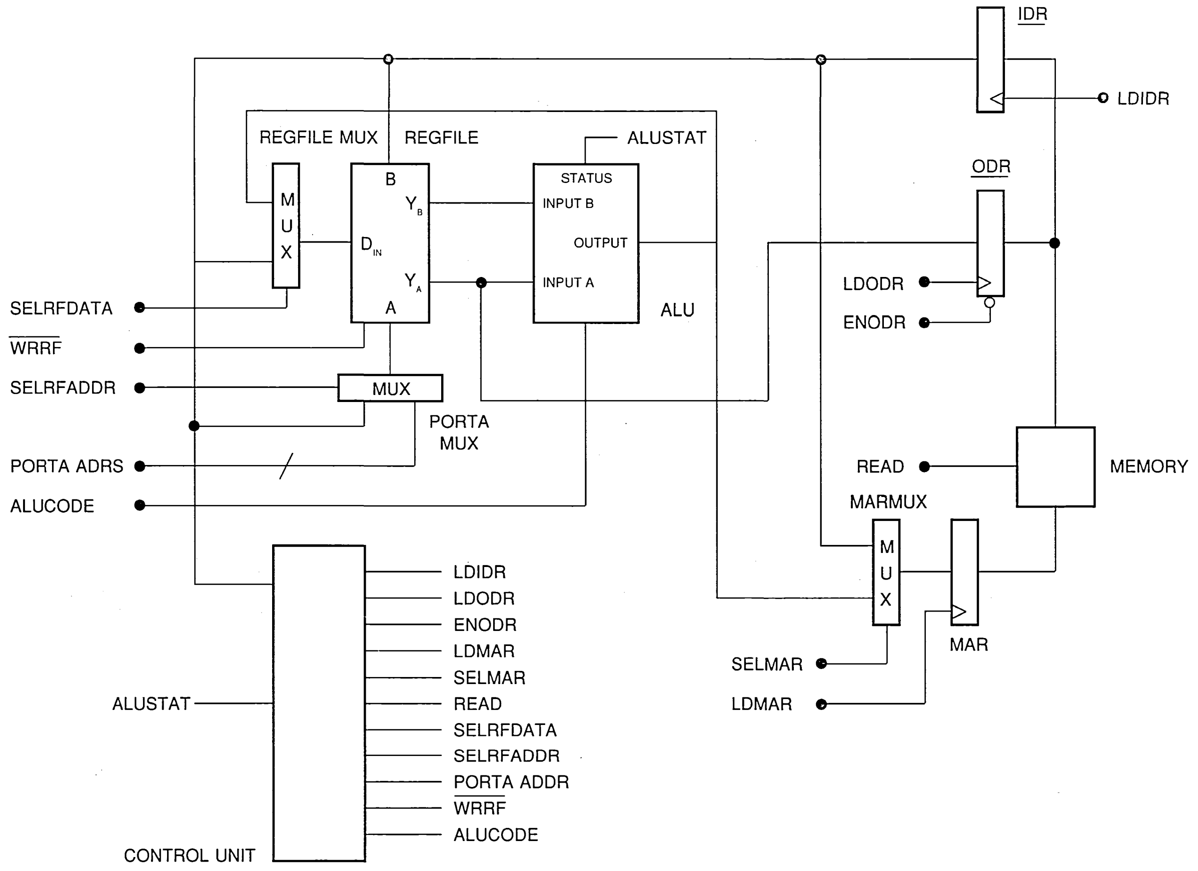


Figure 2-1. Simple Computer System

	OPCODE†			
SHIFTL*	0	COUNT	▨	▨
SHIFTR*	1	COUNT	▨	▨
ADD	2	RD	RS	▨
SUB	3	RD	RS	▨
AND	4	RD	RS	▨
OR	5	RD	RS	▨
JMP	6	ADDRESS		
CALL SUBROUTINE	7	ADDRESS		
RETURN	8	ADDRESS		
BRNCH 1**	9	ADDRESS		
BRNCH 2**	A	ADDRESS		
LOAD*	B	ADDRESS		
STORE*	C	ADDRESS		
INC	D	RD	▨	▨
DEC	E	RD	▨	▨
MOV	F	RD	RS	▨

RD: SPECIFIES DESTINATION REGISTER
RS: SPECIFIES SOURCE REGISTER

* REGISTER RO IMPLIED AS DESTINATION OR SOURCE
** REGISTER RO IMPLIED AS REGISTER TESTED
† OPCODE GIVEN IN HEX

Figure 2-2. Instruction Set and Format

register R2 with that of register R3 and replace register R2 with the result. Here are the steps that take place:

- Fetch instruction.

  Load MAR with address of instruction.

  Issue READ signal to external memory bus.

  Latch data from memory bus into IDR.
- Decode instruction.

  Extract OPCODE field from IDR.

  Determine operation to be performed by examining OPCODE.
- Execute instruction.

  Issue code for ADD to ALU.

  Wait for operation to take place.

  Issue $\overline{\text{WRRF}}$ pulse to register file to write result back to register R2.

  Increment program counter R7.
- Go back to the first step to continue processing program.

Figure 2-3 illustrates the timing diagram that reflects this activity. Using this timing diagram, we can design a logic network to implement the sequence of steps enumerated above. Of course, this network supports only the ADD instruction. Creating a logic network supporting all 16 instructions is considerably more complex. Furthermore, adding an additional instruction or modifying an existing one can be a fairly complicated procedure entailing a redesign of the entire network, not just a small portion of it.

2.1.3 Microprogramming Approach

Confronted with this state of affairs, M.V. Wilkes suggested an alternative approach whose purpose was to design "the control circuits of a machine which is wholly logical and which enables alterations or additions to the order code to be made without ad hoc alterations to the circuits." (Note that the term "order code" was an earlier way of referring to OPCODE.)

The essence of Wilkes' idea is that the output of a sequential logic network can be regarded as a series of binary words. The individual bits of these words serve as control signals that are used to direct the operation of the computer. Viewing the words in this way, we can just as well store them in a read-only memory (ROM) and read out the contents of the ROM, clock by clock, to obtain the desired output pattern. Because the output patterns of the sequential logic network and the ROM-based approach are identical, these two approaches can be regarded as interchangeable.

To illustrate these ideas, we will work through the steps to create the sequence of control signals needed to implement the ADD instruction described above. The circuit of Figure 2-4 is used for this purpose. As shown, the ROM is partitioned into two parts. Fifteen of the output bits are used as our control signals, while the remaining four bits are fed back to the address register. These bits, together with the OPCODE, are used to address the ROM.

The timing diagram of Figure 2-3 has been redrawn in Figure 2-5 to show how to derive the data we want to place in the ROM. As shown, we treat the timing information like a series of binary numbers that change on a clock-to-clock basis. The top line corresponds to the lowest-order bit with subsequent lines spanning the higher order bits. Working in this way, we construct the data in Table 2-2. Here, the first line contains the equivalent binary number corresponding to clock period 1 followed by the numbers corresponding to subsequent clock periods. Notice that clock period 1 is the first clock period after the ADD instruction has been clocked into the IDR.

Table 2-3 lists the part of the ROM contents that we are using in our example. The low-order hex digit contains the NEXT ADDRESS that is used to sequence through the ROM addresses, and the remaining four digits contain our desired control information. Notice that the ROM contents are at addresses 20 through 28 hex. We derive this ROM address range by combining the low-order four bits from the ROM—the NEXT ADDRESS—with the OPCODE field. From Figure 2-1, we see that the OPCODE for the ADD instruction is 2.

Now let's follow the activity starting from the point when the ADD instruction is fetched and residing in the IDR. As the address register is clocked, the contents of the ROM starting from location 20 is clocked out, followed by the contents of locations 21 through 28. If we trace the activity of the control bits during this time, we wind up with the timing diagram of Figure 2-4, which is the desired result.

Each word in the ROM (the "control" ROM) is called a microinstruction, and a sequence of microinstructions is called a microprogram.

The architecture we arrived at employing microprograms has a hierarchy of programs. The programs that reside in the system memory are normally called "macro" programs (or assembly level programs). The program that directly controls the hardware and is stored in the control store and is invisible to the general user is the microprogram. Macroprograms are said to be made up of macroinstructions. Each macroinstruction, as we have seen, has an OPCODE field and data fields.

CLOCK PERIOD	VALUE (HEX)
1	0865
2	1864
3	1064
4	1F64
5	2FE4
6	27E4
7	2FEC
8	0864
9	0864

Table 2-2. Binary Equivalent of Timing Diagram for Add Instruction

ROM ADDRESS	ROM DATA
20	08651
21	18642
22	10643
23	1F644
24	2FE45
25	27E46
26	2FEC7
27	08648
28	08640

Table 2-3. ROM Values Corresponding to Add Instructions

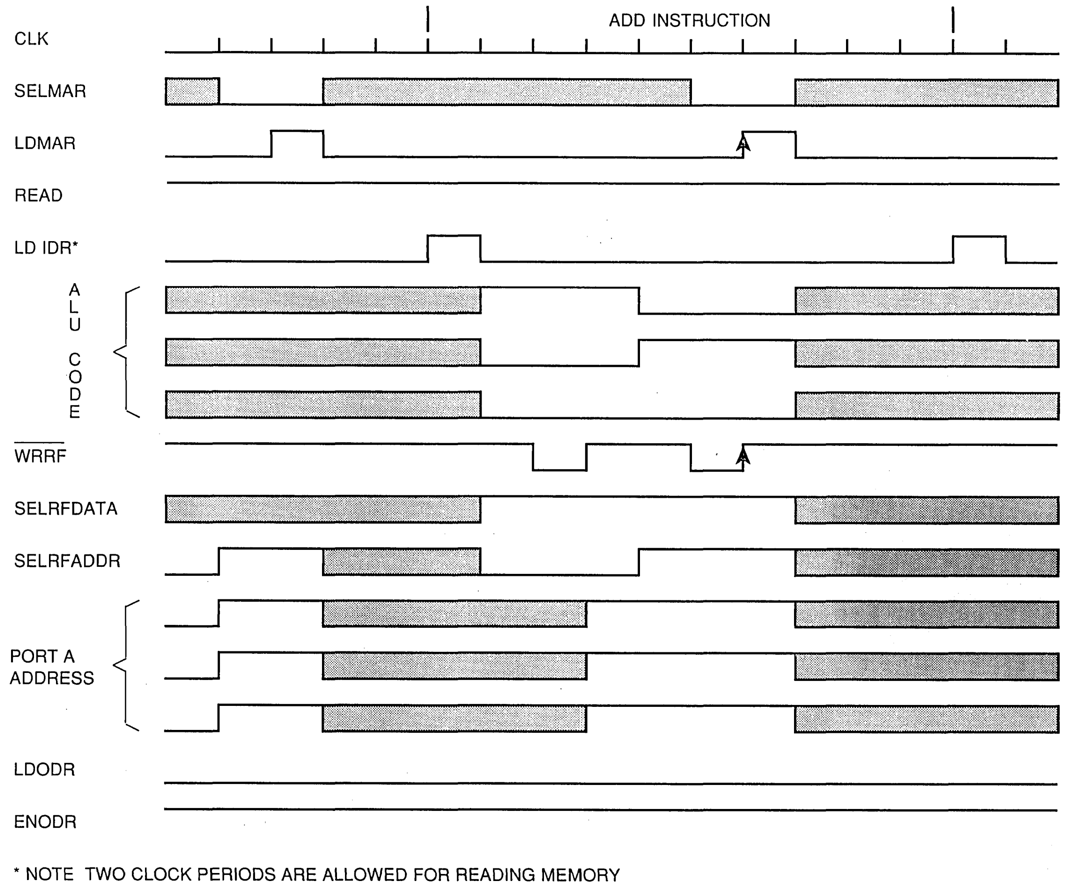


Figure 2-3. Timing Diagram for Fetching, Decoding and Executing Add Instruction

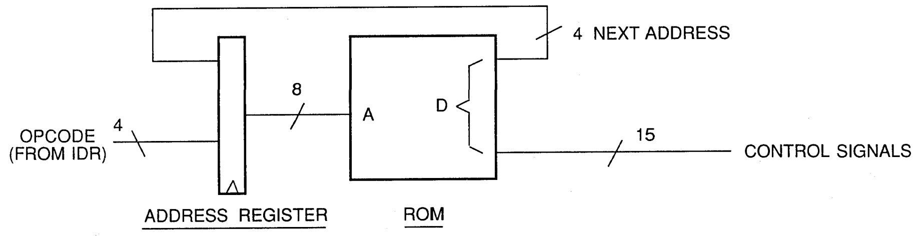


Figure 2-4. ROM-Based Control Unit

For each macroinstruction, there is a sequence of microinstructions that need to be executed to implement it. Each of these small sequences is normally called a microroutine, and the combination of all these microroutines is normally called the microprogram. Microroutines are sometimes referred to as microprograms.

Because our computer system has 16 macroinstructions or operations (see Figure 2-2), we have, correspondingly, 16 microroutines, each of which is used to decode and execute the corresponding macroinstruction and fetch the following macroinstruction. Assuming the NEXT ADDRESS (micro) field is initially zero, we use the (macro) OPCODE to point to the starting address of the appropriate microroutine (see Figure 2-6).

A microinstruction consists of two parts. The first part contains information about the sequencing of microinstructions (NEXT ADDRESS field), and the second part consists of a set of control signals that carry out several activities in parallel (see Figure 2-7).

Within one macroinstruction, we may use some of the control bits to specify an ALU operation, another to specify a READ and still another to perform some other activity. In a sense, each one of these output fields represents a MICROOPCODE because it specifies an action to take place. In this way, a microinstruction may consist of several MICROOPCODEs that are acted upon in parallel.

A major difference between macro- and microinstructions is that a macroinstruction consists of one OPCODE, while a microinstruction consists of several MICROOPCODEs.

2.1.4 Microsubroutines

Given the 16 instructions that our simple computer system can process, we use 16 corresponding microprograms. Each microprogram contains microinstructions for fetching the next instruction. Because of the 16 microprograms, these microinstructions are repeated 16 times. Clearly, this is wasteful, especially in the case of a more complex computer that has tens of instructions. A more efficient way to have the active microprogram jump to a separate microprogram that carries out the steps for fetching the next instruction and then return to the point where it left off. This procedure is called a microsubroutine and is similar to the subroutine of a program. However, our simple address register-ROM combination is not adequate for this task. To accomplish the above task, we need these new facilities:

- Microsubroutine register for storing the next microinstruction to be executed upon return from the microsubroutine.
- Mechanism for loading the address register with the contents of the microsubroutine register so that the initiating microprogram can proceed. Figure 2-8 shows a circuit that allows the execution of microsubroutines as well as sequential microinstructions. The address multiplexer funnels three address sources to the address register.
- NEXT ADDRESS field. The NEXT ADDRESS field has been widened to allow the microsubroutine to be located anywhere in the ROM.
- Microsubroutine register. This register holds the address of the next sequential instruction to be executed upon return from a microsubroutine.

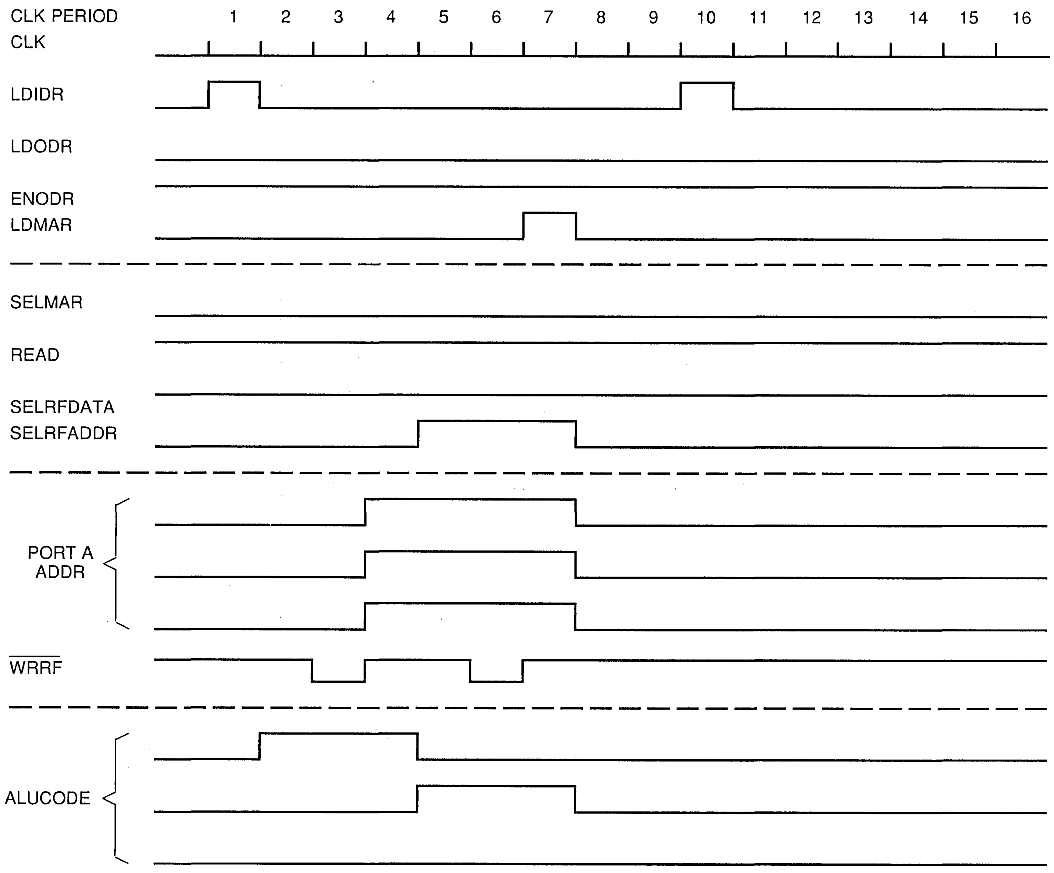


Figure 2-5. Timing Diagram for Obtaining ROM Values

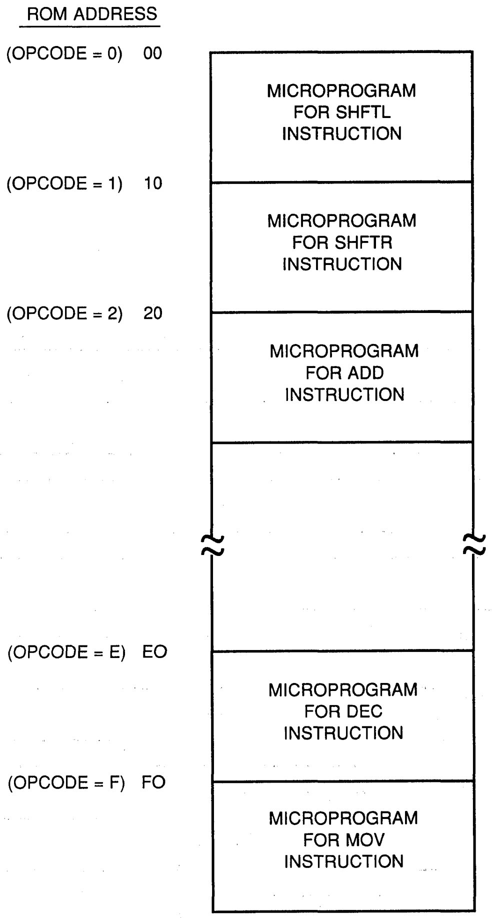

ROM ADDRESS
(OPCODE = 0) 00
MICROPROGRAM FOR SHFTL INSTRUCTION
(OPCODE = 1) 10
MICROPROGRAM FOR SHFTR INSTRUCTION
(OPCODE = 2) 20
MICROPROGRAM FOR ADD INSTRUCTION
(OPCODE = E) EO
MICROPROGRAM FOR DEC INSTRUCTION
(OPCODE = F) FO
MICROPROGRAM FOR MOV INSTRUCTION

Figure 2-6a. Address Space Map for ROM

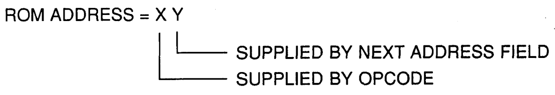


Figure 2-6b. Composition of ROM Address

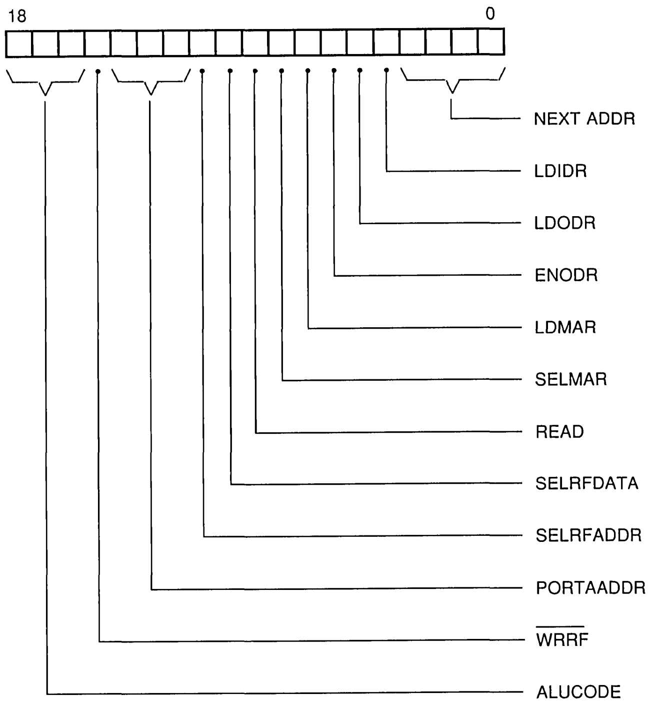


Figure 2-7a. Microword Functions

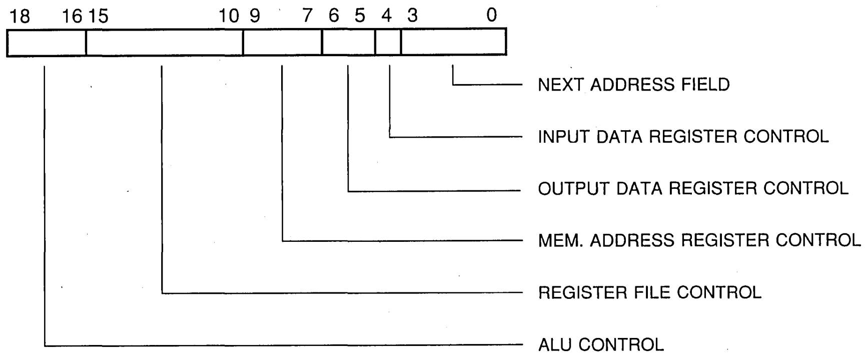

18
16
15
10
9
7
6
5
4
3
0
NEXT ADDRESS FIELD
INPUT DATA REGISTER CONTROL
OUTPUT DATA REGISTER CONTROL
MEM. ADDRESS REGISTER CONTROL
REGISTER FILE CONTROL
ALU CONTROL

Figure 2-7b. Microword Format

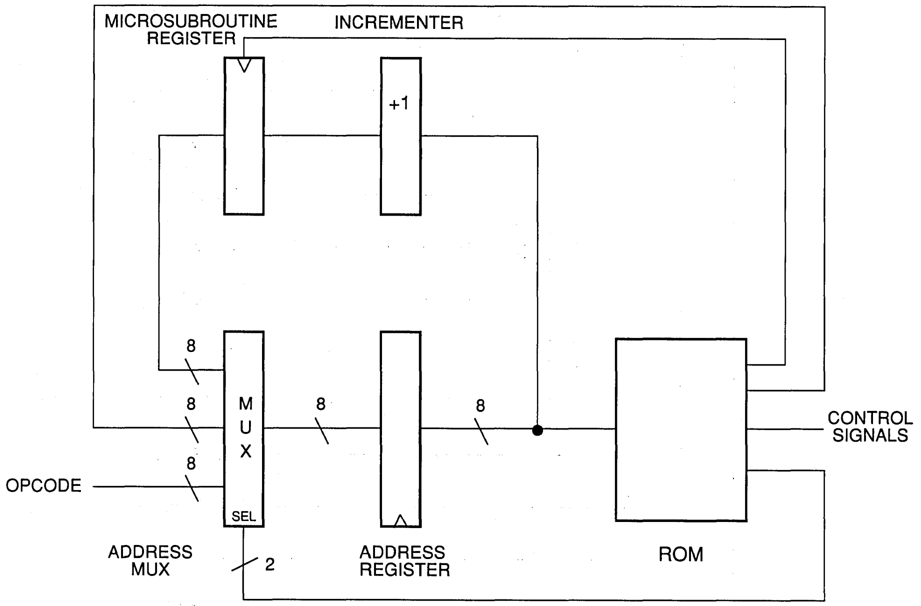


Figure 2-8. Circuit for Accommodating Microsubroutine

The microsubroutine register receives the output of the address register incremented by one. This allows us to proceed with the next sequential microinstruction upon completion of a subroutine.

- OPCODE field. This source is similar in function to the circuit described previously in Figure 2-4. It differs in that the low-order four bits are forced to zero, while the upper four bits are derived from the OPCODE field as before.

The key to the operation of this new circuitry is the ability to control the selection inputs of the multiplexer (mux). We accomplish this control by adding an entirely new field to the microinstruction as shown in Figure 2-9. In operation, this mux control field starts to take on some of the behavior of an OPCODE in a program, as shown in Figure 2-10 where we give symbolic names to the binary codes comprising the mux select lines.

Figure 2-11 shows how to use the microsubroutine facility to call a microsubroutine from any microprogram. As shown, each calling microprogram executes a JMP FETCH microinstruction. As a result the microsubroutine register receives the content of the address register incremented by one and control is passed to the address specified symbolically as FETCH (the beginning of the microsubroutine). Execution of the microsubroutine now proceeds until the last microinstruction, an RTS, is executed. This enables the calling microprogram to resume execution at the address following the JMP FETCH instruction by transferring the contents of the microsubroutine register back to the address register.

2.1.5 Solving Two Fundamental Problems

A careful examination of Figure 2-8 reveals two fundamental problems:

- Glitches on control signals. After a new address is sent to the ROM, glitches may occur on the control signals during the access time of the ROM.
- Delay of one clock period on OPCODE input. Because the OPCODE field is extracted from the IDR, a delay of one clock period is introduced until the OPCODE is clocked into the address register.

The circuit of Figure 2-12 corrects these problems by adding a register at the output of the ROM—called a pipeline register—and addressing the ROM with the output of the mux rather than the address register. Notice that this arrangement still preserves the functionality of the circuit in Figure 2-8.

2.1.6 Microprogram Counter

If we connect the output of the address incrementer to another input of the mux, an unexpected benefit results. With this arrangement, we can execute sequential microinstructions without explicitly defining the next address in the NEXT ADDRESS field of the microinstruction. The combination of address register and incrementer gives us a microprogram counter that behaves in the same manner as a program counter for a program. In addition to the microprogram counter, however, we still need the NEXT ADDRESS field to allow us to transfer control to any location within the ROM (see Figure 2-13).

2.1.7 Implementing the Branch Instructions

So far we have created a powerful structure for executing microinstructions. We can execute sequential microinstructions, call microsubroutines and jump to any arbitrary location in ROM. However, we cannot implement the macro BRANCH instructions because we lack the ability to alter the sequence of microinstruction execution and thus the sequence of macroinstruction execution, in response to the state of the ALU status lines.

For example, when the macro BRANCHZ instruction executes, transfer of control passes to the instruction located at the address specified in the ADDRESS field when the content of register R0 is zero. Otherwise, macroprogram execution continues with the next sequential macroinstruction. Our control unit must therefore be able to load the MAR with the content of either the macroprogram counter or the IDR, depending upon the state of the ZERO ALU status line. This means incorporating a conditional branch facility into our control unit. Because microcode instructions execute the conditional macrocode instructions, the microcode instructions themselves need conditional operation capability.

Let's take a look at Figure 2-14, which is the circuit of Figure 2-13 with an added block, EXECUTION LOGIC. The inputs to this block are the mux select lines from the pipeline register, an external condition and a new bit from the pipeline register. This bit, which we will call a conditional JUMP select line, allows us to alter the sequence of microinstruction execution depending upon the state of the external condition input. Notice also that we have relabeled the NEXT AD-

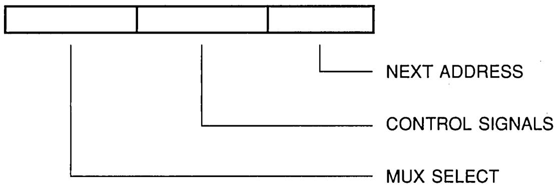


Figure 2-9. Microinstruction Format with MUX Select Lines Added

MUX SELECT	SYMBOLIC REFERENCE	ACTION
00	JMPOP	TRANSFER CONTROL TO ADDRESS POINTED TO BY OPCODE
01	JMP	PROCEED WITH MICROINSTRUCTION POINTED TO IN NEXT ADDRESS FIELD
10	RTS	RETURN FROM MICROSUBROUTINE BY PROCEEDING WITH ADDRESS POINTED TO BY MICROSUBROUTINE REGISTER

Figure 2-10. Symbolic Interpretation of MUX Select Lines

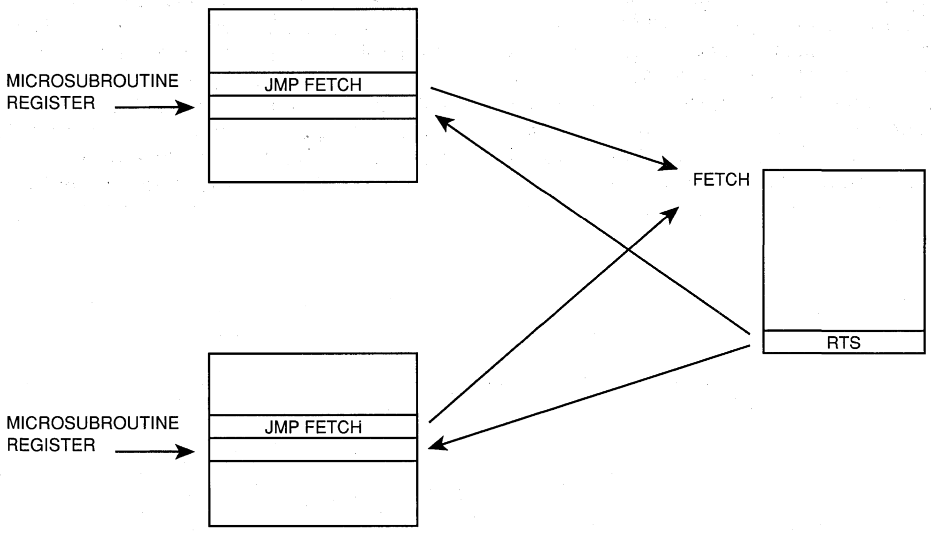


Figure 2-11. Calling A Microsubroutine

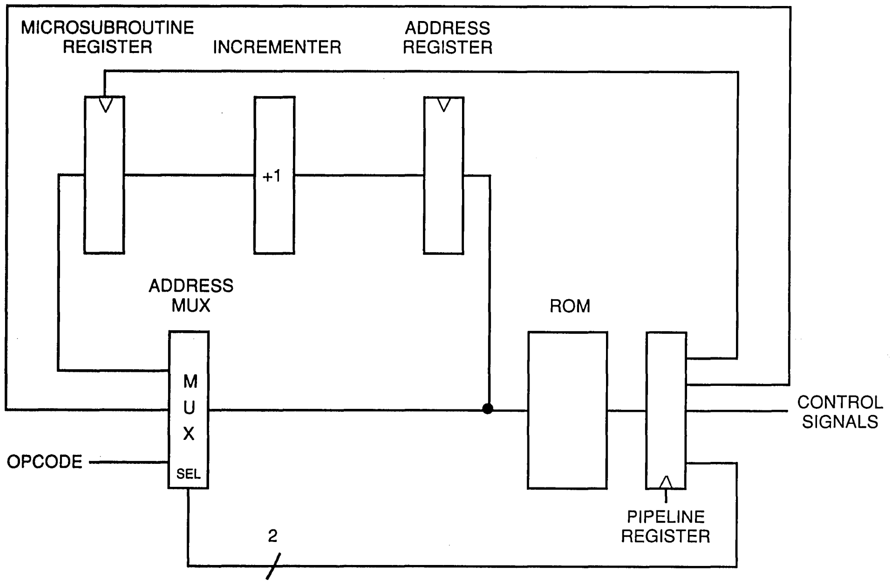


Figure 2-12. Circuit to Correct Glitches on Control Signals

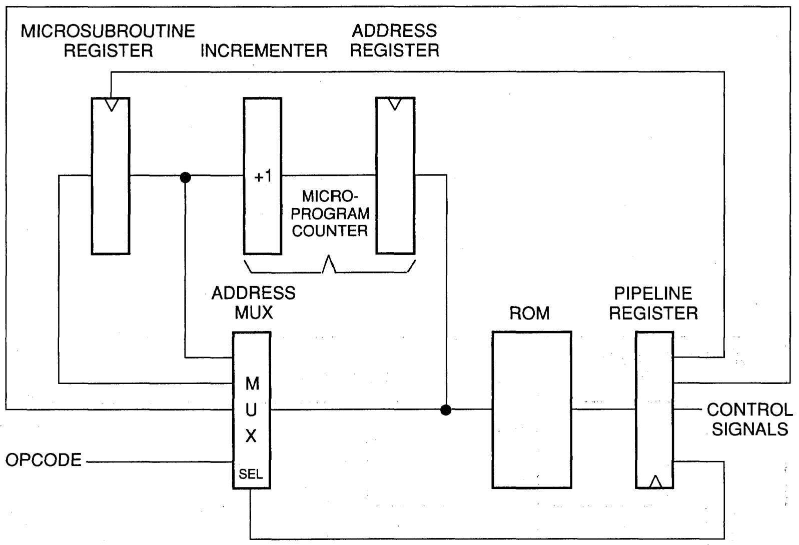


Figure 2-13a. Adding Microprogram Counter

MUX SELECT	SYMBOLIC REFERENCE	ACTION
00	JMPOP	TRANSFER CONTROL TO ADDRESS POINTED TO BY OPCODE
01	CALL MICROSUB OR JMP	TRANSFER CONTROL TO ADDRESS POINTED TO IN NEXT ADDRESS FIELD
10	RTS	RETURN FROM MICROSUBROUTINE BY PROCEEDING WITH ADDRESS POINTED TO BY MICROSUBROUTINE REGISTER
11	CONT	PROCEED WITH ADDRESS POINTED TO BY MICROPROGRAM COUNTER

Figure 2-13b. Interpretation of MUX Select Lines

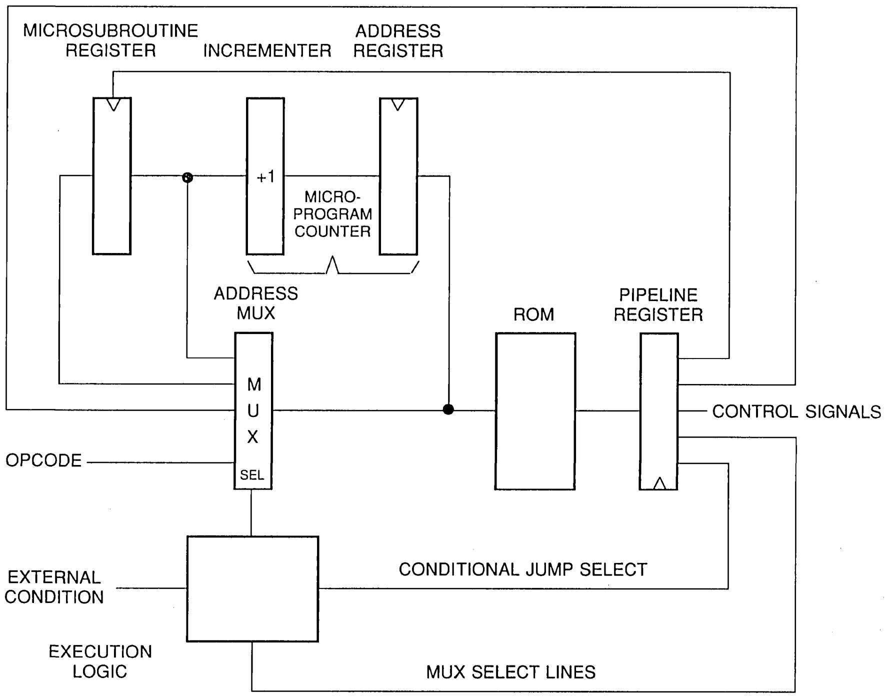


Figure 2-14a. Adding Conditional Branching Capability to Control Unit

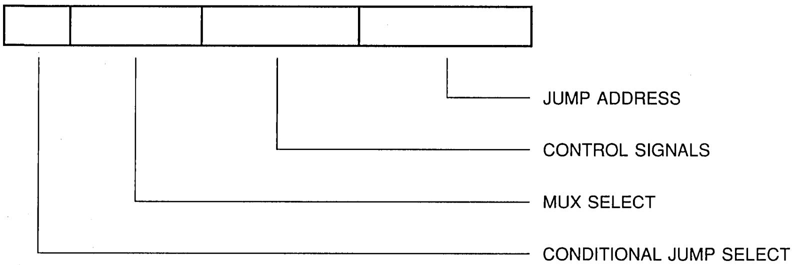


Figure 2-14b. Microword Format

DRESS field as JUMP ADDRESS in Figure 2-14B because we need this field only to specify jump addresses. This reflects the fact that the micropro-gram counter allows us to execute sequential micro-instructions.

Figure 2-15 provides a summary of the functions performed by the EXECUTION LOGIC block. Notice, in particular, that when the conditional jump select line is high and the external condition is low, the control unit continues execution with the following microin-struction. Alternatively, if the jump select line is high and the external condition is high, we take the course of action specified in Figure 2-13b.

By adding the conditional jump capability, we can now implement the BRANCHZ macroinstruction. To do this, we use the ZERO ALU status line as our external condition and use a microinstruction that implements the conditions specified symbolically as CONDJMP in Figure 2-15. This microinstruction resides in the microprogram that executes the BRANCHZ macroin-struction. In that microprogram, a decision is made to load the MAR with the content of the program counter or with the address contained in the IDR. The JUMP ADDRESS field of the CONDJMP microinstruction therefore contains the address of a sequence of microinstructions that load the MAR with the content of the IDR while the microinstructions following the CONDJMP microinstruction load the MAR with the content of the program counter. (See Figure 2-16.)

To this point, our conditional branch capability is restricted to one external input, the ZERO ALU status line. But we must accommodate the other ALU status lines as well.

Figure 2-17 shows the addition of a mux to our control unit. The inputs to the mux are the ALU status lines, and the mux select lines are controlled by two addi-tional bits that have been added to the microinstruc-tion. This structure now allows us to JUMP condition-ally on any one of the ALU status lines. In particular, we can now implement both the BRANCHN and BRANCHZ macroinstructions.

2.1.8 Implementing the Shift Instructions

From Figure 2-2 we see that our control unit has the facilities for decoding and executing every instruction except the SHIFT instructions. For these instructions, we must shift the number of places specified in the instruction, even though the ALU allows only one shift per clock cycle. This requirement implies that we must repeat the microinstruction for executing a shift the number of times specified by the shift count. To do so, however, we must add a counter to our control unit and incorporate new microinstructions for performing these functions:

- **Load** the counter with the shift count.
- Decrement the counter.
- Branch to a location specified in the NEXT ADDRESS field if the counter is not zero.

Let's take a look at part of a microprogram for execut-ing a SHIFT instruction. Because we have not yet incorporated these functions into the control unit, we present the microprogram using English statements that describe the action we want the microinstruction to perform:

- Load counter with shift count
- LOOP: Present code for shift to ALU
- Decrement counter
- If counter not equal to zero, branch to LOOP
- Call FETCH

Each line of this microprogram corresponds to one microinstruction. Rather than specify an absolute location for this microprogram, we use the label LOOP as a placeholder. The last line of the microprogram refers to the microsubroutine structure we described previously. In this case, we request the operation by using "Call" and denote the beginning location of the microsubroutine for fetching the next instruction by the label FETCH.

Figure 2-18A incorporates a counter into the control unit that allows us to implement the SHIFT instruc-tions. As shown in Table 2-4, we only need four bits to specify all the functions performed by the control unit. The EXECUTION BLOCK takes these four bits, the condition code input and the zero-out signal from the counter and provides four outputs to control the counter and the mux.

Figure 2-18B shows the microword format that sup-ports the control unit. The field designated microse-quence control substitutes the encoding of Table 2-4 for some of the individual fields shown in Figure 2-17B.

2.1.9 Machine Language Versus Microprogramming

The sequencing of microinstructions is carried out by a set of OPCODES as listed in Table 2-4. For the most part, these OPCODES resemble those of an instruc-tion set for a computer, albeit more limited in scope.

In essence, the execution of microinstructions bears a strong resemblance to the execution of machine instructions. As we have seen in this chapter, the

CONDITION	CONDITIONAL JUMP SELECT	MUX SELECT	SYMBOLIC REFERENCE	ACTION
X[†]	0	00	JMPOP	SAME AS IN FIGURE 2-12 B
X	0	01	CALL MICROSUB OR JMP	
X	0	10	RTS	
X	0	11	CONT	
0	1	00	COND JMPOP	PROCEED WITH NEXT MICROINSTRUCTION
1	1	00	COND JMPOP	TRANSFER CONTROL TO ADDRESS POINTED TO BY OPCODE
0	1	01	COND CALL MICROSUB OR COND JMP	PROCEED WITH NEXT MICROINSTRUCTION
1	1	01	COND CALL MICROSUB OR COND JMP	TRANSFER CONTROL TO ADDRESS POINTED TO IN NEXT ADDRESS FIELD
0	1	10	COND RTS	PROCEED WITH NEXT INSTRUCTION
1	1	10	COND RTS	PROCEED WITH ADDRESS POINTED TO BY MICROSUBROUTINE REGISTER
0	1	11	CONT	PROCEED WITH NEXT MICROINSTRUCTION
1	1	11	CONT	PROCEED WITH NEXT MICROINSTRUCTION

[†] X = DON'T CARE

Figure 2-15. Conditional Branch Selection

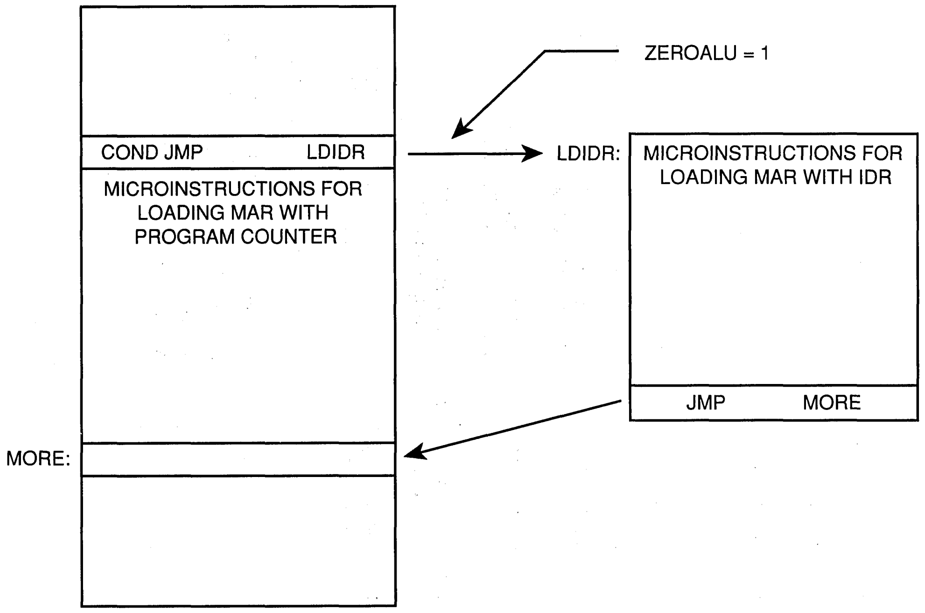

ZEROALU = 1
COND JMP
LDIDR
MICROINSTRUCTIONS FOR LOADING MAR WITH PROGRAM COUNTER
LDIDR:
MICROINSTRUCTIONS FOR LOADING MAR WITH IDR
JMP
MORE
MORE:

Figure 2-16. Conditional Branch Operation

control unit supports subroutines, loops and conditional and unconditional branches. In fact, these structures are virtually identical to those of a computer. As a result, the designs of a microprogram and a machine program are very similar and are based on the same considerations.

2.1.10 Control Unit as Processor

In a very important sense, we have created in miniature some of the elements of the computer system whose control network we are now designing. It is as if we have a computer within a computer. The difference is that while our control unit computer is rudimentary, it has the ability to implement the control section of an elaborate computer system. Furthermore, this structure allows us to rapidly make changes to the control unit simply by changing a microprogram. Contrast this to making a change to a sequential logic network supporting over 100 instructions. It is not hard to see that this is intrinsically a simpler approach.

2.1.11 Conclusion

The subject of microprogramming and the design of control unit structures for executing microprograms continues to evolve We have presented the broad outlines of microprogramming along with the design of an underlying control unit structure similar to the design of the Am29PL141 chip.

The traditional use of microprogramming has been in the design of the control unit of various computer systems, ranging from microcomputers to mainframes. However, the use of microprogramming goes far beyond that. The major point is that by using what amounts to software techniques, we are able to implement almost any general sequential network. In doing so, we gain the advantage of designing complicated networks that are easy to modify. Beyond that, the design process itself is made more efficient because it generally takes less time to write a microprogram to implement hardware structures than it takes to design hardware.

2.2 Am29PL100 MICROPROGRAMMING EXAMPLES

Section 2-1 introduced the major ideas of microprogramming as an aid in understanding the structure and use of the Am29PL100 family of programmable controllers. In this section, we build upon that knowledge and show how to design sequential logic circuits using the Am29PL100.

The goal of this section is to blur the distinction between microprogramming and sequential logic design. As pointed out in Section 2.1, microprogramming is a general logic design technique and therefore is an alternative approach to implementing complex digital logic circuits.

During the course of this section, we will design five circuits that are representative of the range of problems suitable for Am29PL100 implementation:

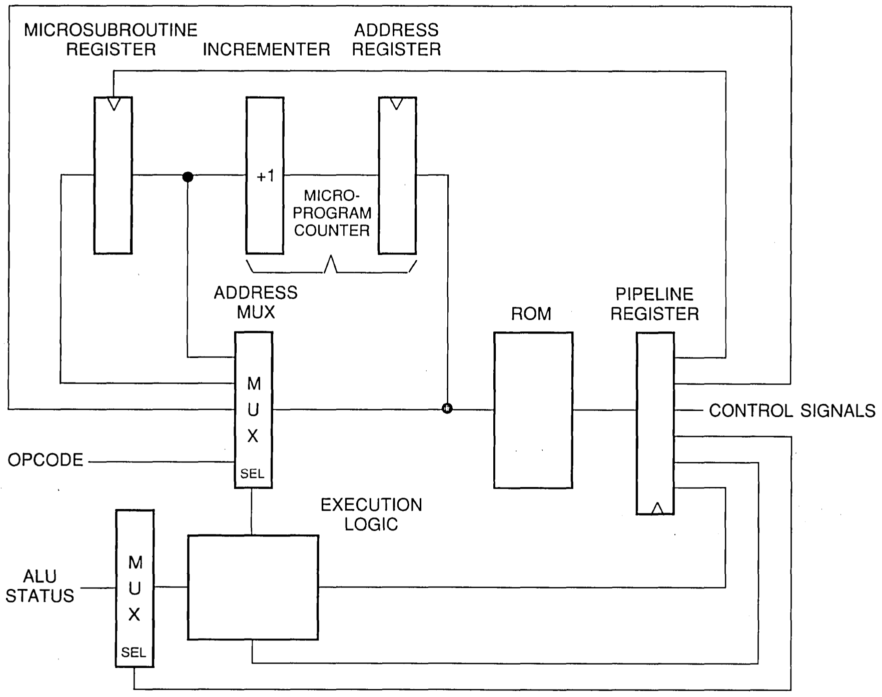


Figure 2-17a. Adding External Condition MUX to Con trol Unit

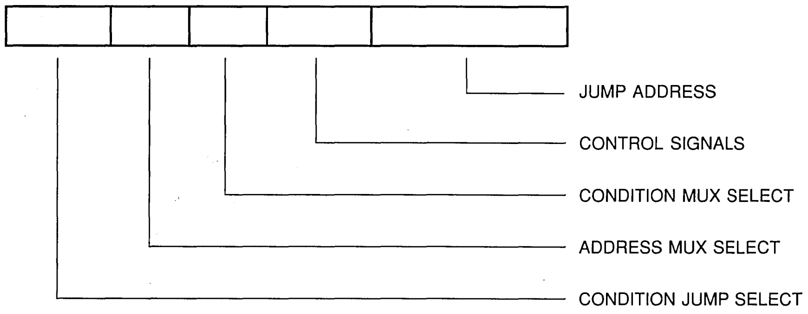


Figure 2-17b. Microword Format

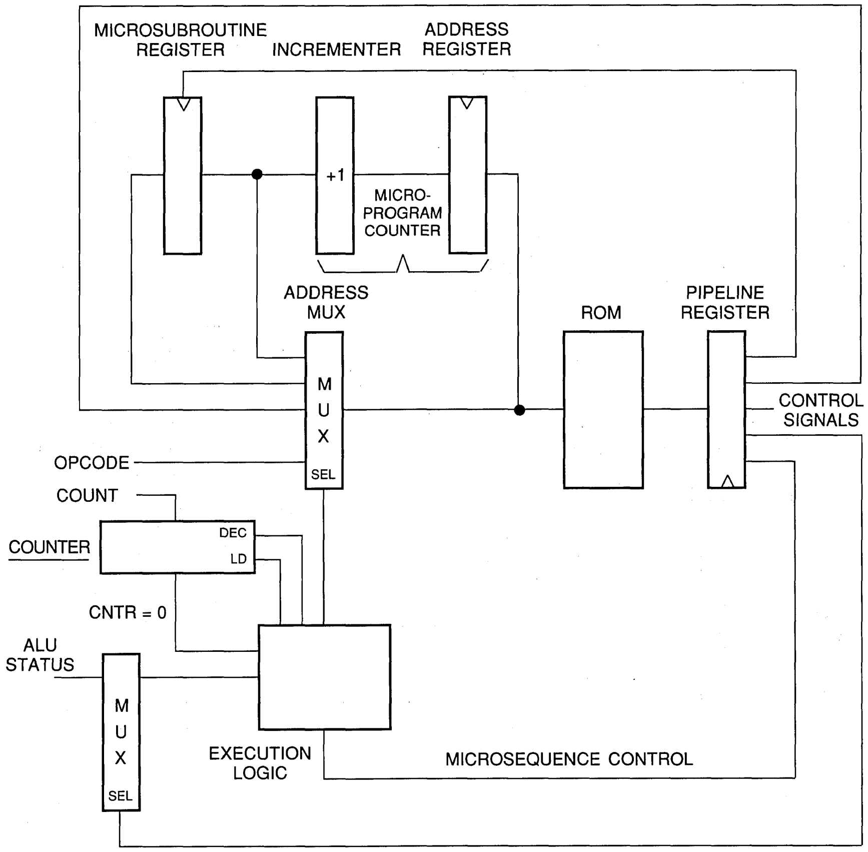


Figure 2-18a. Adding Counter to Control Unit

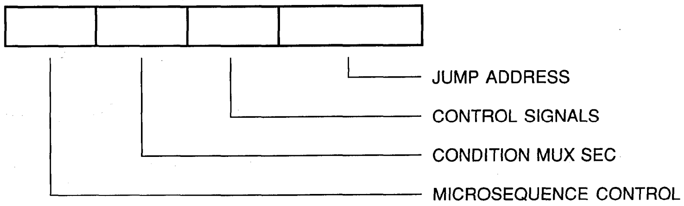


Figure 2-18b. Microword Format

CPU	BUS REQUEST LINE	
1	/BR3	HIGHEST
2	/BR2	•
3	/BR1	•
4	/BR0	LOWEST

Table 2-4. Bus Request Priority

- **Burst counter**
- **Memory controller**
- **Bus arbiter**
- **VME bus arbiter**
- **Frame store**

Each of these examples shows the interplay between the hardware function to perform and the microprogramming structures that can be used to implement that function. By the end of this section, we will have a repertoire of techniques that can be used to tackle a wide range of logic-design problems.

This chapter assumes a familiarity with the Am29PL141 data sheet to the point of knowing the general and comparison microinstruction format. On the other hand, it is not necessary, at this point, to know the function of any of the microinstructions.

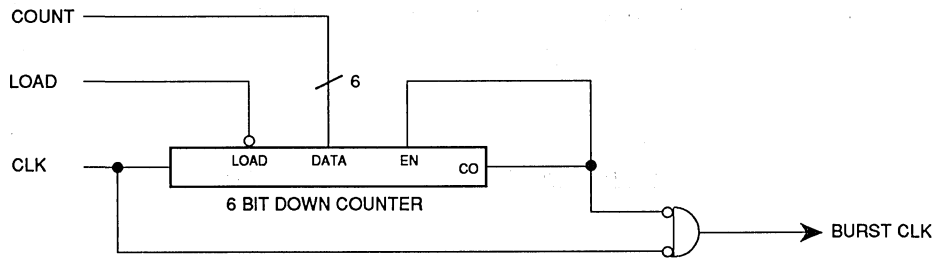
Each example introduces a number of microinstructions and supporting programming structures. As we work through the examples, consult the data sheet (order number 04179) for a detailed description of each microinstruction that is introduced.

2.2.1 Example 1: Burst Counter

A counter is an indispensable building block of almost any sequential logic circuit. It is used to count events, measure the duration of an event, measure the time interval between events, create time delays, and address memory.

Let's consider the implementation of a burst counter. Figure 2-19 shows a diagram of this circuit that operates according to the timing diagram of Figure 2-20. The following major events take place:

- Wait for LOAD time to go low.
- Load counter with value contained on counter inputs.
- Assert carry-out signal low.
- Decrement counter.



PIN FUNCTIONS

LOAD: WHEN LOW, VALUE OF COUNT LOADED INTO COUNTER ON RISING EDGE OF CLK

CO: HIGH WHEN COUNTER IS IN STATE ZERO

EN: COUNTING ENABLED WHEN LOW

Figure 2-19. 6-Bit Burst Counter

- Wait for counter to count down to zero.
- Assert carry-out signal high.
- Go to the first step

The Am29PL141 also can be used to implement a burst counter since it contains a counter and the microinstructions used to load and decrement the counter. To see how this works, take a look at the circuit shown in Figure 2-21 and the microprogram shown in Figure 2-22. This circuit allows us to program a burst of 1-64 clock pulses depending on the value contained on the TEST inputs (see Table 2-5).

To gain a fuller understanding of the microprogram, let's explore the function of the four statements in greater detail:

Statement 1 — This statement puts the Am29PL141 into an idle condition. By setting the POL field to 1, this microinstruction executes repeatedly while the LOAD line is high. When the LOAD line goes low, execution proceeds with the microinstruction at location 10 as specified by the contents of the DATA field. Output P0 remains high while this microinstruction executes, thus preventing clock-pulse generation.

The CC input is selected as the conditional input instead of one of the TEST inputs. This frees up all six TEST inputs for use in loading the counter.

Statement 2 — This statement loads the counter from the TEST inputs. The DATA field contains a mask value of 3F, allowing a burst of up to 64 clock pulses. Alternatively, a mask value of F permits only the low-order four TEST inputs to be loaded into the counter, allowing a burst of up to 16 clock pulses. As in Statement 1, output P0 remains high during the execution of this microinstruction, thus preventing clock pulse generation.

In this statement, we want to load the counter unconditionally. However, the microinstruction LDTM is conditional and is dependent on the state of one of the TEST inputs, the CC input or the EQ FF. To force this microinstruction to execute unconditionally, we take advantage of the fact that the EQ FF is zero after the Am29PL141 is reset. We therefore select the EQ FF in the TEST field and set the POL field to 1.

Statement 3 — This microinstruction decrements the counter repeatedly until a count of zero is reached. Execution then proceeds with the next sequential instruction. Output P0 remains low while this microinstruction executes, allowing clock pulses to be gated.

Normally, this microinstruction executes once and, if the counter is not zero, a branch is taken to the location specified by the DATA field. In this example, the DATA field contains the address of its own microinstruction, resulting in repeated execution.

Statement 4 — This statement causes a branch back to Statement 1 since the counter is zero. The circuit then remains in an idle state until the next LOAD pulse occurs.

The timing diagram of Figure 2-23 reflects the behavior of this microprogram. This timing diagram differs slightly from Figure 2-20 in that the carry-out line (P0) goes low one clock period after the LOAD line goes low and the number of burst clocks is greater by one. Otherwise, the Am29PL141 implementation preserves the basic functionality of the discrete logic design.

So far we have considered a burst counter operating at rates up to 20 MHz. For a burst counter operating at rates up to 10 MHz, we can use the P0 output as the burst clock and eliminate the external AND gate (see Figure 2-24a). To see how this is done, look at the microprogram contained in Figure 2-24B. In this

TEST INPUT VALUE	BURST COUNT
0	1
1	2
2	3
•	•
•	•
•	•
62	63
63	64

Table 2-5. Burst Count Versus Test Input Value

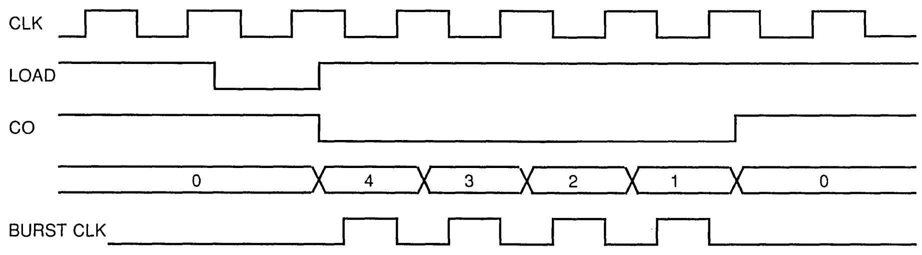


Figure 2-20. Timing Diagram of Burst Counter (Count = 4)

program, we use the LPPL microinstruction to loop through Statements 3 and 4 the number of times initially contained in the counter plus 1. In this way, we sequence through program locations 11 and 12 and in so doing create a square wave output at P0 with a frequency of one-half the clock input. Figure 2-25 indicates the locations traversed by this program upon bringing the LOAD line low with a value of 2 at the TEST inputs.

This example shows how the Am29PL141 can be used to implement a burst counter. While this implementation is limited to bursts of 64 clock pulses, burst counts of up to 2 million also can be created. Furthermore, a burst counter is but one type of counter that can be implemented with the Am29PL141. Later, we will design more complicated counter structures.

2.2.2 Example 2: Simple Memory Controller

The Am29PL141 also can be used as a dynamic memory controller (see Figures 2-26 and 2-27). In this application, the Am29PL141 performs two functions:

- Generates the proper timing signals to control the dynamic memory array
- Synchronizes the timing of the memory system with the host CPU

In our design, we have a 64-KByte memory array built out of eight 64-Kbit memory chips. Each memory chip has 8 address lines (see Figure 2-28), yet 16 address lines are needed to address a single memory location. Consequently, the 16 addresses are segmented into two groups of eight bits, referred to as the Row and Column addresses. As shown in Figure 2-29, the Row addresses are first strobed into the memory by the control line Row Address Strobe ($\overline{\text{RAS}}$). Then the Column addresses are strobed by the control signal Column Address Strobe ($\overline{\text{CAS}}$). $\overline{\text{CAS}}$ is also used in conjunction with the control signal READ to write data into or read data from the memory chip. If READ is low, data is written into the memory chip on the falling edge of $\overline{\text{CAS}}$. Alternatively, if READ is high, data is read from the memory after $\overline{\text{CAS}}$ goes low.

Figure 2-30 presents a timing diagram of the memory system. The CPU asserts the signal $\overline{\text{RD}}$ to read

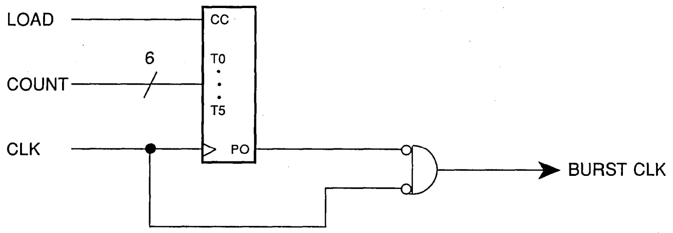


Figure 2-21. Am29PL141 Implementation of Burst Counter

STATEMENT #	LOCATION	OE	OPCODE	POL	TEST	DATA	OUTPUTS
1	0	0	WAIT	1	CC	10	1
2	10	0	LDTM	1	EQ	3F	1
3	11	0	LPPL	0	0	11	0
4	12	0	GO TO PLZ	0	0	0	1

NOTES: THE FOLLOWING SYMBOLS ARE USED AND HAVE THE INDICATED VALUES

SYMBOL	VALUE
CC	6
EQ	7

Figure 2-22. Microprogram for Burst Counter

data from the memory and the signal $\overline{WR}$ to write data into the memory. In response to either one of these signals, the memory controller generates the proper sequence of control signals that satisfy the timing requirements specified in Figure 2-29.

As shown in Figure 2-27, two address buffers, controlled by the signals $\overline{ROWEN}$ and $\overline{COLEN}$, furnish the Row and Column addresses respectively. In addition, a data buffer is used to isolate the memory system from the CPU DATA bus. When writing data into or reading data from the memory, the memory controller generates the signal $\overline{DATEN}$ to enable the data buffer. Also the signal READ controls the direction of the buffer. During a read cycle this signal is high and during a write cycle it is low.

Figure 2-31 shows how the CPU and memory system are interlocked by the signal OPCOMP. The memory controller generates this signal to allow the CPU to know when the memory system is about to write data into the memory or provide valid data for a READ operation. If the CPU cannot respond immediately to OPCOMP, it continues to assert either $\overline{RD}$ or $\overline{WR}$. In doing so, the memory cycle is lengthened. This is reflected in Figure 2-30, which shows that the signal $\overline{RD}$ is negated two clock times after OPCOMP is asserted, while the signal $\overline{WR}$ is negated only one clock period after OPCOMP is asserted. Consequently, the read cycle is one clock period longer than the write cycle.

The memory controller performs three fundamental tasks:

- Scans $\overline{RD}$ and $\overline{WR}$ lines.

  The memory controller continuously scans the $\overline{RD}$ and $\overline{WR}$ lines. If neither line is active, the memory controller asserts the control signals with the values shown in Figure 2-32A.

- Executes READ cycle.

  When the $\overline{RD}$ signal goes low, the memory controller initiates the sequence of control signals shown for the READ cycle in Figure 2-30. Figure 2-32B enumerates the values of the control signals corresponding to clock periods 1–5 of Figure 2-30.

- Executes WRITE cycle.

  When the $\overline{WR}$ signal goes low, the memory controller initiates the sequence of control signals shown for the WRITE cycle in Figure 2-30. Figure 2-32C enumerates the values of the control signals corresponding to clock periods 1'-4' of Figure 2-30.

To implement the memory controller using the Am29PL141, we need to write three microprograms, one for each of the tasks discussed above.

Let's take a look at the microprogram in Figure 2-33. Notice that the microinstruction fields contain symbols whose values are defined at the bottom of the figure. These symbols make the microprogram easier to read and are a convention used in other figures in this chapter.

As shown in Figure 2-33, Statements 1-3 carry out the steps for scanning the $\overline{RD}$ and $\overline{WR}$ lines (see Figure 2-27 for pin assignments). If neither $\overline{RD}$ nor $\overline{WR}$ goes low, the microprogram will continuously cycle through these microinstructions. While it does, the control outputs have a value of 7C hex, which gives us the proper state for the quiescent condition (see Figure 2-32a). Also notice that the GOTOPL microinstruction in Statement 3 executes unconditionally since the EQ FF is zero after the Am29PL141 is reset.

Statements 1 and 2 both use the CALPL microinstruction. For this microinstruction, if the condition selected on the TEST input is satisfied, execution pro-

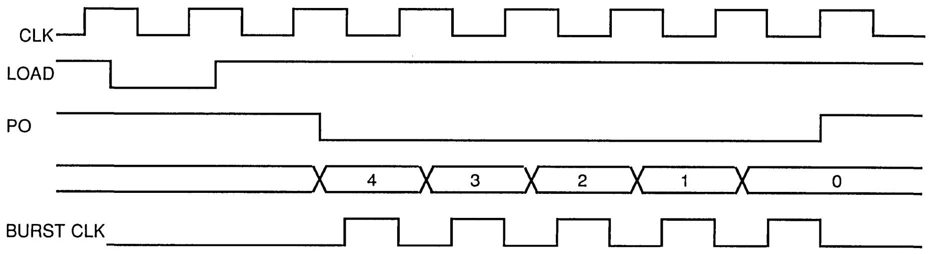


Figure 2-23. Timing Diagram for Am29PL141 Burst Counter

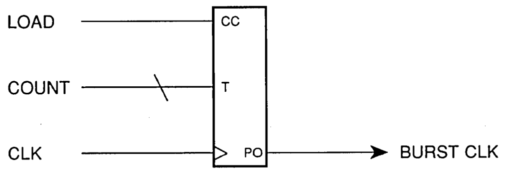


Figure 2-24a. Microprogram for Burst Counter Using PO Output

STATEMENT #	LABEL	LOC	OE	OPCODE	POL	TEST	DATA	OUTPUTS
1	START	0		WAIT	1	CC	10	1
2		10		LDTM	1	EQ	3F	1
3	LOOP	11		CONT	0	0	0	0
4		12		LPPL	0	0	LOOP	1
5		13		GOTOPLZ	0	0	START	1

SYMBOL	VALUE
CC	6
EQ	7
START	0
LOOP	11

Figure 2-24b. Burst Counter Using PO Output As Burst Clock

LOAD	LOCATIONS	PO
1	0	1
•	•	•
•	•	•
0	0	1
1	10	1
•	11	0
•	12	1
•	11	0
•	12	1
•	11	0
•	12	1
•	13	1
•	0	1
1	0	1
	•	
	•	

Figure 2-25. Program Locations Traversed After Load Line Goes LOW

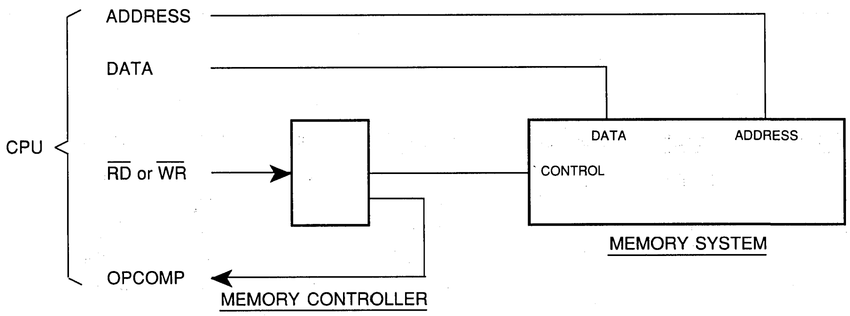


Figure 2-26. Block Diagram of Memory System

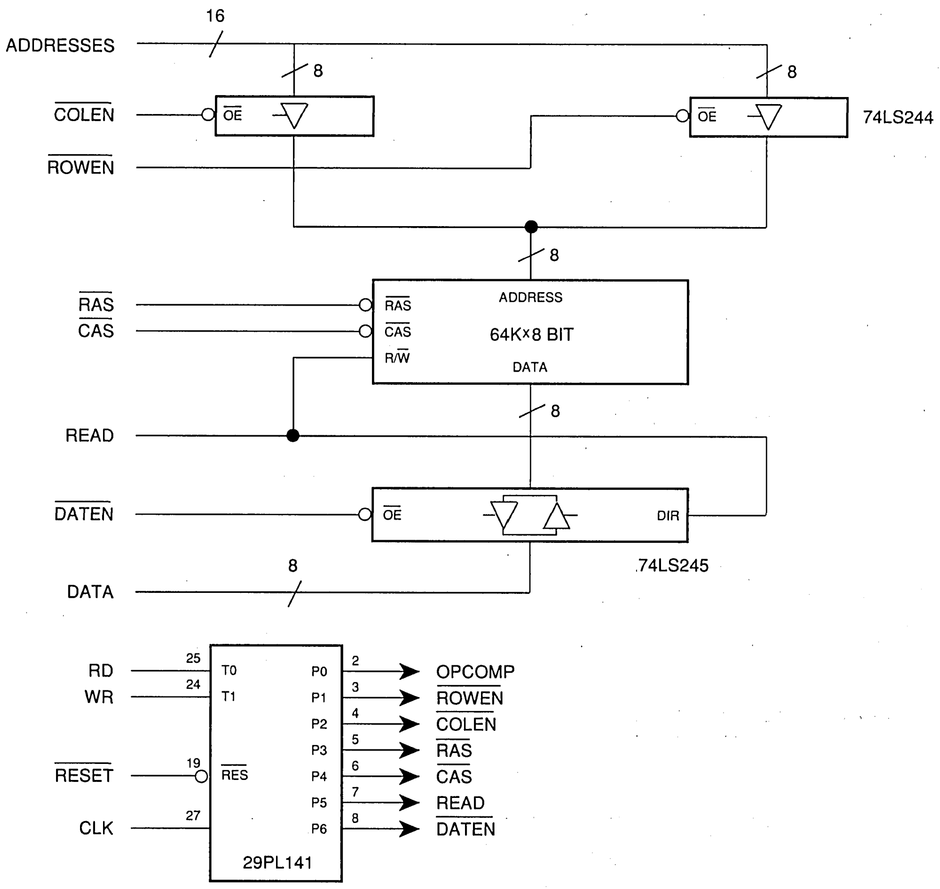


Figure 2-27. Simple Memory Controller

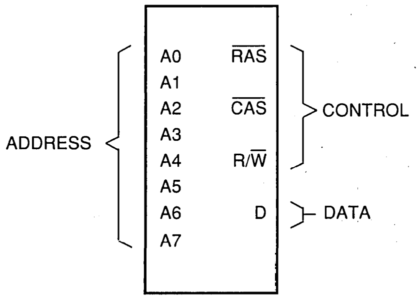
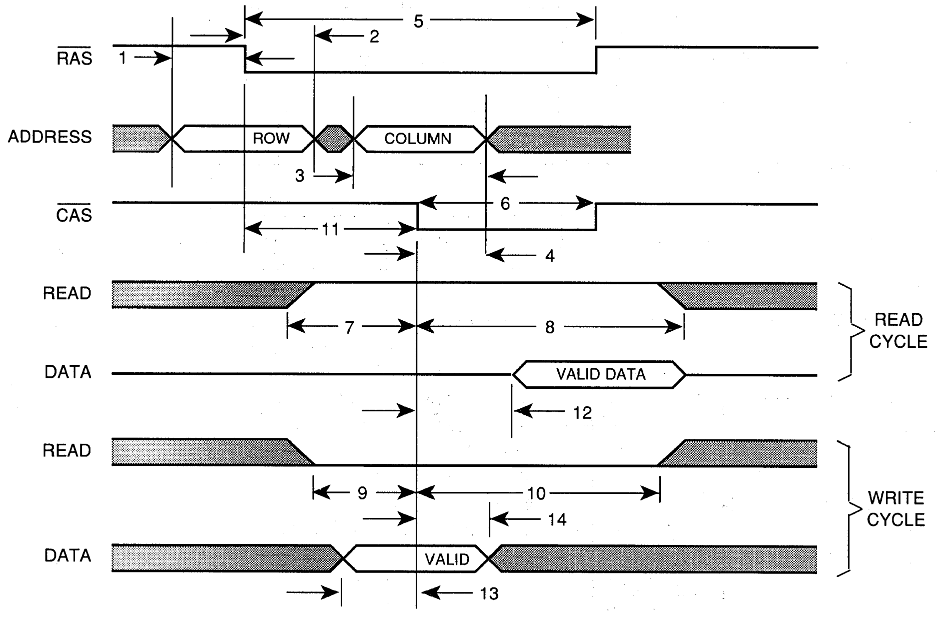


Figure 2-28. 64K Bit Dynamic Memory Chip

ceeds with the microinstruction whose address is specified in the DATA field. In Statement 1 for example, when $\overline{RD}$ is zero, execution proceeds with the microinstruction at location 10, since the value of the symbol READ in the DATA field is 10 hex. Also, the address of the following microinstruction is saved in the subroutine register (SREG). When the called microsubroutine executes a RET microinstruction, the content of the SREG is used to point to the next microinstruction. In our example, the microinstruction at location 2 is executed after a RET microinstruction.

Figure 2-34 presents a flow diagram for the microprogram of Figure 2-33. The purpose of this diagram is to present the major activities performed by the microprogram without explicitly identifying the specific microinstructions. Flow diagrams are valuable visual aids as they help focus attention on the interplay between the signal inputs and outputs.



		MIN	MAX
1.	ROW ADDRESS SETUP TIME	20	
2.	ROW ADDRESS HOLD TIME	0	
3.	COLUMN ADDRESS SETUP TIME	20	
4.	COLUMN ADDRESS HOLD TIME	0	
5.	PULSE DURATION $\overline{RAS}$ LOW	180	10000
6.	PULSE DURATION $\overline{CAS}$ LOW	80	10000
7.	READ COMMAND SETUP TIME	20	
8.	READ COMMAND HOLD TIME	20	
9.	WRITE COMMAND SETUP TIME	20	
10.	WRITE COMMAND HOLD TIME	20	
11.	$\overline{RAS}$ TO $\overline{CAS}$ DELAY	80	
12.	ACCESS TIME FROM $\overline{CAS}$	80	
13.	DATA IN SETUP TIME	20	
14.	DATA IN HOLD TIME	20	

Figure 2-29. Memory Timing Requirements

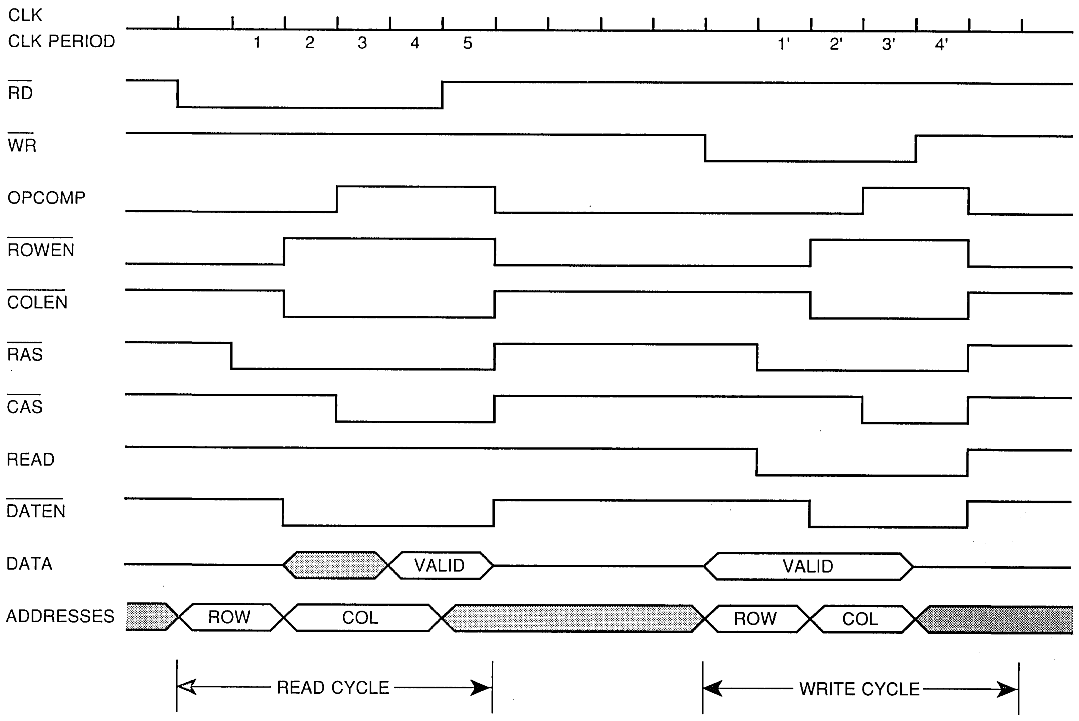


Figure 2-30. Timing Diagram of Simple Memory Controller

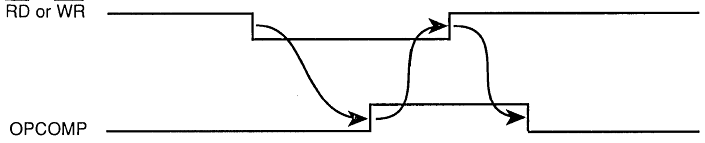


Figure 2-31. Interlocking of RD, WR And OPCOMP

(P6) $\overline{\text{DATEN}}$	(P5) READ	(P4) $\overline{\text{CAS}}$	(P3) $\overline{\text{RAS}}$	(P2) $\overline{\text{COLEN}}$	(P1) $\overline{\text{ROWEN}}$	(P0) $\overline{\text{OPCOMP}}$	EQUIVALENT HEX NO
1	1	1	1	1	0	0	7C

Figure 2-32a. Memory Control Signal Values During Quiescent State

CLOCK PERIOD	(P6) $\overline{\text{DATEN}}$	(P5) READ	(P4) $\overline{\text{CAS}}$	(P3) $\overline{\text{RAS}}$	(P2) $\overline{\text{COLEN}}$	(P1) $\overline{\text{ROWEN}}$	(P0) OPCOMP	EQUIVALENT HEX NO
1	1	1	1	0	1	0	0	74
2	0	1	1	0	0	1	0	32
3	0	1	0	0	0	1	1	23
4	0	1	0	0	0	1	1	23
5	0	1	0	0	0	1	1	23

Figure 2-32b. Memory Control Signal Values During Read Cycle

CLOCK PERIOD	(P6) $\overline{\text{DATEN}}$	(P5) READ	(P4) $\overline{\text{CAS}}$	(P3) $\overline{\text{RAS}}$	(P2) $\overline{\text{COLEN}}$	(P1) $\overline{\text{ROWEN}}$	(P0) OPCOMP	EQUIVALENT HEX NO
1'	1	0	1	0	1	0	0	54
2'	0	0	1	0	0	1	0	12
3'	0	0	0	0	0	1	1	03
4'	0	0	0	0	0	1	1	03

Figure 2-32c. Memory Control Signal Values During Write Cycle

STATEMENT NO	LABEL	LOC	OE	OPCODE	POL	TEST	DATA	OUTPUTS	COMMENTS
				MAIN LOOP FOR SCANNING $\overline{RD}$ & $\overline{WR}$ INPUTS					
1	SCAN	0	0	CALPL	HIGH	RD	READ	7C	CALL READ IF $\overline{RD}$ = 0
2		1	0	CALPL	HIGH	WR	WRITE	7C	CALL WRITE IF $\overline{WR}$ = 0
3		2	0	GOTOPL	HIGH	EQ	SCAN	7C	BRANCH TO SCAN
				MICROSUBROUTINE TO EXECUTE READ CYCLE					
4	READ	10	LOW	CONT	ZERO	ZERO	ZERO	74	ASSERT $\overline{RAS}$
5		11	LOW	CONT	ZERO	ZERO	ZERO	32	SWITCH ADDRESS BUFFERS
6		12	LOW	CONT	ZERO	ZERO	ZERO	23	ASSERT $\overline{CAS}$ & OPCOMP
7		13	LOW	WAIT	LOW	RD	MORE	23	WAIT UNTIL $\overline{RD}$ = 1
8	MORE	14	LOW	RET	HIGH	EQ	SCAN	7C	RETURN TO SCAN
				MICROSUBROUTINE TO EXECUTE WRITE CYCLE					
9	WRITE	20	LOW	CONT	ZERO	ZERO	ZERO	RASWR	ASSERT $\overline{RAS}$
10		21	LOW	CONT	ZERO	ZERO	ZERO	SWITCH	SWITCH ADDRESS BUFFERS
11		22	LOW	CONT	ZERO	ZERO	ZERO	CAS-WR-OP	ASSERT $\overline{CAS}$ & OPCOMP
12	LOOP	23	LOW	GOTOPL	LOW	WR	LOOP	CAS-WR-OP	WAIT UNTIL $\overline{WR}$ = 1
13		24	LOW	RET	HIGH	EQ	SCAN	IDLE	RETURN TO SCAN

SYMBOL	**VALUE**	**SYMBOL**	**VALUE**	**SYMBOL**	**VALUE**
SCAN	0	HIGH	1	RASWR	54
READ	10	LOW	0	SWITCH	12
LOOP	13	ZERO	0	CAS-WR-OP	03
WRITE	20	RD	0	IDLE	7C
MORE	24	WR	1		

Figure 2-33. Microprogram for Simple Memory Controller

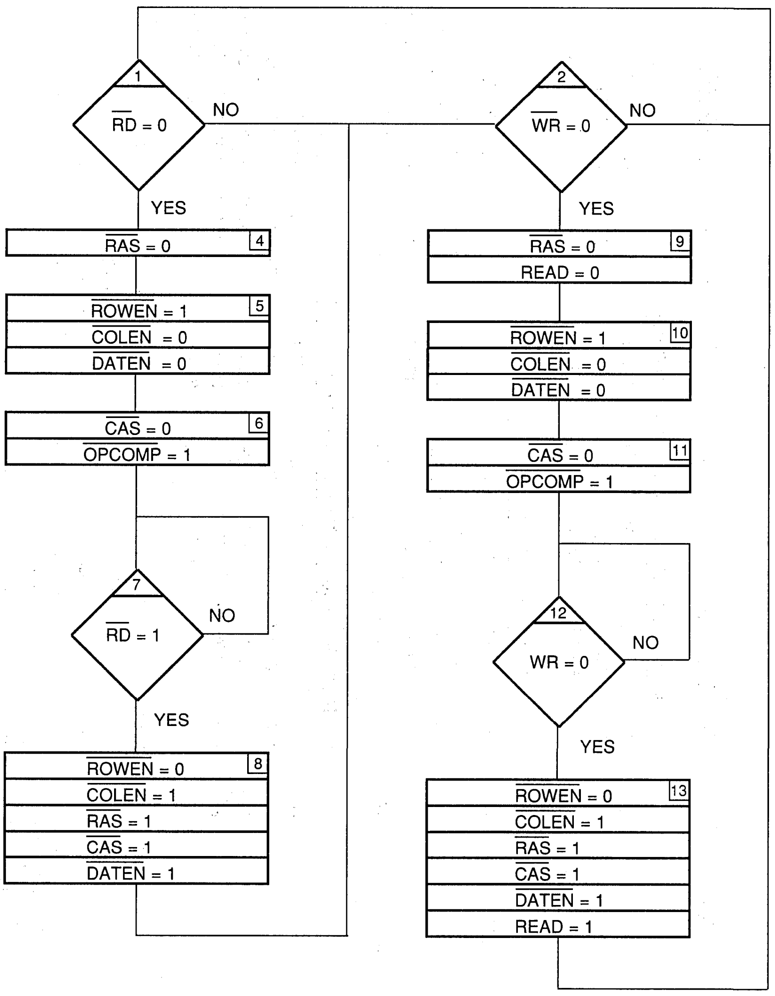


Figure 2-34. Flow Diagram for Simple Memory Controller Microprogram

The two diamonds at the top of Figure 2-34 pictorially present the activity of Statements 1 and 2 of the scan loop. Statement 3 is not explicitly identified although it is implied by the line that joins the two diamonds at the top of the figure.

The left-hand side of Figure 2-34 lists the steps performed by the READ microsubroutine. Each rectangle contains a statement that identifies the desired state of the control outputs. The number at the upper right of each rectangle refers to a statement number listed in Figure 2-33. Notice that executing Statements 4-6 results in the control outputs tracing out the activity specified by clock periods 1-3 in Figure 2-30 (also see Figure 2-32b).

Statement 7 performs the function denoted by the diamond in the middle of the flow diagram for the READ microsubroutine. The purpose of this statement is to implement the interlocking between the CPU and the memory. As discussed above, the READ cycle is lengthened if the CPU does not negate its $\overline{\text{RD}}$ signal after receiving the control signal OPCOMP from the memory controller. In effect, the memory controller freezes the state of the control lines until the CPU negates the $\overline{\text{RD}}$ signal.

The WAIT microinstruction holds the control outputs in the state defined by the OUTPUT field (23 hex) until $\overline{\text{RD}}$ goes high. Execution then proceeds with Statement 8, which returns the control outputs to the quiescent state. Statement 8 contains a conditional RET microinstruction that is dependent on the state of the EQ FF. Consequently, execution returns unconditionally to Statement 2 in the scan loop as discussed above.

When the $\overline{\text{WR}}$ line goes low, the WRITE microsubroutine is executed. Statements 9-11 (see Figure 2-33) are virtually identical to Statements 4-6 of the READ microsubroutine. The only difference is in the values of the OUTPUT field, which reflects the fact that the READ control signal is low in the WRITE microsubroutine and high in the READ microsubroutine. (See Figures 2-32b and 2-32c.) As a result, the timing shown in Figure 2-30 for clock periods 1'-3' is properly synthesized.

The microinstruction GOTOPL of Statement 12 executes repeatedly until $\overline{\text{WR}}$ goes high since the DATA field contains the address of its own microinstruction. In effect, it behaves identically to the WAIT microinstruction and is an alternative way to implement a wait state. Statement 13 is identical to Statement 8, and when it executes, control returns to Statement 3 in the scan loop.

This example shows the power of the Am29PL141. Using just 13 microinstructions, the complicated timing structure of the memory controller is completely implemented. In contrast to a discrete logic implementation, changing the timing is easily accomplished by changing just a few lines of microcode.

2.2.3 Example 3: Bus Arbiter

A common problem in digital design is arbitrating among different requests and then taking appropriate action. Examples of arbitrating include:

Interrupt controller — An interrupt controller fields a number of interrupt requests from various devices and then provides a code indicating which interrupt will be serviced.

Dual-port memory arbiter — A memory arbiter fields requests for access to a shared memory from two different CPUs and then grants memory access to one of them.

Bus arbiter — A bus arbiter fields requests from several CPUs for use of a shared bus and then grants bus access to one of them.

A general view of this process is presented in Figure 2-35, which shows the three major phases of the arbitration process:

- Encode requests into a binary word.
- Select one of the requests.
- Implement a series of actions to carry out the requests.

Requests may arise one at a time or simultaneously. In the memory controller example, the CPU generated either a $\overline{\text{RD}}$ signal or a $\overline{\text{WR}}$ signal, but not both at the same time. However, in many systems, simultaneous requests are made. For these, as part of the selection process, the arbiter must incorporate a facility for prioritizing the process.

To explore these ideas further, let's design a bus arbiter for a 4-processor system sharing a common memory (see Figures 2-36 and 2-37). Each processor has three arbitration control lines that are used to gain access to the bus (see Figures 2-38 and 2-39):

Bus Request ($\overline{\text{BR}}$) — A CPU requests bus use by asserting its individual $\overline{\text{BR}}$ signal to the bus arbiter.

Bus Grant ($\overline{\text{BG}}$) — When $\overline{\text{BBYS}}$ is not asserted, the bus arbiter arbitrates among any pending requests and asserts the appropriate $\overline{\text{BG}}$ signal to the successful requester.

Bus Busy ($\overline{\text{BBYS}}$) — After receiving $\overline{\text{BG}}$ from the

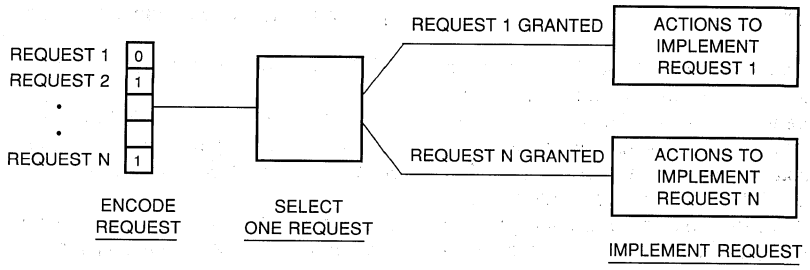


Figure 2-35. Three Phases of Arbitration Process

arbiter, the selected requester asserts $\overline{\text{BBYS}}$. When the CPU finishes using the bus, it releases $\overline{\text{BBYS}}$. The bus arbiter then arbitrates any pending requests. Notice that $\overline{\text{BBYS}}$ is a common, open-collector line driven by all the CPUs.

Our system has four processors. At any given instant, therefore, up to four bus requests can be active. To provide orderly access to the bus, the bus arbiter prioritizes these requests in accordance with Table 2-6.

Figure 2-40 illustrates how the priority scheme works by showing two bus transactions. For the first transaction, CPU1 and CPU2 both request the bus at the same time. From Table 2-6 we see that CPU1, which uses the $\overline{\text{BR3}}$ line, has a higher priority than CPU2, which uses the $\overline{\text{BR2}}$ line. Consequently, the bus arbiter grants the bus to CPU1. After completing its transactions, CPU1 releases the bus by negating the $\overline{\text{BBYS}}$ line. Sensing the negation of $\overline{\text{BBYS}}$, the arbiter arbitrates pending requests. Since $\overline{\text{BR2}}$ is still pending, and there are no other requests, the bus arbiter asserts $\overline{\text{BG2}}$. CPU2 now proceeds with its bus transactions.

Figure 2-41 shows a simplified state diagram of the bus arbiter that defines its behavior given any combination of bus requests. Five major states are shown:

State 0 — This state represents the idle condition. The bus arbiter remains in this state while the bus is not busy and there are no pending bus requests.

State 1 — This state is reached when the bus is not busy and CPU1 has asserted its bus request line. Notice that other CPUs also may have made requests. These requests are ignored, however, in keeping with the priority defined by Table 2-6.

State 2 — This state is reached when the bus is not busy, CPU1 is not requesting the bus, and CPU2 has asserted its bus request line. Bus requests from CPU3 and CPU4, if active, are ignored since they are of lower priority.

State 3 — This state is reached when the bus is not busy, neither CPU1 nor CPU2 have pending bus requests, and CPU3 has asserted its bus request line. A request from CPU4, if active, is ignored.

State 4 — This state is reached when the bus is not busy and only CPU4 has asserted its bus request line.

Figures 2-42a and 2-42b show state transition tables defining the transitions between State 0 and each of the other states described above. Notice, in particular, that the state of the bus request lines is encoded into a hex word, giving us 16 possible values. A correspondence then is made between each value and the next state reached by the bus arbiter.

Let's now turn our attention to implementing the bus arbiter using the Am29PL141 (see Figure 2-37).

The state diagram of Figure 2-41 suggests that we need to write five microprograms, one for each state. The basic functions of these microprograms are outlined below (see Figure 2-43):

Microprogram for State 0 — This microprogram examines the state of the TEST inputs and then, using the state transition table in Figure 2-42b, decides which microprogram to jump to next.

Microprogram for States 1–4 — Each of these microprograms implements the timing indicated in Figure 2-39. They are functionally identical to each

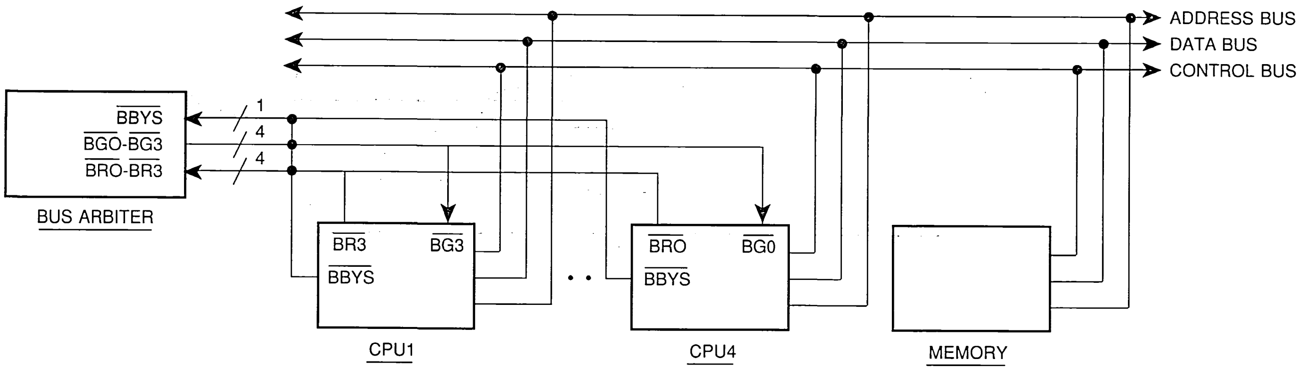


Figure 2-36. Four-Processor Computer System With Shared Memory

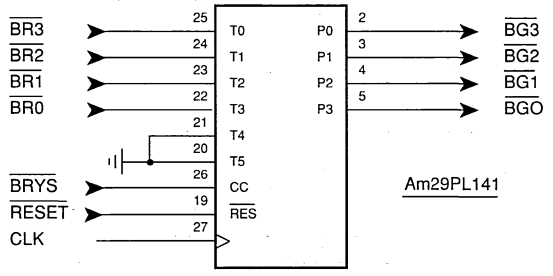


Figure 2-37. Am29PL141-Based Arbiter

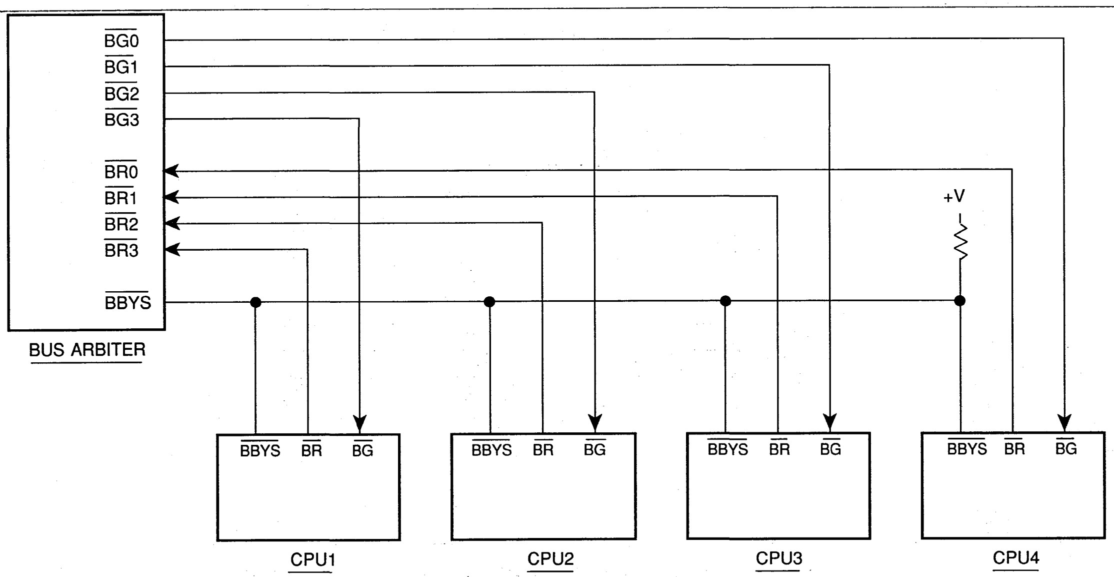

$\overline{\text{BG0}}$
$\overline{\text{BG1}}$
$\overline{\text{BG2}}$
$\overline{\text{BG3}}$
$\overline{\text{BR0}}$
$\overline{\text{BR1}}$
$\overline{\text{BR2}}$
$\overline{\text{BR3}}$
$\overline{\text{BBYS}}$
BUS ARBITER
+V
$\overline{\text{BBYS}}$ $\overline{\text{BR}}$ $\overline{\text{BG}}$
CPU1
$\overline{\text{BBYS}}$ $\overline{\text{BR}}$ $\overline{\text{BG}}$
CPU2
$\overline{\text{BBYS}}$ $\overline{\text{BR}}$ $\overline{\text{BG}}$
CPU3
$\overline{\text{BBYS}}$ $\overline{\text{BR}}$ $\overline{\text{BG}}$
CPU4

Figure 2-38. Detailed View of Bus Arbiter Subsystem

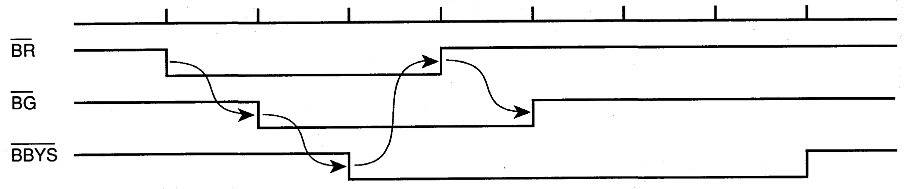


Figure 2-39. Basic Timing Diagram of Bus Arbiter

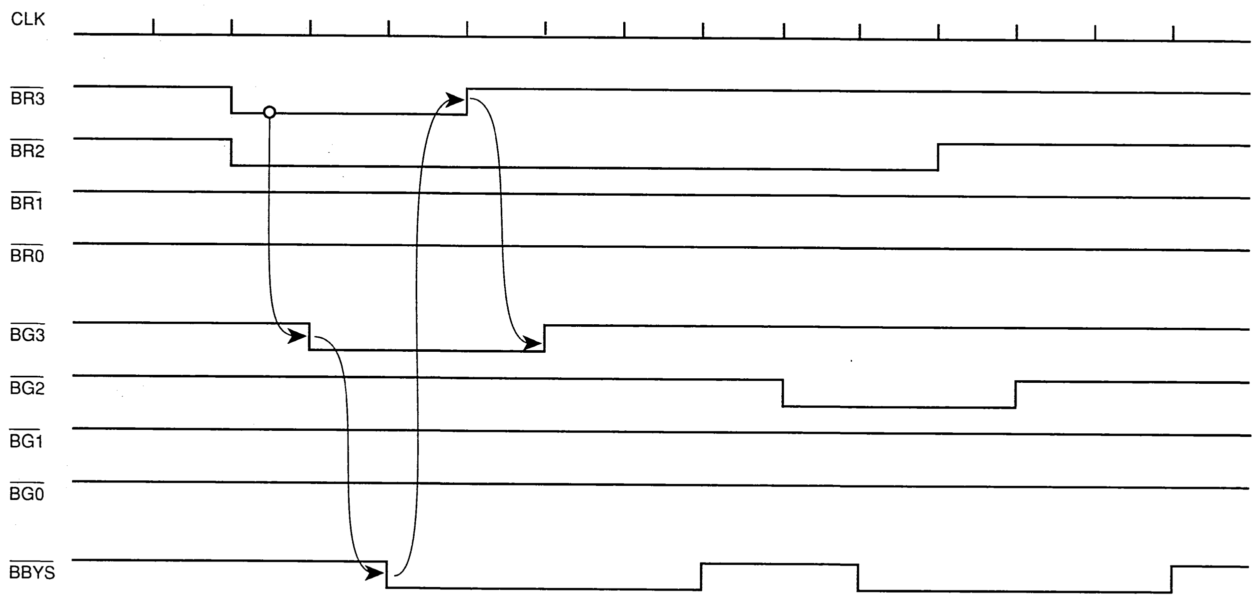
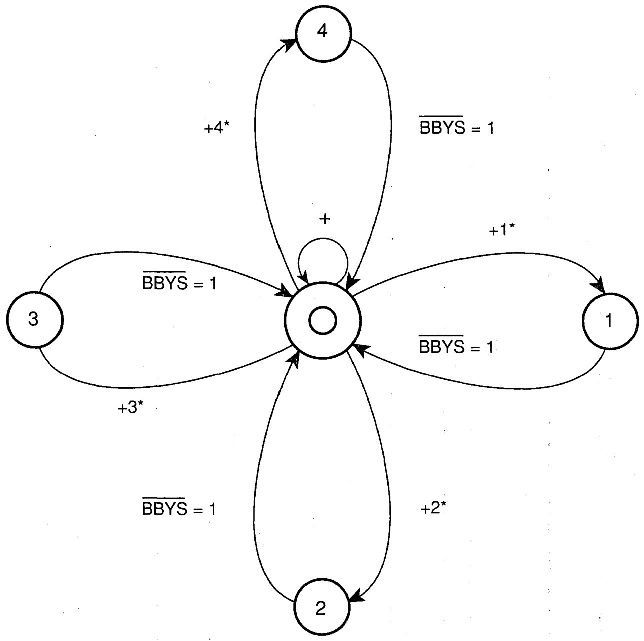


Figure 2-40. Timing Diagram for Two Active Requests



$+1^*: (\overline{BR3} = 0) * (\overline{BBYS} = 1)$

$+2^*: (\overline{BR3} = 1) * (\overline{BR2} = 0) * (\overline{BBYS} = 1)$

$+3^*: (\overline{BR3} = 1) * (\overline{BR2} = 1) * (\overline{BR1} = 0) * (\overline{BBYS} = 1)$

$+4^*: (\overline{BR3} = 1) * (\overline{BR2} = 1) * (\overline{BR1} = 1) * (\overline{BR0} = 0) * (\overline{BBYS} = 1)$

$+: (\overline{BR3} = 1) * (\overline{BR2} = 1) * (\overline{BR1} = 1) * (\overline{BR0} = 1) * (\overline{BBYS} = 1)$

Figure 2-41. Simplified State Diagram of Bus Arbiter Showing State Selection

other and differ only in the set of bus request/grant lines that are activated.

Let's take a closer look at State 0. In this state, it is necessary to arbitrate among all bus requests within one clock cycle. Consequently, we cannot write a microprogram that simply scans each of the bus request lines, one after the other, as we did in the memory controller microprogram. Fortunately, the Am29PL141 has the GOTOTM microinstruction, which allows us to examine the TEST lines in parallel.

The GOTOTM microinstruction is a conditional branch instruction that uses the TEST inputs to point to the next microinstruction to be executed.Figure 2-44 indicates how the GOTOTM microinstruction works. As shown, location F contains the GOTOTM microinstruction with a mask value of F. With this mask value, only the lower four TEST inputs are used.

Let's see what happens when this microinstruction executes. If the test inputs are all high (see Table 2-7), control passes back to location F. Therefore, this microinstruction executes repeatedly until the TEST inputs assume a new value. When that happens, control is passed to the microinstruction whose address is specified in Table 2-7. During the next clock cycle, the selected GOTOPL microinstruction in Figure 2-44 executes and a branch is taken to the address specified in the DATA field.

The structure outlined in Figure 2-44 allows us to implement the State 0 microprogram. As noted above, Figure 2-42b contains the state transition

$\overline{\text{BR0}}$	$\overline{\text{BR1}}$	$\overline{\text{BR2}}$	$\overline{\text{BR3}}$	NEXT STATE
X	X	X	0	1
X	X	0	1	2
X	0	1	1	3
0	1	1	1	4
1	1	1	1	0

Figure 2-42a. Compact State Transitions

$\overline{\text{BR0}}$	$\overline{\text{BR1}}$	$\overline{\text{BR2}}$	$\overline{\text{BR3}}$	HEX EQUIV	NEXT STATE
0	0	0	0	0	1
0	0	1	0	2	1
0	1	0	0	4	1
0	1	1	0	6	1
1	0	0	0	8	1
1	0	1	0	A	1
1	1	0	0	C	1
1	1	1	0	E	1
0	0	0	1	1	2
0	1	0	1	5	2
1	0	0	1	9	2
1	1	0	1	D	2
0	0	1	1	3	3
1	0	1	1	B	3
0	1	1	1	7	4
1	1	1	1	F	0

Figure 2-42b. Expanded State Transitions

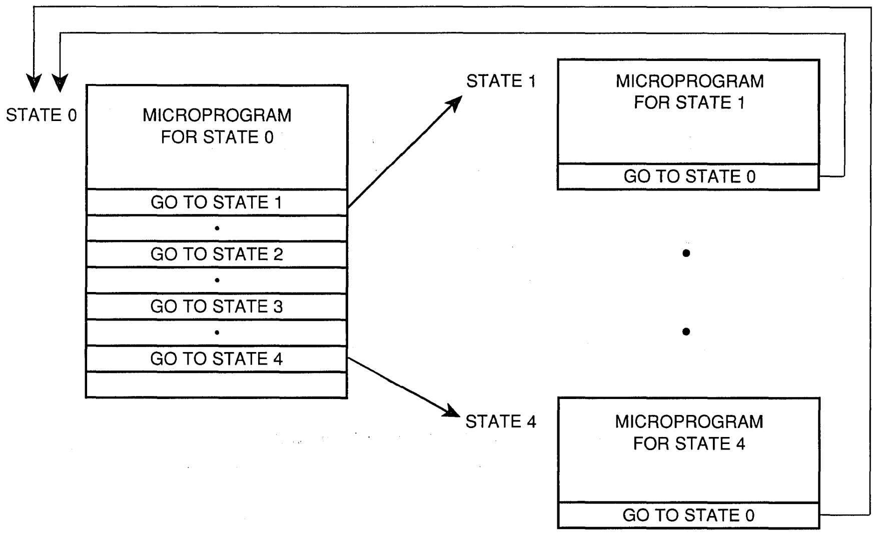


Figure 2-43. Overall Structure and Interaction Between Microprograms for Bus Arbiter

table that defines the conditions under which transitions take place from State 0 to States 1–4. Now take a look at Figure 2-45a and notice the locations containing a GOTO ST1 microinstruction. (GOTO refers to the GOTOPL microinstruction, and ST1 is the address contained in the DATA field.) These locations are precisely those corresponding to the HEX EQUIVALENT column of Figure 2-42b. In a similar vein, Figure 2-45b shows that locations 1, 5, 9 and D contain GOTO ST2 microinstructions corresponding again to the HEX EQUIVALENT column in Figure 2-42b.

Locations O-E in Figure 2-44 are called a jump table as they contain a table of addresses of microprograms. In the GOTOTM microinstruction, the TEST inputs function as an index into this table. In our example, the jump table implements the priority scheme indicated by Table 2-6. Notice, however, that we can establish other priorities by simply changing the addresses in the jump table.

Let's consider now the microprograms for States 1–4. As stated above, the basic function of each of these microprograms is to implement the timing diagram shown in Figure 2-39. This task is similar to the READ and WRITE microsubroutines of the memory controller.

Figure 2-39 has been redrawn in Figure 2-46 to show the activity that takes place in State 1. Notice, in particular, that $\overline{\text{BBYS}}$ is asserted two clock periods after $\overline{\text{BG}}$ rather than one clock period later as in Figure 2-39. The reason for this is that the $\overline{\text{BR}}$, $\overline{\text{BG}}$ and $\overline{\text{BBYS}}$ are interlocked and not explicitly dependent upon any specified time delay between signal transitions.

Figure 2-47 presents a flow diagram that helps bring out the relationships between the bus arbitration signals. As shown, the question within a diamond refers to the state of the selected signal on the TEST inputs. The rectangle to the left of each diamond contains the state of the OUTPUT signal that is relevant for State 1.

Notice that by tracing through the statements in Figure 2-47, we wind up synthesizing the timing of Figure 2-46. Also, State 1 can be seen as three substates since each diamond represents, in fact, a new state of the bus arbiter. These new states are labelled in parentheses at the right of each diamond in Figure 2-47. Figure 2-41 has been redrawn in Figure 2-48 to include these new states and represents the complete state diagram of the bus arbiter.

The microprogram for State 1 is presented in Figure 2-49. As shown, it is written in the Am29PL141

CPU	BUS REQUEST LINE	PRIORITY	
1	$\overline{BR3}$	1	HIGHEST
2	$\overline{BR2}$	2	
3	$\overline{BR1}$	3	
4	$\overline{BR0}$	4	LOWEST

Table 2-6. Bus-Request Priority

T3	T2	T1	T0	ADDRESS POINTED TO
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	A
1	0	1	1	B
1	1	0	0	C
1	1	0	1	D
1	1	1	0	E
1	1	1	1	F

Table 2-7. Value of Test Inputs Versus Branch Address

assembler language. This language emphasizes the logical functioning of the microinstructions and relieves the programmer of having to fill in each field of the microinstruction. As a result, microprograms are easier to write and understand. Notice in Figure 2-49 that the three program statements correspond to the three diamonds in Figure 2-47.

The entire microprogram for the bus arbiter is shown in Figure 2-50. The microprogram for State 0 consists of a 15-entry jump table and one microinstruction and the microprograms for States 1–4 each consist of three microinstructions.

The only statement we have not yet discussed is the microinstruction at location 3F, labelled "PON." Recall from the Am29PL141 data sheet that when RESET is activated, address 3F is presented to the Am29PL141 PROM. As a result, the microinstruction at location 3F is the first one executed after RESET is released. For our example, the microprogram waits at location 3F until the $\overline{BBYS}$ goes high. When this happens, the entire system has reached a stable operating condition and the arbiter enters State 0.

The bus arbiter presented in this section is an example of a fairly complex digital design that you can, however, implement very simply using the Am29PL141. In fact, comparing the state diagram in Figure 2-48 to the microprogram in Figure 2-50, we see that each state is implemented using just one microinstruction.

Many digital systems can be represented by state diagrams such as that in Figure 2-48. The techniques developed in this example are quite general and are applicable to the design of a broad range of digital systems.

2.2.4 Example 4: VME Bus Arbiter

In the last example, we designed a bus arbiter that used a fixed priority (see Table 2-8). In general, this scheme works well. However, when all of the CPUs

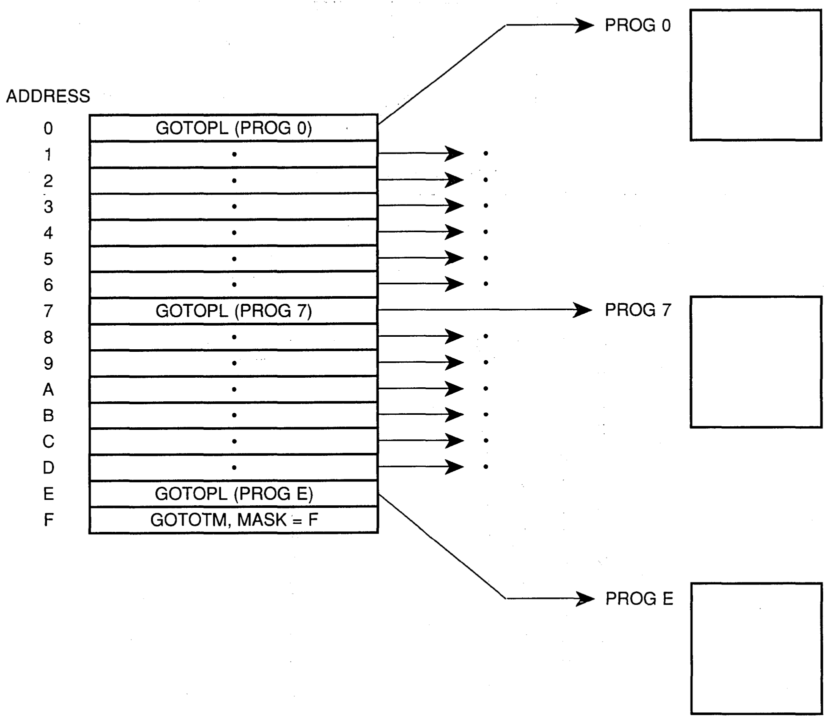


Figure 2-44. Operation of GOTOTM Microinstruction

are making frequent bus requests, the CPUs at the two highest priority levels gain access to the bus, but those at the lower levels do not.

Figure 2-51 shows a timing diagram of the activity that takes place when three CPUs repeatedly make bus requests. As shown, the arbiter alternately honors requests from CPU1 and CPU2. In contrast, CPU3 is totally locked out and is never granted bus access. To circumvent this problem, a bus arbiter also can employ a rotating priority scheme. In this scheme, each time the bus arbiter goes through an arbitration cycle, it does so using a different set of priorities.

Let's illustrate this with an example. Consider a 3-processor system sharing a bus. As shown in Figure 2-52, three priority tables are used, designated PRI1, PRI2 and PRI3. Now look at Figure 2-53. In this timing diagram, all three CPUs make repeated bus requests as in Figure 2-51. This time, however, all three CPUs have equal access to the bus.

The standard VME Bus Specification Manual defines two bus arbiter options: priority (PRI) and round robin select (RRS). The PRI VME bus arbiter uses a fixed priority scheme and functions in a manner similar to the bus arbiter of Example 3. The RRS VME bus arbiter employs a rotating priority scheme as described above, except with four processors instead of three.

Our next design is the RRS VME bus arbiter. It is similar to the bus arbiter of Example 3 in that it employs the same protocol for the interaction between the Bus Request ($\overline{BR}$), Bus Grant ($\overline{BG}$) and Bus Busy ($\overline{BBYS}$) lines. The difference lies in its rotating, rather than fixed, priority scheme. Before proceeding, however, let's review some of the highlights of the design in Example 3.

Figures 2-54 and 2-55 present an overview of the structure of the microprograms for implementing the bus arbiter in Example 3. As shown, the GOTOTM microinstruction, in conjunction with the jump table, implements the fixed priority scheme used by the bus arbiter (see Table 2-6 and Figure 2-42b). Recall,

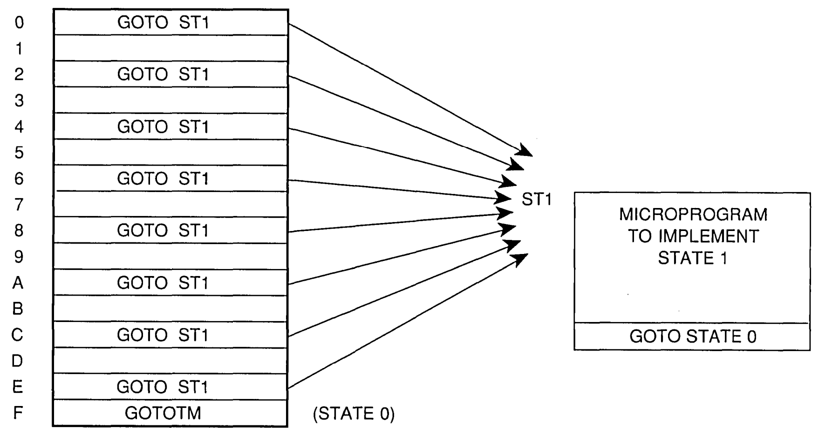


Figure 2-45a. Locations Containing Branch Addresses to State 1 Microprogram

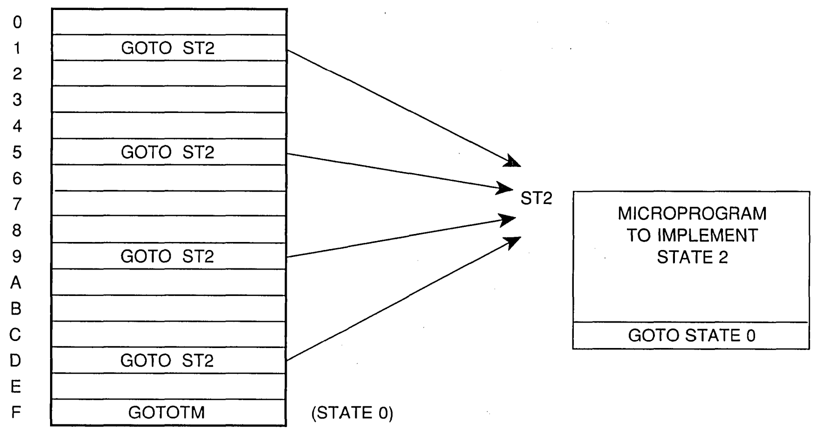


Figure 2-45b. Locations Containing Branch Addresses to State 2 Microprogram

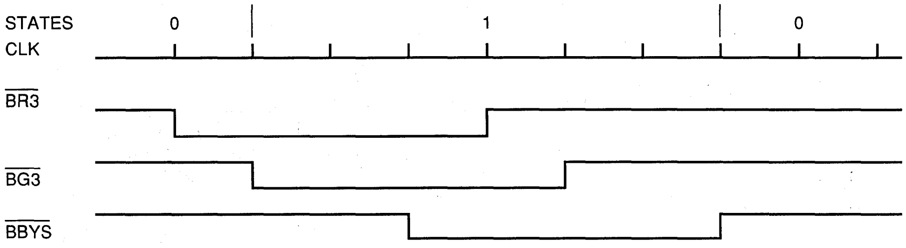


Figure 2-46. Timing for BR3-BG3 Cycle

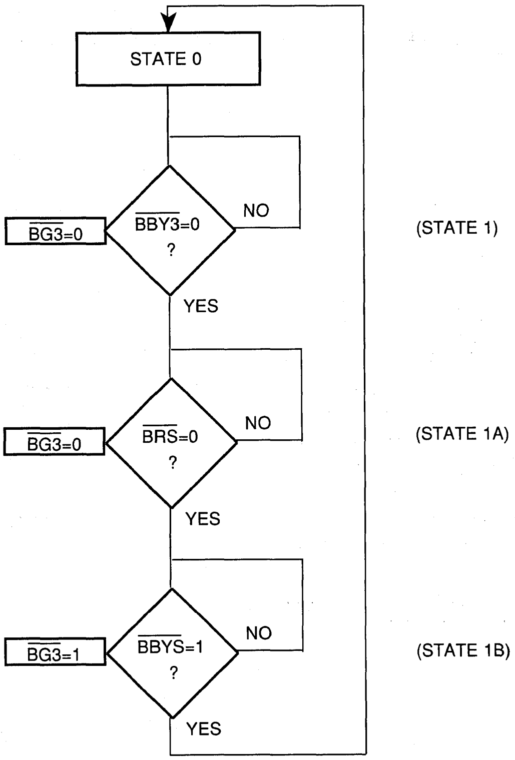
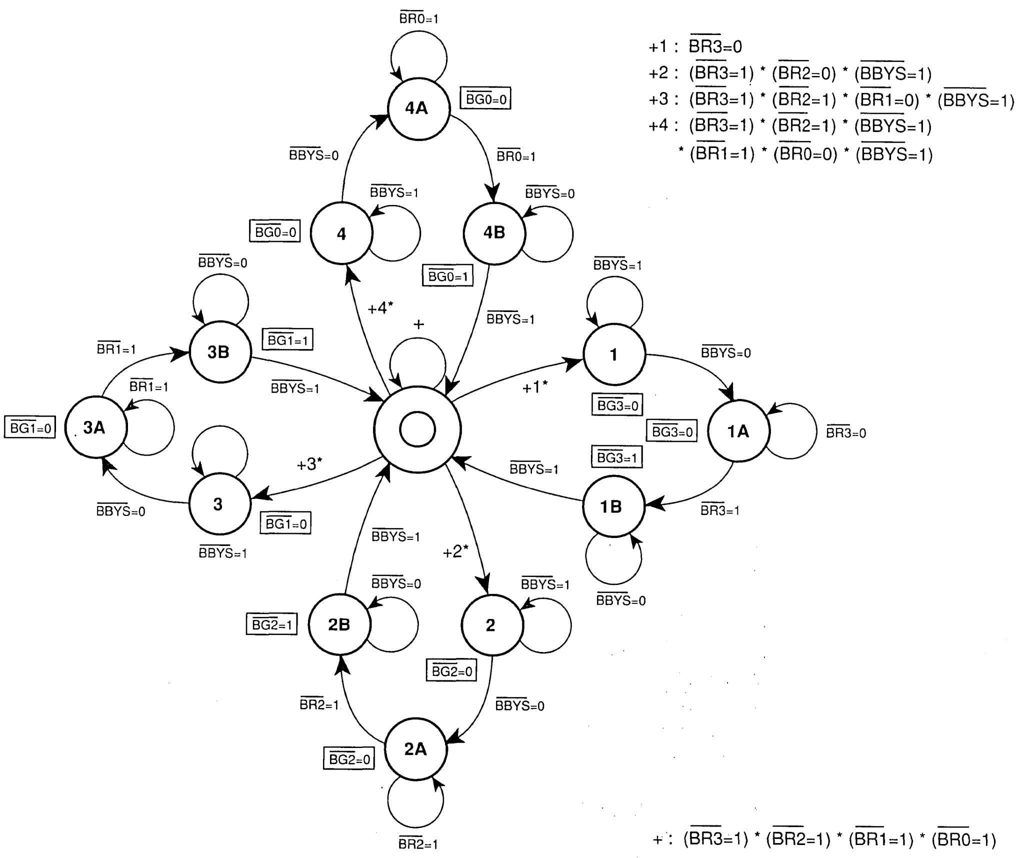


Figure 2-47. Program Flow for State 1 Microprogram



+1 : $\overline{BR3}$=0
+2 : ($\overline{BR3}$=1) * ($\overline{BR2}$=0) * ($\overline{BBYS}$=1)
+3 : ($\overline{BR3}$=1) * ($\overline{BR2}$=1) * ($\overline{BR1}$=0) * ($\overline{BBYS}$=1)
+4 : ($\overline{BR3}$=1) * ($\overline{BR2}$=1) * ($\overline{BBYS}$=1)
* ($\overline{BR1}$=1) * ($\overline{BR0}$=0) * ($\overline{BBYS}$=1)
+ : ($\overline{BR3}$=1) * ($\overline{BR2}$=1) * ($\overline{BR1}$=1) * ($\overline{BR0}$=1)
4A
4
4B
3A
3B
3
1
1A
1B
2
2A
2B
+1*
+2*
+3*
+4*
+
$\overline{BR0}$=1
$\overline{BG0}$=0
$\overline{BG0}$=1
$\overline{BBYS}$=0
$\overline{BBYS}$=1
$\overline{BR1}$=1
$\overline{BG1}$=0
$\overline{BG1}$=1
$\overline{BG3}$=0
$\overline{BG3}$=1
$\overline{BR3}$=0
$\overline{BR3}$=1
$\overline{BG2}$=0
$\overline{BG2}$=1
$\overline{BR2}$=1

Figure 2-48. State Diagram for Bus Arbiter

DEVICE (PL141)

DEFINE $\overline{BBYS}$=CC
$\overline{BR3}$=TO
$\overline{BG3}$=FE#H;

BEGIN .ORG 20#H

STATE1: $\overline{BG3}$, WHILE ($\overline{BBYS}$) WAIT ELSE GOTO PL (STATE 1A);
" STAY IN STATE 1 UNTIL $\overline{BBYS}$ GOES LOW "
" BRING $\overline{BG3}$ LOW "

STATE1A: $\overline{BG3}$, WHILE (NOT $\overline{BR3}$) WAIT ELSE GOTO PL (STATE 1B);
" STAY IN STATE 1A UNTIL $\overline{BR3}$ GOES HIGH "
" KEEP $\overline{BG3}$ LOW "

STATE1B: IDLE, WHILE (NOT $\overline{BBYS}$) WAIT ELSE GOTO PL (STATE 0);
" STAY IN STATE 1B UNTIL $\overline{BBYS}$ GOES HIGH "
" BRING $\overline{BG3}$ HIGH "

Figure 2-49. Microprogram for State 1

```
DEVICE    (PL141)

DEFINE    BR3  = T0
          BR2  = T1
          BR1  = T2
          BR0  = T3
         BBYS  = CC
          FAIL = EQ
          BG3  = FE#H
          BG2  = FD#H
          BG1  = FB#H
          BG0  = F7#H
          IDLE = FF#H;

BEGIN
          .ORG 0#H " JUMP TABLE "
LOC0 :  BG3 , IF (NOT FAIL) THEN GOTO PL (STATE 1) ;  "   1   "
LOC1 :  BG2 , IF (NOT FAIL) THEN GOTO PL (STATE 2) ;  "   1   "
LOC2 :  BG3 , IF (NOT FAIL) THEN GOTO PL (STATE 1) ;  "   1   "
LOC3 :  BG1 , IF (NOT FAIL) THEN GOTO PL (STATE 3) ;  "   1   "
LOC4 :  BG3 , IF (NOT FAIL) THEN GOTO PL (STATE 1) ;  "   1   "
LOC5 :  BG2 , IF (NOT FAIL) THEN GOTO PL (STATE 2) ;  "   1   "
LOC6 :  BG3 , IF (NOT FAIL) THEN GOTO PL (STATE 1) ;  "   1   "
LOC7 :  BG0 , IF (NOT FAIL) THEN GOTO PL (STATE 4) ;  " JUMP TABLE "
LOC8 :  BG3 , IF (NOT FAIL) THEN GOTO PL (STATE 1) ;  "   1   "
LOC9 :  BG2 , IF (NOT FAIL) THEN GOTO PL (STATE 2) ;  "   1   "
LOCA :  BG3 , IF (NOT FAIL) THEN GOTO PL (STATE 1) ;  "   1   "
LOCB :  BG1 , IF (NOT FAIL) THEN GOTO PL (STATE 3) ;  "   1   "
LOCC :  BG3 , IF (NOT FAIL) THEN GOTO PL (STATE 1) ;  "   1   "
LOCD :  BG2 , IF (NOT FAIL) THEN GOTO PL (STATE 2) ;  "   1   "
LOCE :  BG3 , IF (NOT FAIL) THEN GOTO PL (STATE 1) ;  "   1   "

          .ORG F#H
STATE0 :  IDLE , IF (BBYS) THEN GOTO TM (F#H) :

                " MICROPROGRAM FOR STATE 0 "
    " SAMPLE BUS REQUEST LINES AND VECTOR TO APPROPRIATE STATE "
```

Figure 2-50. Microprogram for Bus Arbiter

however, that by changing the addresses in the jump table, the priority imposed on the bus request lines can be altered. We exploit this fact in the design of the VME bus arbiter.

The four priority tables for the VME bus arbiter are shown in Figure 2-56. Each time the arbiter goes through an arbitration cycle it uses each of these tables in succession. As we discovered above, a priority table is implemented by a jump table working in conjunction with a GOTOTM microinstruction. Since we now have four priority tables, we need four separate jump tables.

Figure 2-57 shows an overview of a microprogram that implements the VME bus arbiter (also see Figure 2-58). As shown, there are four jump tables, one for each priority table. In addition, since each one of these jump tables needs to be in a separate region of program memory, we have four separate GOTOTM microinstructions.

The microinstructions for States 1–4 are duplicated four times, the reason being that the last WAIT microinstruction in each of these programs must be modified to point to a new State 0 so as to switch to a new priority table. However, the fundamental functions of each of these microprograms remain the same, namely, to implement the protocol for the $\overline{BR}$, $\overline{BG}$ and $\overline{BBYS}$ lines for a selected request level.

Recall, now, that the bus arbiter of Example 3 re-

```
                  " STATE 1 MICROPROGRAM "
STATE1:    BG3 , WHILE (BBYS)       WAIT ELSE GOTO PL (STATE1A)  ;" STAY IN STATE 1 UNTIL BBYS GOES LOW "
                                                                  " BRING BG3 LOW "
STATE1A:   BG3 , WHILE (NOT BR3)    WAIT ELSE GOTO PL (STATE1B)  ;" STAY IN STATE 1A UNTIL BR3 GOES HIGH "
                                                                  " KEEP BG3 LOW "
STATE1B:   IDLE , WHILE (NOT BBYS)  WAIT ELSE GOTO PL (STATE0)   ; " STAY IN STATE 1B UNTIL BBYS GOES HIGH "
                                                                  " BRING BG3 HIGH "

                  " STATE 2 MICROPROGRAM "
STATE2:    BG2 , WHILE (BBYS)       WAIT ELSE GOTO PL (STATE2A)  ; " WAIT FOR  BBYS TO GO LOW; ASSERT BG2 "

STATE2A:   BG2 , WHILE (NOT BR2)    WAIT ELSE GOTO PL (STATE2B)  ; " WAIT FOR  BR2 TO GO HIGH; ASSERT BG2 "

STATE2B:   IDLE , WHILE (NOT BBYS) WAIT ELSE GOTO PL (STATE0)    ; " WAIT FOR  BBYS TO GO HIGH; BRING BG2 HIGH "


                  " STATE 3 MICROPROGRAM "
STATE3:    BG1 , WHILE (BBYS)       WAIT ELSE GOTO PL (STATE3A)  ; " WAIT FOR BBYS TO GO LOW; ASSERT BG1 "

STATE3A:   BG1 , WHILE (NOT BR1)    WAIT ELSE GOTO PL (STATE3B)  ; " WAIT FOR BR1 TO GO HIGH; ASSERT BG1 "

STATE3B:   IDLE , WHILE (NOT BBYS)WAIT ELSE GOTO PL (STATE0)     ; " WAIT FOR BBYS TO GO HIGH; BRING BG1 HIGH "


                  " STATE 4 MICROPROGRAM "
STATE4:    BG0 , WHILE (BBYS)       WAIT ELSE GOTO PL (STATE4A)  ; " WAIT FOR BBYS TO GO LOW; ASSERT BG0 "

STATE4A:   BG0 , WHILE (NOT BR0)    WAIT ELSE GOTO PL (STATE4B)  ; " WAIT FOR BR0 TO GO HIGH; ASSERT BG0 "

STATE4B:   IDLE , WHILE (NOT BBYS) WAIT ELSE GOTO PL (STATE0)    ; " WAIT FOR BBYS TO GO HIGH; ASSERT BG0 HIGH "
                                                                  " RETURN TO STATE 0 "

                  .ORG 3F#H
     PON : IDLE , WHILE (NOT BBYS) WAIT ELSE GOTO PL (STATE0)    ; " WAIT UNTIL BBYS GOES HIGH "
                                                                  " BEFORE STARTING MICROPROGRAM "
```

Figure 2-50 (Continued). Microprogram for Bus Arbiter

quired 28 memory locations as allocated in the table below:

Jump Table	15
State 0	1
State 1–State 1B	3
State 2–State 2B	3
State 3–State 3B	3
State 4–State 4B	3

Therefore, to implement the VME bus arbiter, we need 112 program locations since we are essentially duplicating the microprogram of Example 3 four times. However, the Am29PL141 has only 64 program locations. Consequently, we will use the Am29PL142 for this example given its 128 program locations.

Let's review how the GOTOTM microinstruction works. As described in Example 3, the TEST inputs are used as an index into a jump table. Right now we are using the four low-order TEST inputs that are connected to the four bus request lines. However, this implies that we can have only one jump table and that it must be located in the first 15 locations of program memory. How, then, can we implement four jump tables?

Figure 2-59 shows a circuit diagram of the Am29PL142 in which two of the outputs are coupled back into the TEST inputs. These two outputs are used to locate the four jump tables within the first 64 locations of the Am29PL142.

To see how this works, look at Figures 2-60 and 2-61. As shown, we modify the OUTPUT fields of the microinstructions corresponding to State 0(1)–State 0(4) so as to introduce an offset to the jump table location. For example, if the microprogram branches to the GOTOTM microinstruction located at 2F (labelled State 0(3)), it remains there if the low-order four TEST inputs are all high. Then, if any of the four TEST inputs go low, a branch is taken to the appropriate location within the range 2O–2E, the locations of the jump table for priority 3.

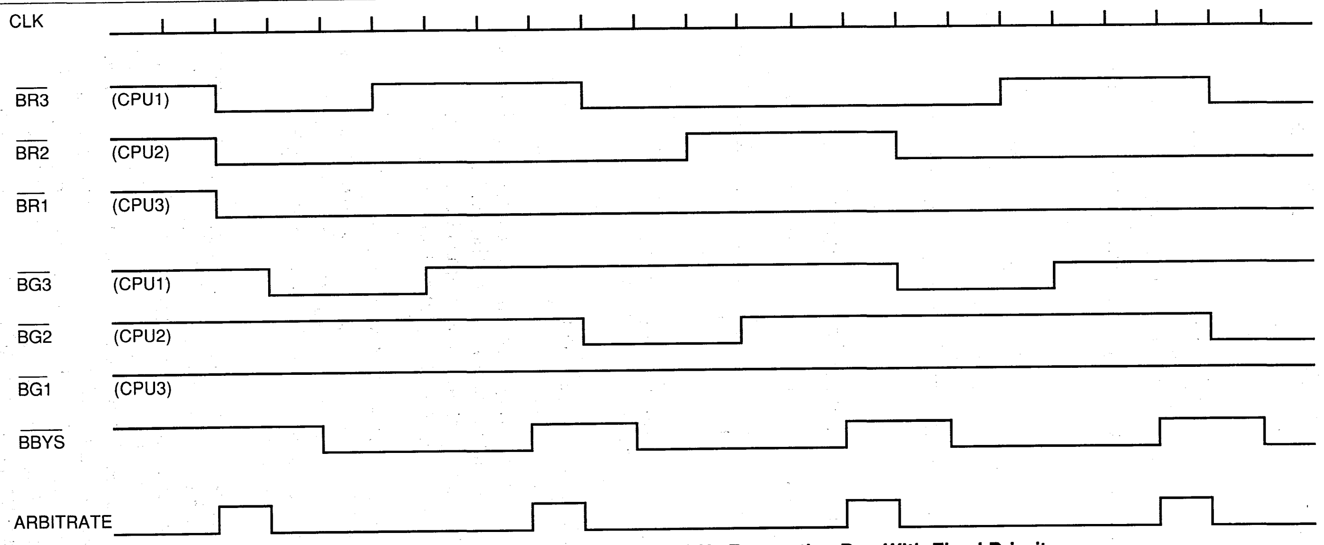


Figure 2-51. Timing Diagram for Three CPUs Requesting Bus With Fixed Priority

CPU	BUS REQUEST LINE
1	$\overline{BR3}$
2	$\overline{BR2}$
3	$\overline{BR1}$

PRI1

CPU	PRIORITY
1	1
2	2
3	3

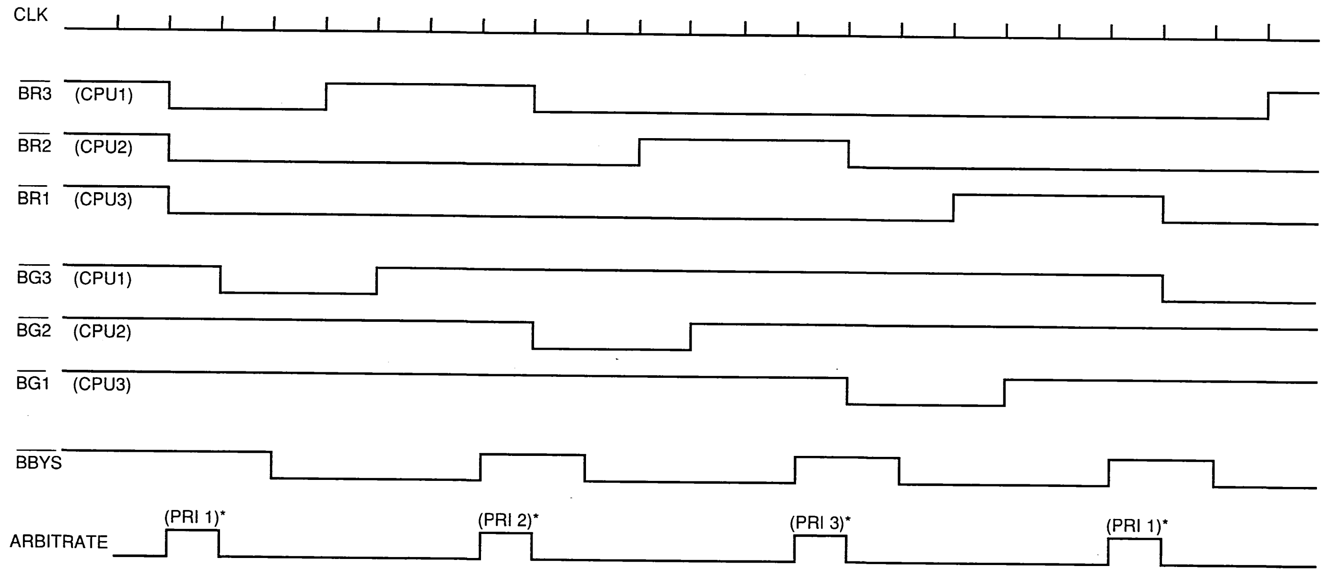
PRI2

CPU	PRIORITY
1	2
2	3
3	1

PRI3

CPU	PRIORITY
1	3
2	1
3	2

Figure 2-52. Three Priority Tables for Rotating Priority Arbiter



* THESE REPRESENT THE PRIORITY TABLES FROM FIGURE 2-52 THAT ARE USED IN THE ARBITRATION PROCESS

Figure 2-53. Timing Diagram for Three CPUs Requesting Bus With Rotating Priority

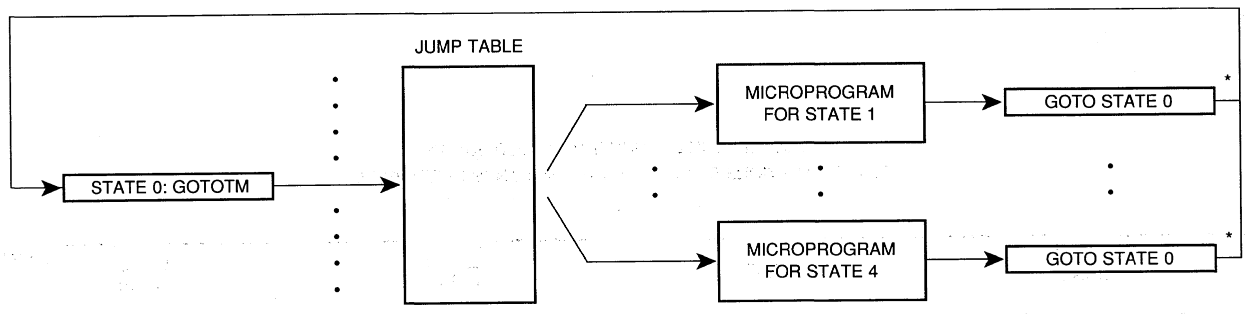
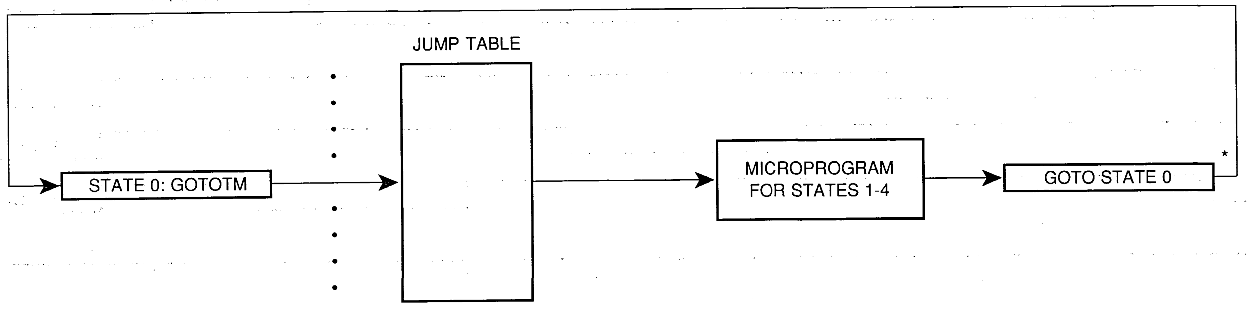


Figure 2-54. Structure of Microprogram for Bus Arbiter in Example 3



* THIS STATEMENT REPRESENTS THE ELSE CLAUSE OF THE WAIT MICROINSTRUCTION:
I.E., " IDLE , WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL (STATE 0) "

Figure 2-55. Condensed Structure of Microprogram for Bus Arbiter in Example 3

CPU	BUS REQUEST LINE
1	$\overline{\text{BR3}}$
2	$\overline{\text{BR2}}$
3	$\overline{\text{BR1}}$
4	$\overline{\text{BRO}}$

PRI 1

CPU	PRIORITY
1	1
2	2
3	3
4	4

PRI 2

CPU	PRIORITY
1	2
2	3
3	4
4	1

PRI 3

CPU	PRIORITY
1	3
2	4
3	1
4	2

PRI 4

CPU	PRIORITY
1	4
2	1
3	2
4	3

Figure 2-56 Priority Tables for VME Bus Arbiter

Figure 2-62 shows the state transition tables corresponding to each one of the priority tables. Based on the same procedure described in Example 3, these tables are used to fill in the branch addresses for every location in the jump tables (see Figure 2-63).

In contrast to the Am29PL141, the Am29PL142 clocks the TEST inputs into a TEST register. As a result, a microinstruction using the TEST inputs operates upon the value of the TEST inputs that existed one clock cycle earlier.

Let's illustrate the effects of this 1-clock delay by an example. The intent of the microprogram in Figure 2-65 is to load the counter (CREG) with the values 1, 2 and 3 in succession (also see Figure 2-64). However, as shown in the timing diagram of Figure 2-66, the CREG remains zero because of the 1-clock delay. The microprogram in Figure 2-67 mitigates this effect by having the microinstructions preceding the LDTM microinstructions present the desired values at the TEST inputs (see Figure 2-68).

Taking this 1-clock delay into consideration, we now see that the modifications of the GOTOTM microinstructions in Figure 2-60 are insufficient to implement the four jump tables. In fact, executing any one of these microinstructions causes a branch to the jump table for priority 1. To correct this, we also need to modify the microinstructions shown in Figure 2-65, since these execute one clock period before the GOTOTM microinstructions.

We now have successfully written a microprogram to implement the VME bus arbiter. In looking over Figure 2-57, however, it seems redundant to duplicate the microprogram for States 1–4 (from Example 3) four times. After all, functionally, each microprogram behaves identically. In fact, the only reason for the duplication is to provide a means to branch to a new jump table.

Figure 2-70 presents an overview of an alternate implementation of the VME bus arbiter. As shown, the microprogram for States 1–4 now is made into a microsubroutine, thus eliminating the need for duplication. With this structure, executing a particular CALL TM microinstruction causes a branch to the proper jump table (see Figure 2-71). Then, when the subsequent RET microinstruction is executed (see Figure 2-72), control returns to the statement following this CALL TM microinstruction. A new jump table then is selected by executing the next CALL TM microinstruction. Finally, notice that the jump tables in Figure 2-63 are modified to fit this new structure (see Figure 2-73).

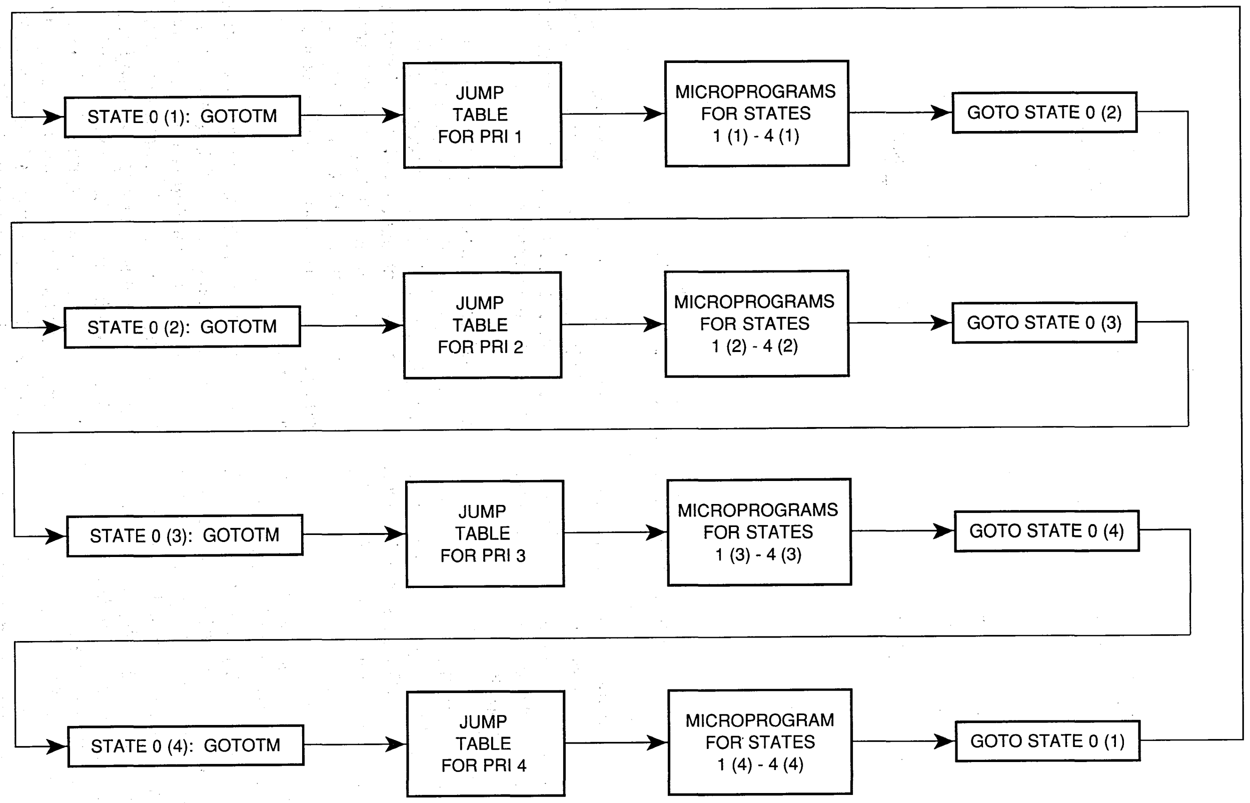


Figure 2-57.Condensed Structure of Microprogram for VME Bus Arbiter

FORMAT	INTERPRETATION
STATE 0 (N)	STATE 0 USING PRIORITY TABLE N
STATE 1 (N)	STATE 1 USING PRIORITY TABLE N
STATE 2 (N)	STATE 2 USING PRIORITY TABLE N
STATE 3 (N)	STATE 3 USING PRIORITY TABLE N
STATE 4 (N)	STATE 4 USING PRIORITY TABLE N
	WHERE N = 1 - 4

Figure 2-58. Nomenclature Used in Figure 2-57

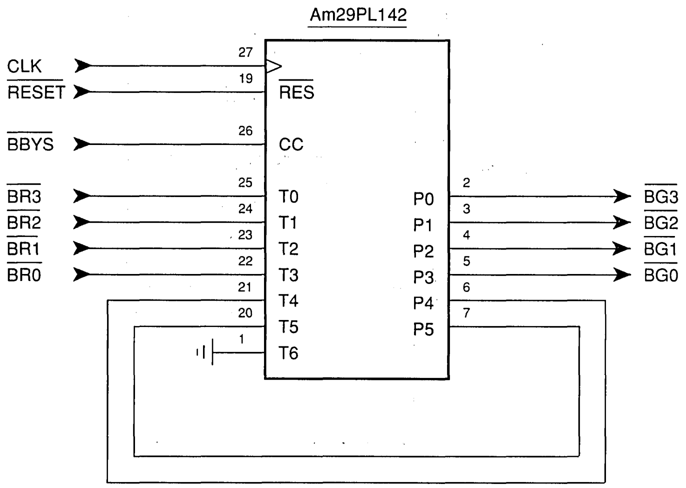


Figure 2-59. Am29PL142-Based VME Bus Arbiter

```
DEFINE          PRI 1 = 0F #H
                PRI 2 = 1F #H
                PRI 3 = 2F #H
                PRI 4 = 3F #H;

                .ORG F #H
STATE0 (1) :    PRI 1, IF (BBYS) THEN GOTO TM (3F #H);
                .ORG 1F #H
STATE0 (2) :    PRI 2, IF (BBYS) THEN GOTO TM (3F #H);
                .ORG 2F #H
STATE0 (3) :    PRI 3, IF (BBYS) THEN GOTO TM (3F #H);
                .ORG 3F #H
STATE0 (4) :    PRI 4, IF (BBYS) THEN GOTO TM (3F #H);
```

Figure 2-60. Modification of GOTOTM Microinstructions to Implement Four Jump Tables

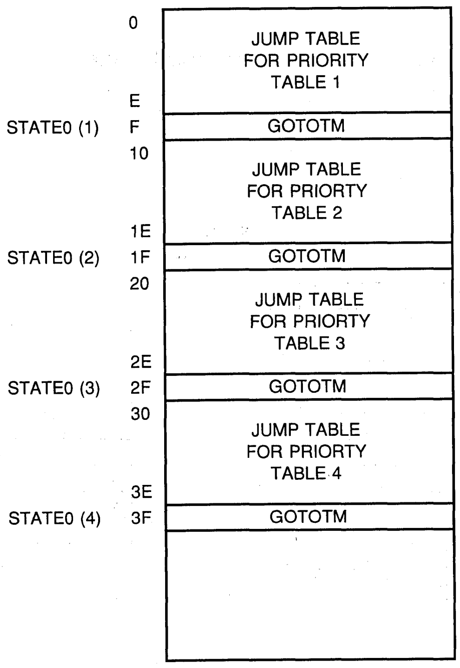

0
JUMP TABLE FOR PRIORITY TABLE 1
E
STATE0 (1)
F
GOTOTM
10
JUMP TABLE FOR PRIORTY TABLE 2
1E
STATE0 (2)
1F
GOTOTM
20
JUMP TABLE FOR PRIORTY TABLE 3
2E
STATE0 (3)
2F
GOTOTM
30
JUMP TABLE FOR PRIORTY TABLE 4
3E
STATE0 (4)
3F
GOTOTM

Figure 2-61. Location of Jump Tables within Program Address Space

PRI 1

$\overline{BR0}$	$\overline{BR1}$	$\overline{BR2}$	$\overline{BR3}$	HEX EQUIV* @ TEST INPUT	NEXT STATE
0	0	0	0	0	1
0	0	1	0	2	1
0	1	0	0	4	1
0	1	1	0	6	1
1	0	0	0	8	1
1	0	1	0	A	1
1	1	0	0	C	1
1	1	1	0	E	1
0	0	0	1	1	2
0	1	0	1	5	2
1	0	0	1	9	2
1	1	0	1	D	2
0	0	1	1	3	3
1	0	1	1	B	3
0	1	1	1	7	4
1	1	1	1	F	0

PRI 2

$\overline{BR3}$	$\overline{BR0}$	$\overline{BR1}$	$\overline{BR2}$	HEX EQUIV* @ TEST INPUT	NEXT STATE
0	0	0	0	0	2
0	0	1	0	4	2
0	1	0	0	8	2
0	1	1	0	C	2
1	0	0	0	1	2
1	0	1	0	5	2
1	1	0	0	9	2
1	1	1	0	D	2
0	0	0	1	2	3
0	1	0	1	A	3
1	0	0	1	3	3
1	1	0	1	B	3
0	0	1	1	6	4
1	0	1	1	7	4
0	1	1	1	E	1
1	1	1	1	F	0

* NOTE: THIS COLUMN CONTAINS THE HEX EQUIVALENT AT THE TEST INPUTS. THE $\overline{BR0}$ - $\overline{BR3}$ COLUMNS MUST BE REARRANGED IN THE SEQUENCE $\overline{BR0}$ $\overline{BR1}$ $\overline{BR2}$ $\overline{BR3}$ TO ARRIVE AT THIS NUMBER. THE $\overline{BR0}$ - $\overline{BR3}$ COLUMNS ARE AS SHOWN TO FACILITATE THE CALCULATION OF THE NEXT STATE.

Figure 2-62. State Transition Tables for Each Priority Table

PRI 3					
$\overline{BR2}$	$\overline{BR3}$	$\overline{BR0}$	$\overline{BR1}$	HEX EQUIV* @ TEST INPUT	NEXT STATE
0	0	0	0	0	3
0	0	1	0	8	3
0	1	0	0	1	3
0	1	1	0	9	3
1	0	0	0	2	3
1	0	1	0	A	3
1	1	0	0	3	3
1	1	1	0	B	3
0	0	0	1	4	4
0	1	0	1	5	4
1	0	0	1	6	4
1	1	0	1	7	4
0	0	1	1	C	1
1	0	1	1	E	1
0	1	1	1	D	2
1	1	1	1	F	0

PRI 4					
$\overline{BR1}$	$\overline{BR2}$	$\overline{BR3}$	$\overline{BR0}$	HEX EQUIV* @ TEST INPUT	NEXT STATE
0	0	0	0	0	4
0	0	1	0	1	4
0	1	0	0	2	4
0	1	1	0	3	4
1	0	0	0	4	4
1	0	1	0	5	4
1	1	0	0	6	4
1	1	1	0	7	4
0	0	0	1	8	1
0	1	0	1	A	1
1	0	0	1	C	1
1	1	0	1	E	1
0	0	1	1	9	2
1	0	1	1	D	2
0	1	1	1	B	3
1	1	1	1	F	0

Figure 2-62. (Continued)

LABEL	LOC	BRANCH ADDRESS		LABEL	LOC	BRANCH ADDRESS	
	0	STATE 1 (1)			20	STATE 3 (3)	
	1	STATE 2 (1)			21	STATE 3 (3)	
	2	STATE 1 (1)			22	STATE 3 (3)	
	3	STATE 3 (1)			23	STATE 3 (3)	
	4	STATE 1 (1)			24	STATE 4 (3)	
	5	STATE 2 (1)			25	STATE 4 (3)	
	6	STATE 1 (1)			26	STATE 4 (3)	
	7	STATE 4 (1)	PRI 1 JUMP TABLE		27	STATE 4 (3)	PRI 3 JUMP TABLE
	8	STATE 1 (1)			28	STATE 3 (3)	
	9	STATE 2 (1)			29	STATE 3 (3)	
	A	STATE 1 (1)			2A	STATE 3 (3)	
	B	STATE 3 (1)			2B	STATE 3 (3)	
	C	STATE 1 (1)			2C	STATE 1 (3)	
	D	STATE 2 (1)			2D	STATE 2 (3)	
	E	STATE 1 (1)			2E	STATE 1 (3)	
STATE 0 (1)	F	*		STATE 0 (3)	2F	* * *	
	10	STATE 2 (2)			30	STATE 4 (4)	
	11	STATE 2 (2)			31	STATE 4 (4)	
	12	STATE 3 (2)			32	STATE 4 (4)	
	13	STATE 3 (2)			33	STATE 4 (4)	
	14	STATE 2 (2)			34	STATE 4 (4)	
	15	STATE 2 (2)			35	STATE 4 (4)	
	16	STATE 4 (2)			36	STATE 4 (4)	
	17	STATE 4 (2)	PRI 2 JUMP TABLE		37	STATE 4 (4)	PRI 4 JUMP TABLE
	18	STATE 2 (2)			38	STATE 1 (4)	
	19	STATE 2 (2)			39	STATE 2 (4)	
	1A	STATE 3 (2)			3A	STATE 1 (4)	
	1B	STATE 3 (2)			3B	STATE 3 (4)	
	1C	STATE 2 (2)			3C	STATE 1 (4)	
	1D	STATE 2 (2)			3D	STATE 2 (4)	
	1E	STATE 1 (2)			3E	STATE 1 (4)	
STATE 0 (2)	1F	* *		STATE 0 (4)	3F	* * * *	

*	CONTAINS MICROINSTRUCTION:	PRI 1^{+} , IF ($\overline{BBYS}$) THEN GOTOTM (3F #H)
* *	CONTAINS MICROINSTRUCTION:	PRI 2^{+} , IF ($\overline{BBYS}$) THEN GOTOTM (3F #H)
* * *	CONTAINS MICROINSTRUCTION:	PRI 3^{+} , IF ($\overline{BBYS}$) THEN GOTOTM (3F #H)
* * * *	CONTAINS MICROINSTRUCTION:	PRI 4^{+} , IF ($\overline{BBYS}$) THEN GOTOTM (3F #H)

$^{+}$ THESE ARE OUTPUT FIELD DEFINITIONS

PRI 1 = F #H	PRI 3 = 2F #H
PRI 2 = 1F #H	PRI 4 = 3F #H

Figure 2-63. Jump Table Contents for Each Priority Table

This alternative implementation of the VME bus arbiter is more compact than the previous implementation, but only at the expense of lengthening the arbitration time. With the first implementation, the arbitration process needed only one clock cycle. In contrast, the second implementation introduces two additional clock delays, one at the execution of the RET microinstruction, and the other at the execution of the microinstruction following a CALL TM microinstruction.

2.2.5 Example 5: Frame Store

So far we have used the Am29PL100 to implement a burst counter, a memory controller and two bus arbiters. In addition to these applications, we also can use the Am29PL142 to implement the timing and control logic of a frame store, which is used to display grey-scale images on a CRT monitor. A frame store consists of these basic elements (see Figures 2-74 and 2-75):

- Buffer memory. The buffer memory holds the pixel values corresponding to each one of the pixels displayed on the monitor. In our design, each pixel is represented by a byte. As a result, images with 256 shades of grey can be displayed. Also, since our display is organized as 512 pixels x 512 pixels, the memory is 256 Kbytes.

  The buffer memory is dual-ported, with one port dedicated to the CPU and the other to the display logic. The CPU writes pixel values into the buffer memory, while the display logic reads them out for display.

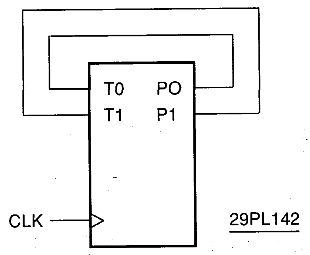


Figure 2-64. Circuit Used in Figures 2-65 and 2-67

STATEMENT #	LABEL	MICROINSTRUCTION
1	START:	1 #H, IF (PASS)* THEN LOAD TM (3 #H);
2		0 #H, CONT;
3		2 #H, IF (PASS) THEN LOAD TM (3 #H);
4		0 #H, CONT;
5		3 #H, IF (PASS) THEN LOAD TM (3 #H);
6		0 #H, IF (PASS) THEN GOTO PL (START);

Figure 2-65. First Example of Effect of Clocking Test Inputs

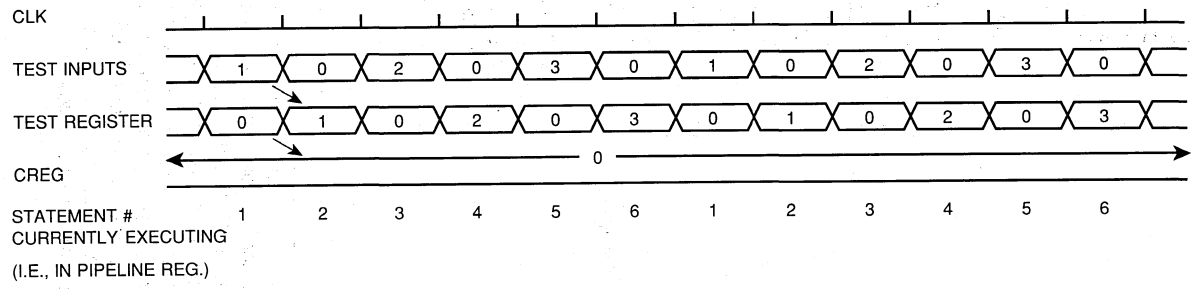


Figure 2-66. Timing Diagram for Microprogram in Figure 2-65

STATEMENT #	LABEL	MICROINSTRUCTION
1	START:	0 #H, IF (PASS) THEN LOAD TM (3 #H);
2		2 #H, CONT;
3		0 #H, IF (PASS) THEN LOAD TM (3 #H);
4		3 #H, CONT;
5		0 #H, IF (PASS) THEN LOAD TM (3 #H);
6		1 #H, IF (PASS) THEN GOTO PL (START);

Figure 2-67. Second Microprogram Showing Effects of Clocking Test Inputs

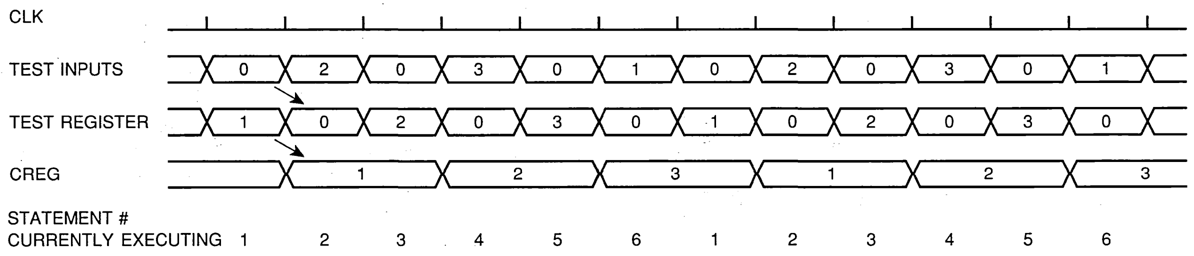


Figure 2-68. Timing Diagram for Microprogram in Figure 2-67

LABEL	OUTPUT	MICROINSTRUCTION
STATE 1B (1) :	PRI 2*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (2)];
STATE 2B (1) :	PRI 2*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (2)];
STATE 3B (1) :	PRI 2*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (2)];
STATE 4B (1) :	PRI 2*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (2)];
STATE 1B (2) :	PRI 3*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (3)];
STATE 2B (2) :	PRI 3*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (3)];
STATE 3B (2) :	PRI 3*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (3)];
STATE 4B (2) :	PRI 3*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (3)];
STATE 1B (3) :	PRI 4*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (4)];
STATE 2B (3) :	PRI 4*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (4)];
STATE 3B (3) :	PRI 4*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (4)];
STATE 4B (3) :	PRI 4*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (4)];
STATE 1B (4) :	PRI 1*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (1)];
STATE 2B (4) :	PRI 1*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (1)];
STATE 3B (4) :	PRI 1*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (1)];
STATE 4B (4) :	PRI 1*,	WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL [STATE 0 (1)];

* PRI 1 = F #H
PRI 2 = 1F #H
PRI 3 = 2F #H
PRI 4 = 3F #H

Figure 2-69. Modifications to VME Arbiter Microprogram to Mitigate Effect of Clocking Test Inputs

- Display logic. The display logic generates two fundamental types of signals:

  Control signals to the CRT — Starting at the upper left side of the screen, the electron beam in the CRT moves horizontally, from left to right, to display each row of pixel values, and vertically, from top to bottom, until all rows are displayed.

  As the electron beam moves from left to right, each pixel is displayed. When all 512 pixels have been displayed, the electron beam is at the right hand side of the screen. At this point, the display logic issues a horizontal sync pulse (HSYNC) that causes the electron beam to return to the left side of the screen and down one row.

  After all 512 rows of pixels have been displayed, the electron beam is at the bottom right of the display. The display logic then generates a vertical sync pulse (VSYNC) that causes the electron beam to return to the upper left side of the screen to repeat the cycle.

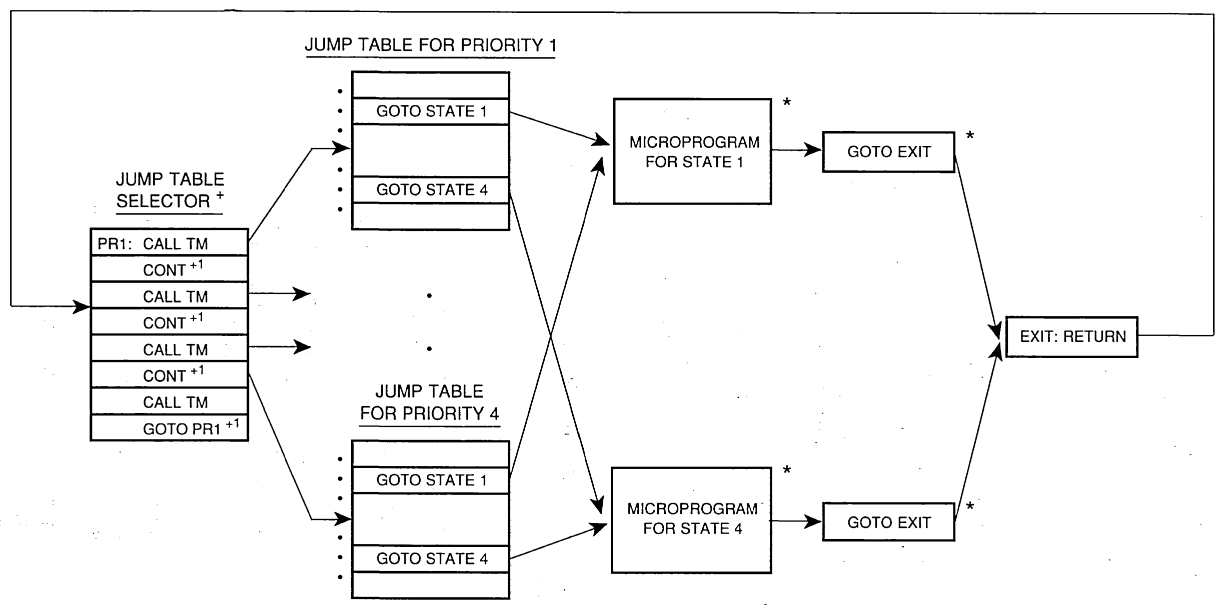
  Memory request signal to memory controller — As the electron beam sweeps across the screen, successive pixel values must be available for display. Consequently, the display logic issues a memory request signal to the memory controller to obtain the proper pixel value at the right time.

- Memory controller. The CPU writes pixel values into the buffer memory, which then are displayed under control of the display logic. Therefore, the memory controller must arbitrate between CPU requests and display logic memory requests and then generate the proper sequence of memory control signals.

2.2.5.1 Implementing the Buffer Memory

As noted above, the buffer memory is dual-ported to allow access by both the CPU and display. In our design, we implement this memory using a special type of dynamic memory chip called a Video Ram (VRAM).

As shown in Figure 2-76, the VRAM consists of four 64-Kbit planes, each plane organized as 256 rows and 256 columns. In addition, each plane has a 256-bit shift register that can be loaded with the contents



+ SEE FIGURE 2-71
* SEE FIGURE 2-72
+1 TAKES CARE OF ONE CLOCK DELAY ON TEST INPUTS

Figure 2-70. Overview of Alternate Microprogram for VME Bus Arbiter

```
PR 1 :   PRI1,  IF ($\overline{\text{BBYS}}$) THEN CALL TM (3F #H); "GOTO PRI 1 JUMP TABLE"
         PRI2,  CONT; "ASSERT PRI 2 ONE CLOCK CYCLE EARLY"
         PRI2,  IF ($\overline{\text{BBYS}}$) THEN CALL TM (3F #H); "GOTO PRI 2 JUMP TABLE"
         PRI3,  CONT; "ASSERT PRI 3 ONE CLOCK CYCLE EARLY"
         PRI3,  IF ($\overline{\text{BBYS}}$) THEN CALL TM (3F #H); "GOTO PRI 3 JUMP TABLE"
         PRI4,  CONT;
         PRI4,  IF ($\overline{\text{BBYS}}$) THEN CALL TM (3F #H); "GOTO PRI 4 JUMP TABLE"
         PRI1,  IF (EQ) THEN GOTO PL (PR 1);
```

Figure 2-71. Microprogram for Jump Table Selector

```
STATE1B:  IDLE, WHILE (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL (EXIT);
          "MODIFICATION OF STATE 1 MICROPROGRAM"

STATE2B:  IDLE, WHILE  (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL (EXIT);
          "MODIFICATION OF STATE 2 MICROPROGRAM"

STATE3B:  IDLE, WHILE  (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL (EXIT);
          "MODIFICATION OF STATE 3 MICROPROGRAM"

STATE4B:  IDLE, WHILE  (NOT $\overline{\text{BBYS}}$) WAIT ELSE GOTO PL (EXIT);
          "MODIFICATION OF STATE 4 MICROPROGRAM"

EXIT :    IDLE, IF ($\overline{\text{BBYS}}$) THEN RET; "BRANCH TO JUMP TABLE"
                                          "SELECTOR MICROPROGRAM"
```

Figure 2-72. Modification of States 1-4 Microprogram to Use Microsubroutine Structure

STATE 1 (1)	CHANGES TO STATE 1
STATE 1 (2)	CHANGES TO STATE 1
STATE 1 (3)	CHANGES TO STATE 1
STATE 1 (3)	CHANGES TO STATE 1
STATE 2 (1)	CHANGES TO STATE 2
STATE 2 (2)	CHANGES TO STATE 2
STATE 2 (3)	CHANGES TO STATE 2
STATE 2 (4)	CHANGES TO STATE 2
STATE 3 (1)	CHANGES TO STATE 3
STATE 3 (2)	CHANGES TO STATE 3
STATE 3 (3)	CHANGES TO STATE 3
STATE 3 (4)	CHANGES TO STATE 3
STATE 4 (1)	CHANGES TO STATE 4
STATE 4 (2)	CHANGES TO STATE 4
STATE 4 (3)	CHANGES TO STATE 4
STATE 4 (4)	CHANGES TO STATE 4

Figure 2-73. Modification of Jump Tables in Figure 2-63

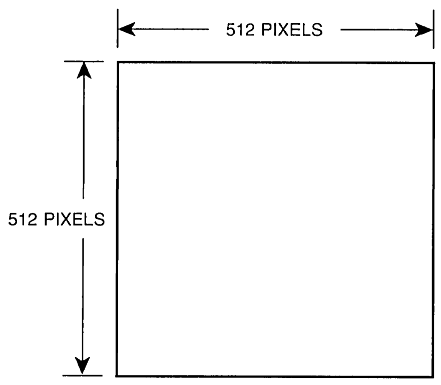


Figure 2-74. Pixel Organization of Display

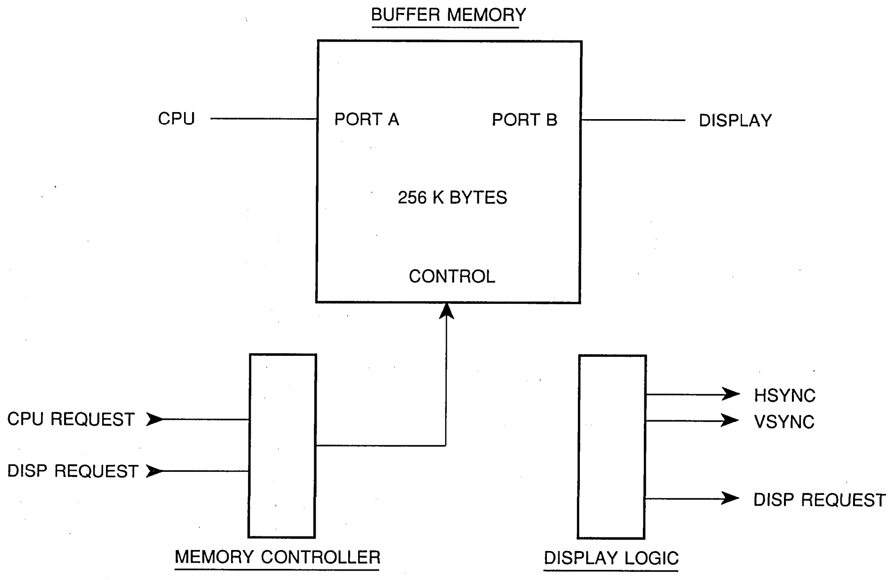


Figure 2-75. High-Level Block Diagram of Frame Store

of all 256 columns of a selected row in a single operation.

Figure 2-77 shows a 64-Kbyte memory system that uses two VRAMs. Aside from the shift register, the only difference between this memory system and the one used in Example 2 is the inclusion of an additional memory control signal called Transfer (TR).

$\overline{TR}$ is used to select one of two modes of operations. If TR is high during a memory cycle (see Figure 2-78), the random access mode is selected, and the VRAM behaves identically to the random-access dynamic memory in Example 2. Alternatively, if $\overline{TR}$ is low during a memory cycle (see Figure 2-79), the serial access mode is selected and the shift register is loaded with all the columns specified by the row address when $\overline{RAS}$ is brought low.

Because of the serial nature of pixel display, the VRAM is especially well suited to implementing display buffer memories. In this application, the CPU uses the random access mode and the display uses the serial access mode. With this arrangement, contention for memory access is minimal since the shift register is loaded only once every 256 shift pulses.

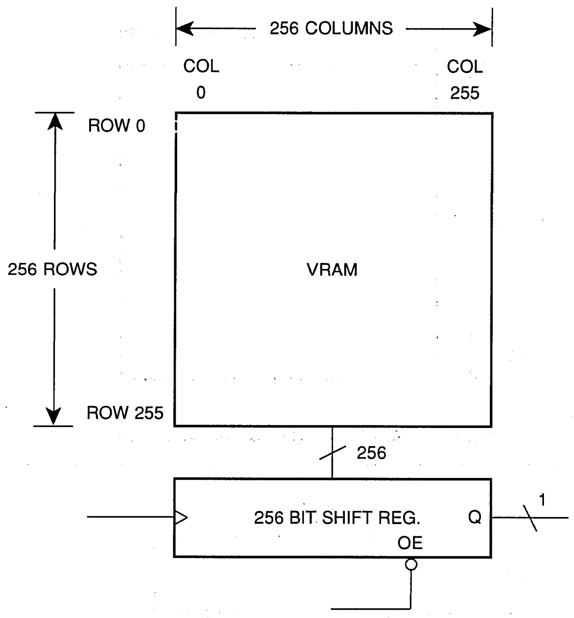


Figure 2-76. Organization of One Plane of VRAM

In a dynamic memory, data is represented by the absence or presence of an electron charge on a capacitor. Because of leakage paths, however, the charge on a capacitor slowly decays. To circumvent this problem, it is necessary to perform a refresh operation on the memory.

A refresh operation is performed by periodically reading each row in the dynamic memory. As a result, any capacitors that were initially charged are restored to their full value. For the VRAM described here, each row must be refreshed once every 4 milliseconds to retain data.

Figure 2-80 shows a timing diagram for a refresh operation. As shown, refresh is initiated by bringing $\overline{\text{CAS}}$ low before $\overline{\text{RAS}}$. The row address is provided by an internal counter within the memory chip that is specifically used for this purpose. At the end of each refresh operation, the counter is incremented in preparation for a subsequent cycle.

2.2.5.2 *Implementing the Display Logic*

In our design, the display is organized as 512 pixels x 512 pixels, with each pixel represented by a byte. As a result, the display buffer is 256 Kbytes, which can be implemented with 8 VRAMs.

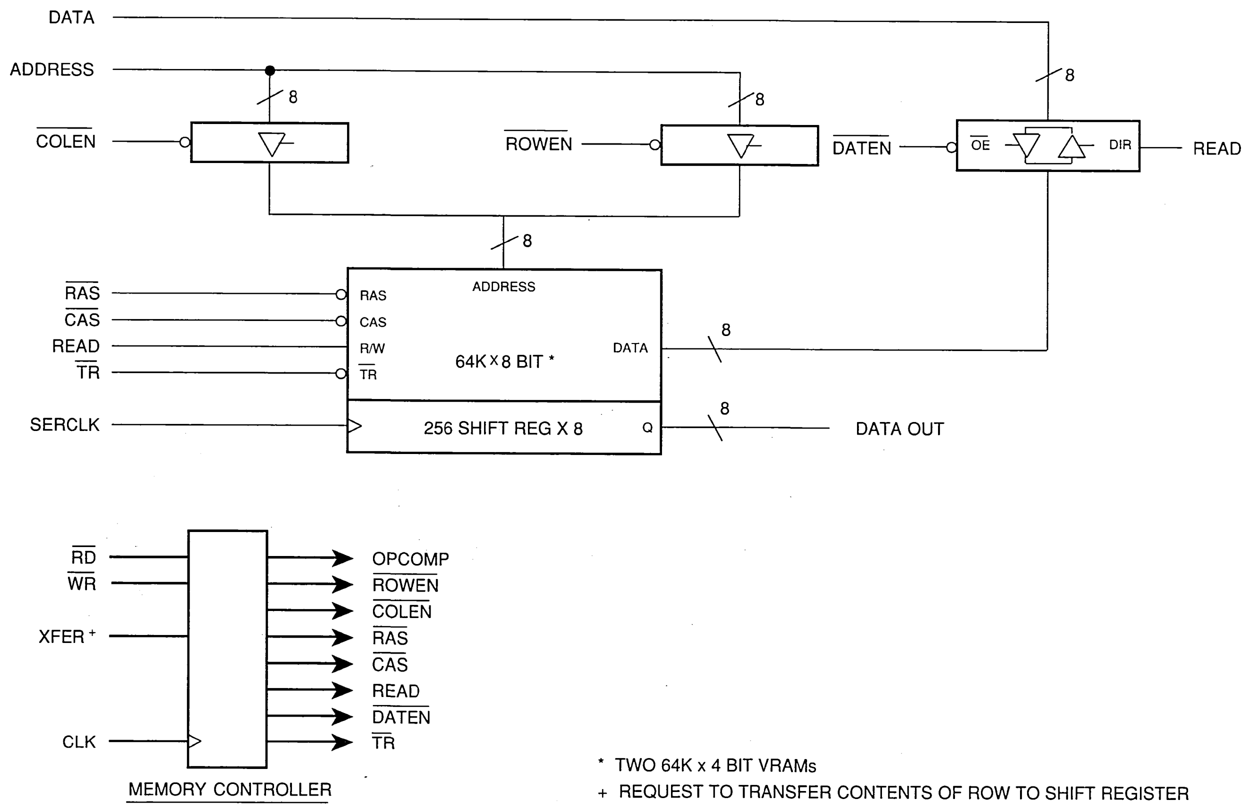
Figure 2-81 shows the physical relationship between the VRAMs and the display. As shown, each quadrant of the display needs 2 VRAMs since each VRAM contributes four bits of each pixel.

To the CPU, the display buffer appears as a continuous block of 256 Kbytes in which pixel values are stored sequentially in successive buffer locations (see Figure 2-82). To address an individual pixel location, CPU address bits A8 and A17 are decoded to select one of the four pairs of VRAMs (see Figures 2-83 and 2-84). The remaining address bits then specify the row and column addresses within the selected VRAM.

Figure 2-85 shows a block diagram of the address decoding mechanism. Notice that the memory control signal RAS is common to all VRAMs, but that CAS is routed to the individually selected VRAM by the address decoder. This arrangement allows the data lines to be tied together because $\overline{\text{CAS}}$, in addition to strobing the column addresses into the VRAMs, also controls the three-state condition of the output data drivers. Consequently, only the pair of VRAMs receiving both $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ is selected. For the other VRAMs, which receive $\overline{\text{RAS}}$ only, data remains valid.

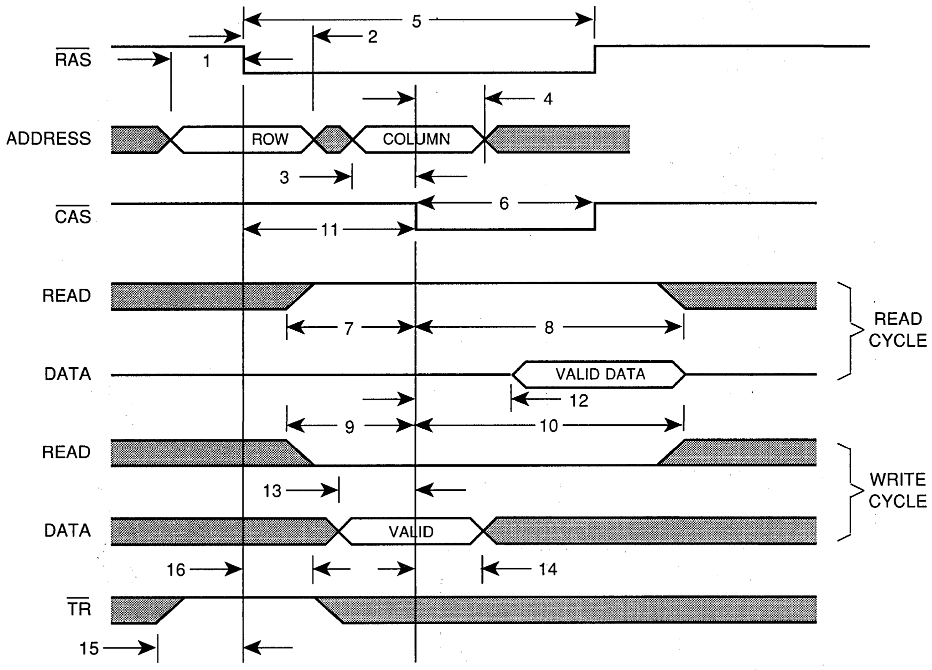
So far, our discussion of the display buffer has centered on the random access mode of the VRAMs, which is used by the CPU. However, as described above, the display uses the serial access mode. Before describing this mode more fully, let's take a look at the display logic in greater detail.

Figure 2-86 presents a high-level view of the display logic circuitry and display buffer. As shown, the display buffer contains a 512-bit shift register that holds the pixel values for an entire row in our display.



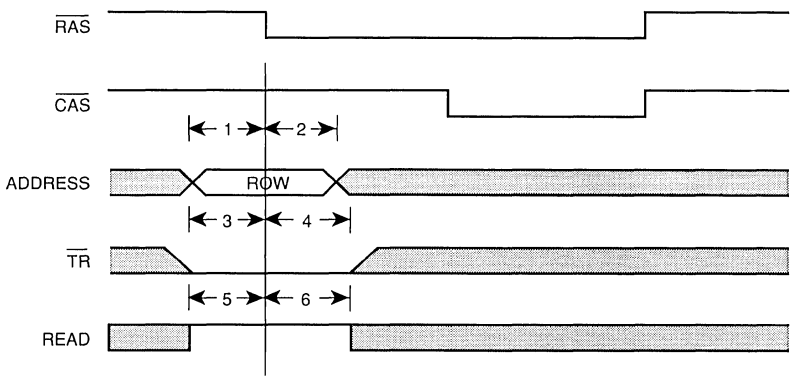
* TWO 64K x 4 BIT VRAMs
+ REQUEST TO TRANSFER CONTENTS OF ROW TO SHIFT REGISTER

Figure 2-77. 64K Byte Memory System Using VRAMs



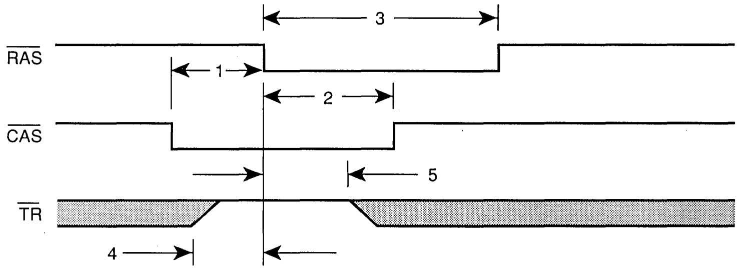
	MIN	MAX
1. ROW ADDRESS SETUP TIME	20	
2. ROW ADDRESS HOLD TIME	0	
3. COLUMN ADDRESS SETUP TIME	20	
4. COLUMN ADDRESS HOLD TIME	0	
5. PULSE DURATION $\overline{\text{RAS}}$ LOW	180	10000
6. PULSE DURATION $\overline{\text{CAS}}$ LOW	80	10000
7. READ COMMAND SETUP TIME	20	
8. READ COMMAND HOLD TIME	20	
9. WRITE COMMAND SETUP TIME	20	
10. WRITE COMMAND HOLD TIME	20	
11. $\overline{\text{RAS}}$ TO $\overline{\text{CAS}}$ DELAY	80	
12. ACCESS TIME FROM $\overline{\text{CAS}}$	80	
13. DATA IN SETUP TIME	20	
14. DATA IN HOLD TIME	20	
15. $\overline{\text{TR}}$ SETUP TIME	20	
16. $\overline{\text{TR}}$ HOLD TIME	0	

Figure 2-78. VRAM Timing Requirements



	MIN	MAX
1. ROW ADDRESS SETUP TIME	20	
2. ROW ADDRESS HOLD TIME	0	
3. TRANSFER SETUP TIME	20	
4. TRANSFER HOLD TIME	0	
5. READ COMMAND SETUP TIME	20	
6. READ COMMAND HOLD TIME	0	

Figure 2-79. Memory to Shift Register Timing



	MIN	MAX
1. $\overline{\text{CAS}}$ LOW TO $\overline{\text{RAS}}$ LOW	20	
2. $\overline{\text{RAS}}$ LOW TO $\overline{\text{CAS}}$ HIGH	20	
3. PULSE DURATION $\overline{\text{RAS}}$ LOW	180	10000
4. $\overline{\text{TR}}$ SETUP TIME	20	
5. $\overline{\text{TR}}$ HOLD TIME	0	

Figure 2-80. Refresh Timing of VRAM

The shift register is loaded 512 times during each vertical sweep of the electron beam since 512 rows of pixels must be displayed. As shown in Figures 2-77 and 2-79, the memory controller initiates a shift register load cycle in response to the signal XFER. The display logic generates this signal once every horizontal sweep during the retrace time, the time interval in which the electron beam returns from the right side of the display back to the left side.

As described above, the signal HSYNC controls the horizontal sweep of the electron beam and the signal VSYNC controls the vertical sweep of the electron beam. The frequency of HSYNC pulses is called the horizontal scan rate and the frequency of VSYNC pulses is called the vertical scan rate.

In our monitor, the horizontal scan rate is 31.5 kHz. As shown in Figure 2-87, the period of time between HSYNC pulses is 31.746 microseconds. Of this time, 27.456 microseconds are devoted to the display of 512 pixels, which gives us a pixel clock frequency of 18.648 MHz. During the active horizontal display time, designated by the signal HDISPEN in Figure 2-87, the display logic gates the pixel clock to create the clock called SERCLK. This clock is used to clock the

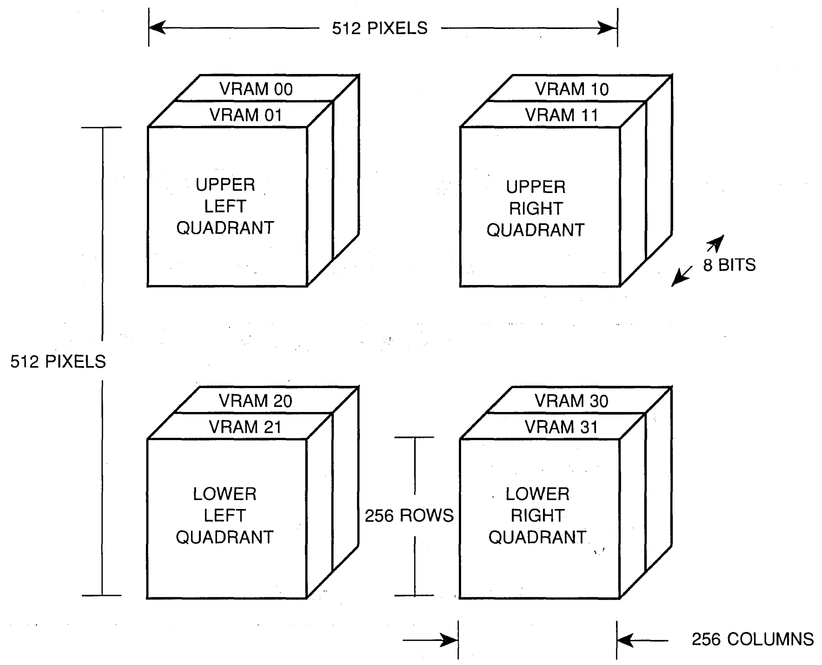


Figure 2-81. Physical Relationship of VRAMs to Display

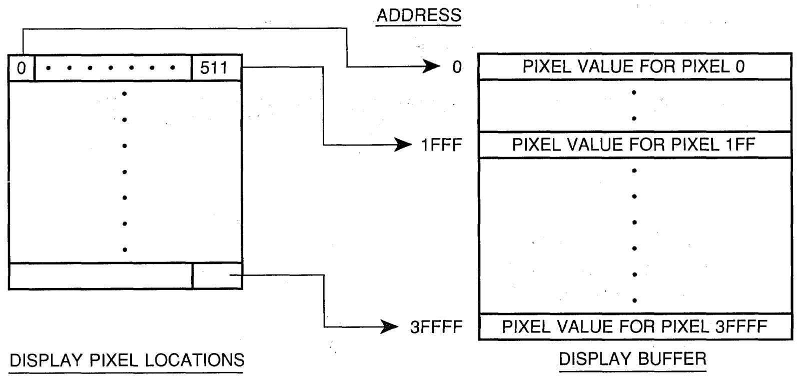


Figure 2-82. Organization of Display Buffer

shift register as shown in Figure 2-86.

Figure 2-88 presents a timing diagram for the vertical scan. As shown, 512 rows of pixels are displayed during the period of time labelled VDISPEN.

The timing diagram in Figure 2-89 highlights the major timing requirements of the display logic. The top of the diagram shows the relationship between HSYNC and VSYNC during one vertical scan period. This interval of time is further broken down into four major periods labelled A, B, C and D, which is shown in greater detail in the same diagram. Using the Am29PL142-based circuit in Figure 2-86, our task now is to write microprograms to synthesize each of these periods.

During time Period C, all 512 rows of pixels are displayed (see Figures 2-89 through 2-91). In essence, the activity that takes place during the cycle labelled HSDCYC is repeated 512 times. Therefore, our first step in writing a microprogram to implement the timing for Period C is to write a subprogram to implement the timing of HSDCYC.

Figure 2-91 provides detailed timing information for

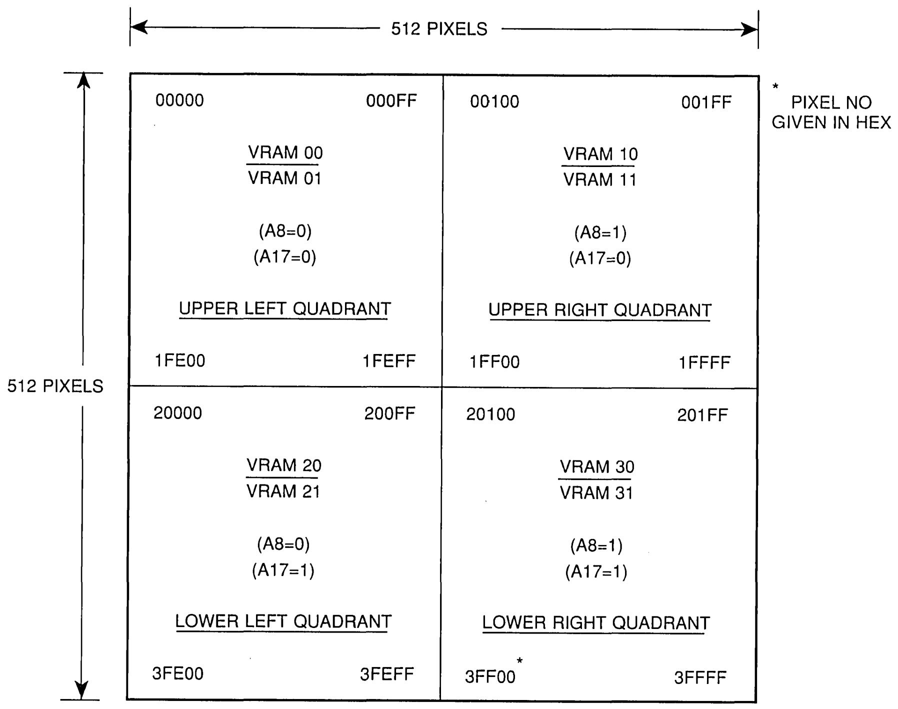


Figure 2-83. Relationship of VRAMs to Display Organization

```
                    ADDRESS BITS
17 16 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0

0  X  X  X  X  X  X  X  X  0  X  X  X  X  X  X  X  X     UPPER LEFT QUADRANT
0  X  X  X  X  X  X  X  X  1  X  X  X  X  X  X  X  X     UPPER RIGHT QUADRANT
1  X  X  X  X  X  X  X  X  0  X  X  X  X  X  X  X  X     LOWER LEFT QUADRANT
1  X  X  X  X  X  X  X  X  1  X  X  X  X  X  X  X  X     LOWER RIGHT QUADRANT

\__________V__________/   \__________V__________/
      ROW ADDRESS              COLUMN ADDRESS
```

Figure 2-84. Addressing of VRAMs Versus Pixel Locations

the HSDCYC cycle. This cycle is broken down into five segments—labelled C0 to C4—during which signal values remain steady. The length of each segment is defined by the number of pixel clocks that elapse during each segment. Recall from Figure 2-86 that the pixel clock also clocks the Am29PL142. Therefore, the number of pixel clocks specifies the number of microinstruction cycles necessary to implement a segment.

The microprogram in Figure 2-92 implements the HSDCYC cycle using the OUTPUT FIELD values derived in Figure 2-93. Each segment is implemented with the DECPL microinstruction, which has the advantage of loading the CREG with a new value during the last cycle of its execution. Notice, also, that segment C3, which requires 512 clocks, is implemented by 4 DECPL microinstructions, each contributing 128 clock cycles. Having implemented the HSDCYC cycle, we must repeat this program 512 times to implement Period C fully.

Figure 2-94 presents a microprogram that accomplishes this goal. The essential feature of this microprogram is that microprogram C0 is now a microsubroutine that is called within a loop that executes 128

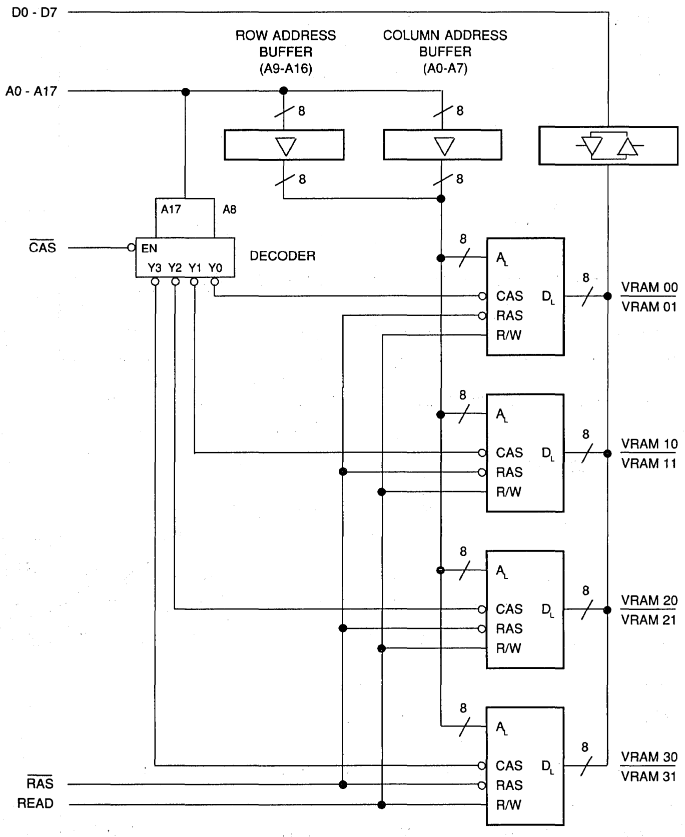


Figure 2-85. Address Decoding of VRAMs

times (see Statements 1–5). In turn, this loop is repeated an additional three times (see Statements 6–20) so that HSDCYC is executed 512 times.

Statement 5, the LOOP PL microinstruction, implements the loop in conjunction with the counter, which holds the count of the number of iterations left in executing the loop. However, microprogram C0 also uses the counter. As a result, the contents of the counter must first be stored on the stack before a call to microprogram C0 is executed. Then, upon return from microprogram C0, the counter is restored from the stack to allow the LOOP PL microinstruction to execute properly. Figure 2-95 reviews the operation of Statements 2–4 as an aide in understanding the Am29PL142 stack mechanism.

The HSDCYC microprogram is primarily responsible for implementing the segments in Figure 2-91. However, the execution of Statements 1–5 in Figure 2-94 also adds cycles that influence the length of segments C0 and C4. For our purposes, therefore, we assume that while Statement 1 initializes the loop counter, its OUTPUT FIELD implements the last cycle of Period B and does not affect the timing synthesized by our current microprogram.

However, the execution of Statements 2 and 3 implements the first two cycles of segment C0, and the execution of Statements 4 and 5 implements the last two cycles of segment C4. For this reason, we must reduce the counter values specified in the statements labelled C0 and LASTC3 in the HSDCYC microprogram (see Figure 2-96).

Let's take a closer look at the transition between Statements 5 and 6 in Figure 2-94. Statement 6 initializes the loop counter for executing the HSDCYC microprogram an additional 128 times. However, it executes only once since it is not part of the loop itself. As a consequence, the first HSDCYC cycle generated by the loop is longer by one clock cycle than the remaining 127 cycles. Clearly, this is not acceptable.

The microprogram in Figure 2-97 corrects this problem and properly generates 512 HSDCYC cycles. Like its predecessor, it consists of four loops, each loop generating 128 cycles of HSDCYC. But now the loop is implemented using the DEC and GOTOPL microinstructions rather than the LOOPPL microinstruction.

Upon entry to the microprogram, the counter is zero. Consequently, the first time Statement 1 executes, the counter is decremented to count 127, which is the desired loop count. When Statement 5 executes, the

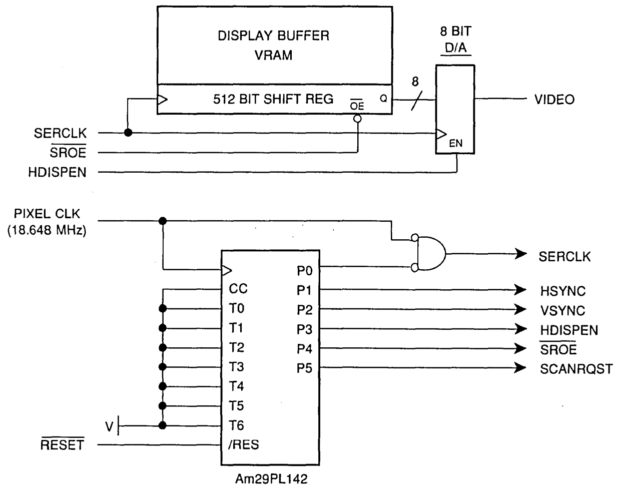


Figure 2-86. High-Level Diagram of Display Logic

state of the counter is tested. If the counter is not zero, control passes back to Statement 1, which decrements the counter again. Finally, after 128 repetitions, the counter counts down to zero, and when Statement 5 executes, control passes to Statement 6.

Executing Statements 6–10 duplicates the actions of Statements 1–5. But now, the transition from Statement 5 to Statement 6 is not problematic because Statement 6 is part of the loop itself and executes repeatedly.

Notice, also, that the OUTPUT FIELD of Statement 1 generates the first clock cycle of segment C0. Consequently, Statement C0 in the HSDCYC microprogram (see Figure 2-96) must be modified again to take this into account.

In writing the HSDCYC microsubroutine, we assumed we had a 512-bit shift register that was loaded once each horizontal scan line with a new set of pixel values (see Figure 2-86). The actual implementation of the display buffer is different, however, because of the physical structure of the VRAM. As shown in Figure 2-76, each VRAM has a single 256-bit shift register. Furthermore, the data in the shift register can only be shifted out so that we cannot simply connect 2 VRAMs together to form one 512-bit shift register.

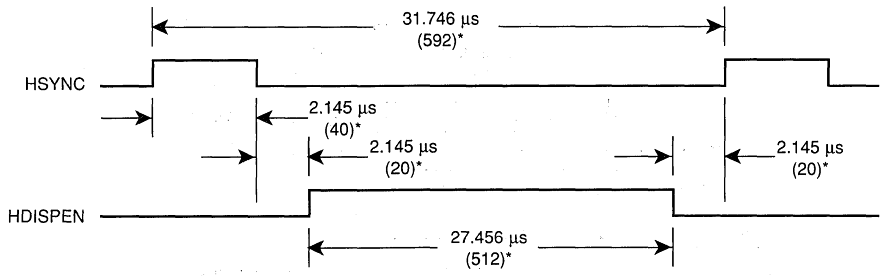
Figure 2-98 shows the organization of the VRAMs to implement the display buffer properly (see also Figures 2-83 and 2-85). As shown, we have four groups of VRAMs, each group corresponding to a quadrant of the display.

To display a row in the upper half of the screen, we must first enable the shift register output of VRAM00 and VRAM01 and then create a burst of 256 clock pulses to shift the pixel data out of the shift register. Then, to display the right half of the row, we must enable the shift register output of VRAM10 and VRAM11 and create another burst of 256 clock pulses to shift out the rest of the pixels.

The procedure for displaying a row of pixels in the lower half of the display is similar to that of the upper half. The only difference is which shift-register outputs are enabled (see Figure 2-99).

We can implement this new timing by adding more control bits to the Am29PL142 and modifying the HSDCYC microsubroutine (see Figure 2-100). However, before returning to HSDCYC, we need to discuss one additional aspect of the display buffer implementation.

As noted above, all 256 rows of each VRAM must be refreshed at least once every 2 milliseconds. This means that once every 15.6 microseconds the memory



HORIZONTAL SCAN RATE = f HSYNC = 31.5 kHz

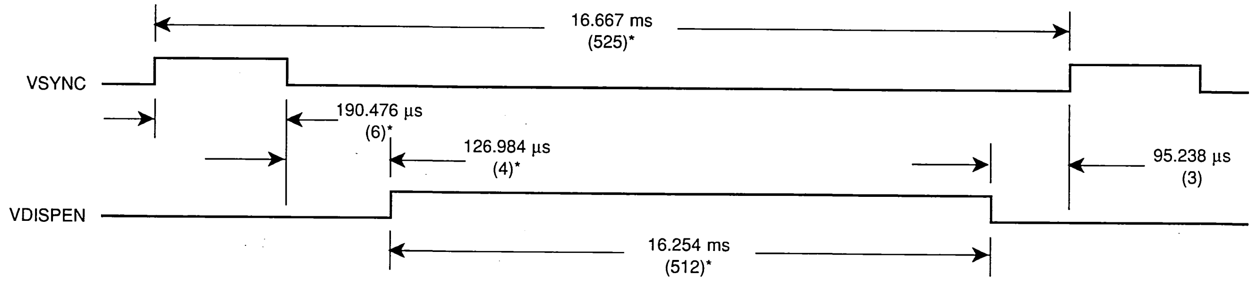
HORIZONTAL SCAN PERIOD = $1/f$HSYNC = 31.746 µs

PIXEL CLOCK PERIOD $= \frac{1/f\text{HSYNC}}{592} = \frac{31.746\ \mu s}{592} = 53.625$ ns

PIXEL CLOCK FREQ = 1/53.625 ns = 18.648 MHz

* THIS IS NUMBER OF PIXEL CLOCK PERIODS

Figure 2-87. Horizontal Scan Line Timing During Active Display



VERTICAL SCAN RATE = fVSYNC = 60 Hz

VERTICAL SCAN PERIOD = 1/fVSYNC = 16.667 ms

* THIS IS NUMBER OF HORIZONTAL SCAN PERIODS (HSYNC PULSES)

Figure 2-88. Vertical Scan Timing

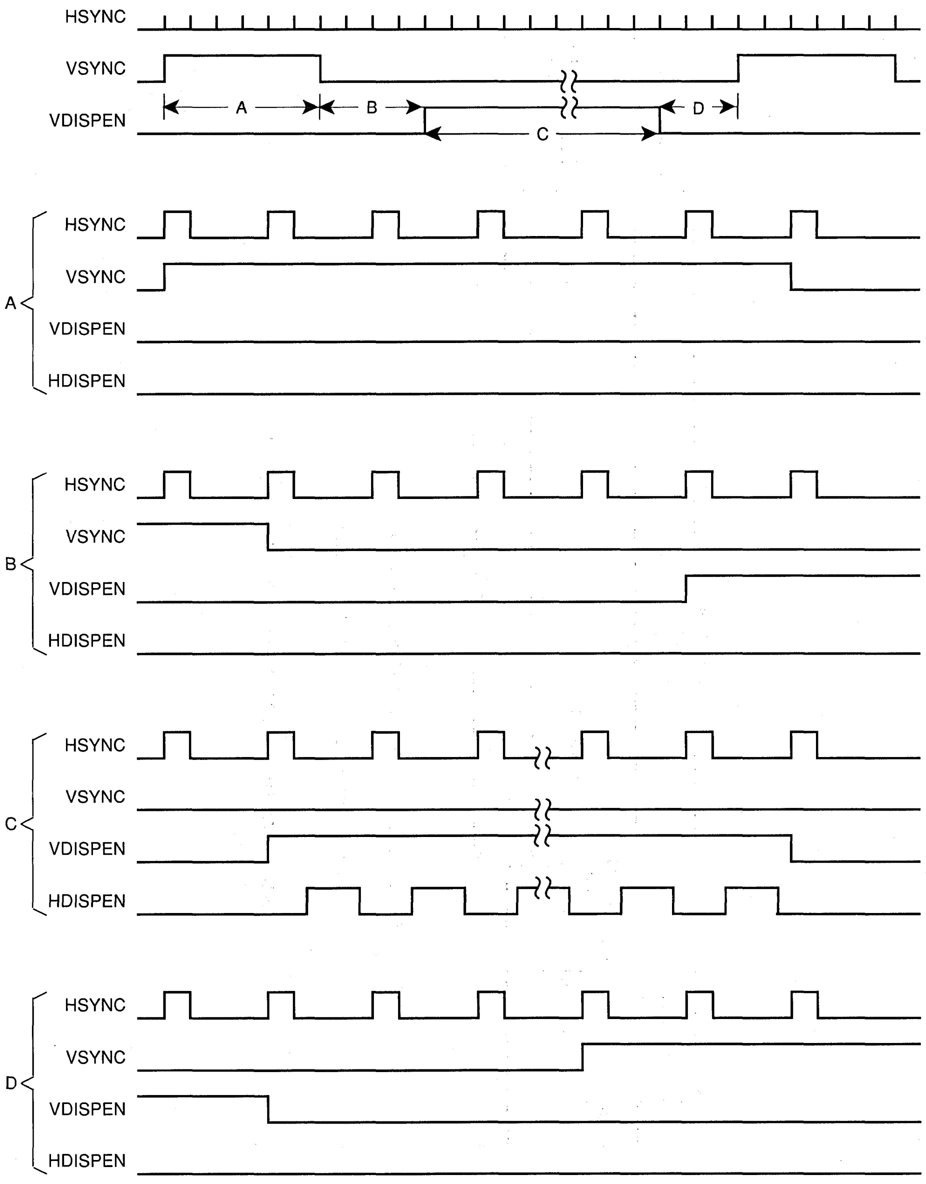


Figure 2-89. Timing Diagram of Display Logic

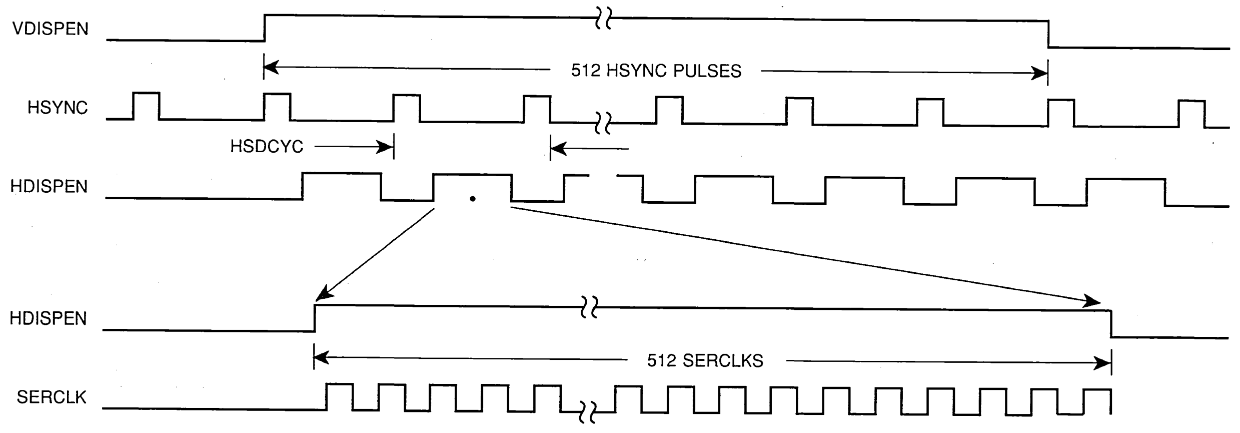
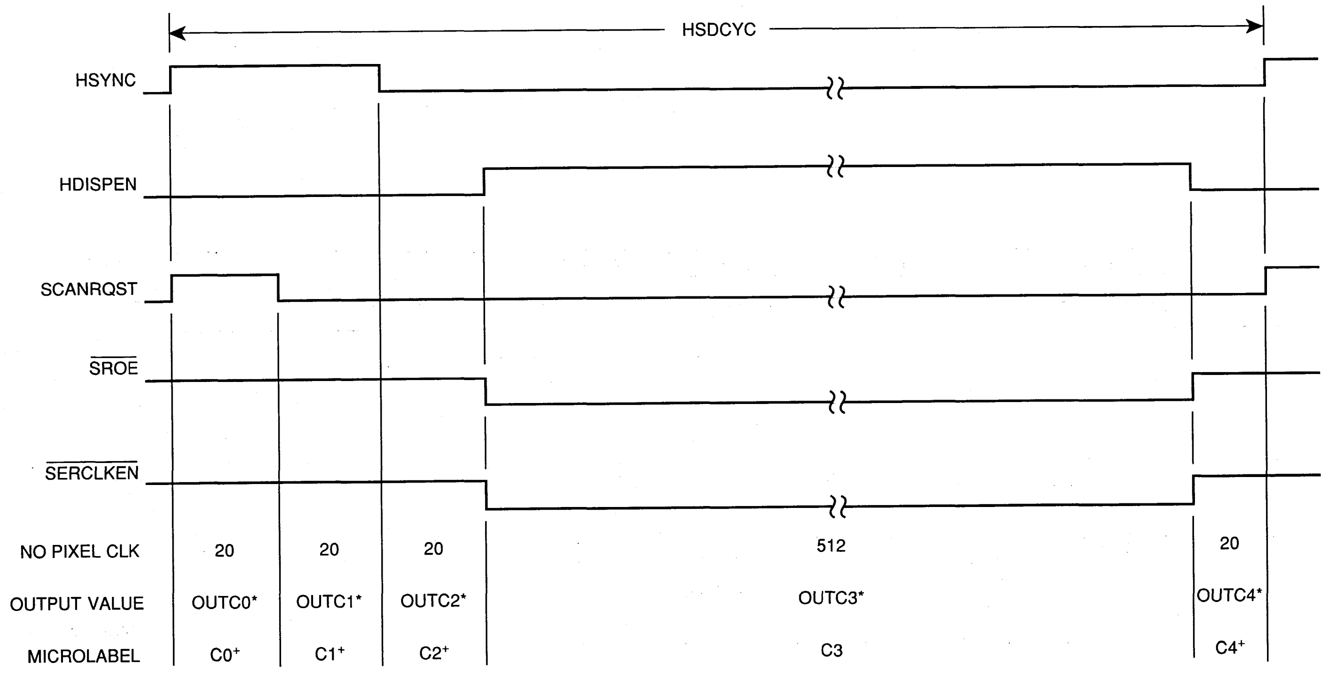


Figure 2-90. Expanded Timing Diagram for VDISPEN



* SEE FIGURE 2-93
+ SEE FIGURE 2-92

Figure 2-91. Timing Diagram for Horizontal Sync Display Cycle (HSDCYC)

controller must initiate a refresh operation. Since the display logic is already generating the timing for control of the display buffer and CRT, we can once again modify the HSDCYC microprogram to generate a refresh request signal (REFRRQST) during a scan line.

Figure 2-101 presents a timing diagram that shows the relationship between REFRRQST and the other control signals generated by HSDCYC. Notice that REFRRQST is generated four times per scan line (once every 6.864 microseconds), which is a faster rate than necessary. We do this to allow enough time for the memory controller to respond to this signal as it is arbitrating among other requests.

Let's return to the microprogram for implementing Period C. As noted above, the output enable signals for the shift registers differ depending upon whether the displayed pixels are in the upper or the lower half of the display (see Figure 2-99). At the same time, however, the other control signals remain the same. For this reason, we need two microsubroutines, labeled UPPER and LOWER.

Figure 2-102 presents a portion of the microprogram for Period C. Like its predecessor (see Figure 2-97), this microprogram also is structured as a series of four loops, each loop generating 128 HSDCYC cycles. However, in this microprogram, the first two loops execute a microsubroutine call to UPPER, and the last two loops execute a microsubroutine call to LOWER. Notice, also, that the values of the OUTPUT FIELD are different, reflecting the additional control signals that have been added (see Figure 2-103).

The microprogram for UPPER is presented in Figure 2-104. Notice that segment C3 now is broken down into eight subsegments as defined in Figure 2-101. The microprogram for LOWER is similar and differs only in the OUTPUT FIELD values that control the shift-register outputs.

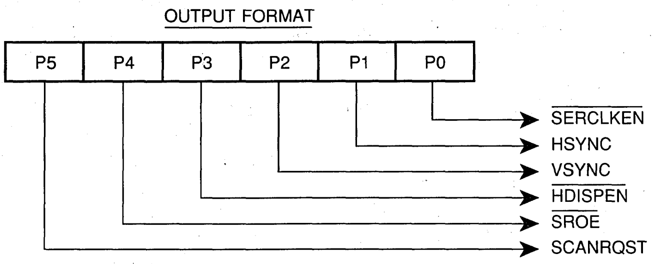
We now have completely implemented Period C. Let's continue and write microprograms to synthesize the timing for Periods A, B and D.

As shown in Figure 2-89, Periods 1, B and D are similar, differing only in their duration and VSYNC assertion. As a result, we have two fundamental cycles, called HSCYCA and HSCYCBD, which are repeated in a manner similar to HSDCYC described above (see Figures 2-105 and 2-106).

The microprograms to synthesize HSCYCA and HSCBD are shown in Figure 2-107 and are refer-

```
DEVICE (PL142)

DEFINE          FAIL  =  T0
               OUTC0  =  33 #H
               OUTC1  =  13 #H
               OUTC2  =  11 #H
               OUTC3  =  08 #H
               OUTC4  =  11 #H;

BEGIN

          C0:    OUTC0, IF (NOT FAIL) THEN LOAD PL (18 #D);
                                " LOAD COUNTER WITH INITIAL COUNT "
                 OUTC0, WHILE (CREG < > Ø) WAIT ELSE LOAD PL (19 #D);
                                " GENERATE WAVEFORM FOR C0 "
          C1:    OUTC1, WHILE (CREG < > Ø) WAIT ELSE LOAD PL (19 #D);
                                " GENERATE WAVEFORM FOR C1 "
          C2:    OUTC2, WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
                                " GENERATE WAVEFORM FOR C2 "
          C3:    OUTC3, WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
                 OUTC3, WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
                 OUTC3, WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
     LAST C3:    OUTC3, WHILE (CREG < > Ø) WAIT ELSE LOAD PL (19 #D);
                                " GENERATE WAVEFORM FOR C3 "
          C4:    OUTC4, WHILE (CREG < > Ø) WAIT ELSE LOAD PL (0 #H);
                                  " GENERATE WAVE FOR C4 "
```

Figure 2-92. Microprogram to Synthesize HSDCYC Cycle



	OUTPUT VALUES						HEX EQUIVALENT
OUTC0	1	1	0	0	1	1	33
OUTC1	0	1	0	0	1	1	13
OUTC2	0	1	0	0	0	1	11
OUTC3	0	0	1	0	0	0	08
OUTC4	0	1	0	0	0	1	11

Figure 2-93. Output Values for Each Segment of HSDCYC

```
STATEMENT NO.  LABEL       MICROINSTRUCTION                    COMMENTS
     1           C:        OUTB, LOAD PL (127 #H) ;        " EXECUTE LAST CYCLE OF PERIOD "
                                                           " B AND SETUP LOOP COUNT "
     2         REPEAT:     OUTC0, PUSHCNTR     ;           " SAVE LOOP COUNT ON STACK "
     3                     OUTC0, CALL PL (C0)+  ;         " EXECUTE HSDCYC ONCE "
     4                     OUTC4, POPCNTR      ;           " RESTORE COUNTER WITH LOOP COUNT;
     5                     OUTC4, WHILE (CREG < > Ø) LOOP TO PL (REPEAT);
                                                           " REPEAT INSTRUCTIONS UNTIL COUNT = ZERO "
     6                     OUTC0, LOAD PL (127 #H);                    ┐
     7         REPEAT1:    OUTC0, PUSHCNTR     ;                       │
     8                     OUTC0, CALL PL (C0)+  ;                     │
     9                     OUTC4, POPCNTR      ;                       │
    10                     OUTC4, WHILE (CREG < > Ø) LOOP TO PL  (REPEAT1);
     •                     •                                           │   EXECUTE
     •                     •                                           >  HSDCYC
     •                     •                                           │  484 TIMES
    16                     OUTC0, LOAD PL (127 #H);                    │
    17         REPEAT3:    OUTC0, PUSHCNTR     ;                       │
    18                     OUTC0, CALL PL (C0)   ;+                    │
    19                     OUTC4, POPCNTR      ;                       │
    20                     OUTC4, WHILE (CREG < > Ø) LOOP TO PL (REPEAT 3), ┘
```

+ SEE FIGURE 2-92.

Figure 2-94. Microprogram for Repeating HSDCYC 512 Times

STATEMENT NO	MICROINSTRUCTION
1	PUSHCNTR
2	CALL PL (EX)
3	POPCNTR
4	EX: RETURN

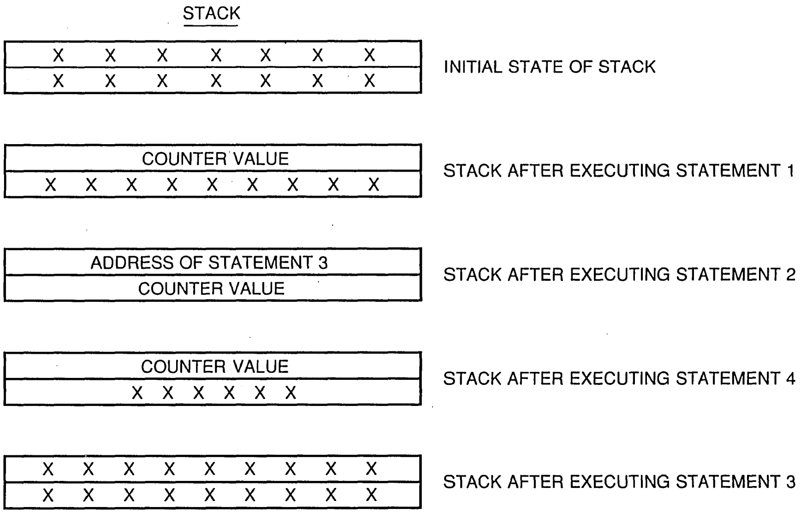


Figure 2-95. Operation of Stack

```
     C0:  OUTC0,  IF (NOT FAIL) THEN LOAD PL (16 #D);
                  " REDUCE COUNT TO 16 TO COMPENSATE "
                  " FOR CYCLES IN STATEMENTS 2 AND 3 IN FIGURE 76 "
          OUTC0,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (19 #D);
     C1:  OUTC1,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (19 #D);
     C2:  OUTC2,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
     C3:  OUTC3,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
          OUTC3,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
          OUTC3,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
 LASTC3:  OUTC3,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL  (16 #D);
                  " REDUCE COUNT TO 16 TO COMPENSATE FOR "
                  " CYCLES  IN STATEMENTS 4 AND 5 IN FIGURE 76 "
                  " AND RETURN MICROINSTRUCTION BELOW "
     C4:  OUTC4,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (0 #H);
          OUTC4, IF (NOT FAIL) RETURN;
          " ADD RETURN TO CONTINUE EXECUTION AT "
          " STATEMENT 4 IN FIGURE 2-94 "
```

Figure 2-96. Modification of HSDCYC Microprogram in Figure 2-92

enced by these names as well (also see Figure 2-108).

As shown in Figure 2-109, the timing for Period A is synthesized by calling HSCYCA within a loop that executes 6 times (also see Figure 2-88). In a similar vein, the timing for Period B is synthesized by executing HSCYCBD four times, and for Period D, by calling HSCYCBD three times.

Let's look at the microprogram for Period A in more detail. The microinstruction in Statement 1 initializes the loop counter. Notice, however, that this microinstruction executes once, when the loop is first entered and the counter is zero. During subsequent iterations of the loop, Statement 1 contributes execution cycles to synthesize segment A0 but does not reload the counter.

The microprogram for Period A repeats until the counter counts down to zero.Then, when Statement 6 executes, control passes to Statement 8, labelled EXITA. The major purpose of Statement 8 is to ensure that during the last iteration of the loop, segment A1 is as long as it is for each of the other iterations. The microprograms for Periods B and D are similar to Period A and are not discussed in detail here.

2.2.5.3 Implementing the Memory Controller

The essential function of the memory controller is to arbitrate among requests from the CPU and display logic and then take appropriate action to service those requests.

Figure 2-110 shows a high-level block diagram of the display buffer and memory controller. Most of the elements shown in this diagram are already known to us from previous discussions. However, the row counter, shown in the upper left-hand corner, is an entirely new element.

Recall that the display logic generates the signal SCANRQST during the horizontal sync period of HSDCYC. This signal is a request to the memory

```
                       COUNTER EQUAL TO ZERO UPON ENTRY

STATEMENT NO.  LABEL       MICROINSTRUCTION                  COMMENTS
     1          C:       OUTC0, DEC            ;     " THE FIRST EXECUTION OF THIS "
                                                     " MICROINSTRUCTION MAKES THE "
                                                     " COUNTER EQUAL TO 127 "
     2                   OUTC0, PUSHCNTR       ;     " SAVE LOOP COUNTER "
     3                   OUTC0, CALL PL (C0)   ;     " EXECUTE MICROPROGRAM FOR "
                                                     " HSDCYC IN FIGURE 78 "
     4                   OUTC4, POPCNTR        ;     " RESTORE LOOP COUNTER "
     5                   OUTC4, IF (CREG < > Ø) THEN GOTO PL (C);
                                                     " THIS MICROINSTRUCTION "
                                                     " IMPLEMENTS LOOP "
     6       REPEAT 1 :  OUTC0, DEC            ;                            ⎫
     7                   OUTC0, PUSHCNTR       ;                            ⎪
     8                   OUTC0, CALL PL (C0)   ;                            ⎪
     9                   OUTC4, POPCNTR        ;                            ⎪
    10                   OUTC4, IF (CREG < > Ø) THEN GOTO PL  (REPEAT1);    ⎪
                         •                                                  ⎪  EXECUTE
                         •                                                  ⎬  HSDCYC
                         •                                                  ⎪  484 TIMES
    16       REPEAT 3 :  •                                                  ⎪
    17                   •                                                  ⎪
    18                   •                                                  ⎪
    19                   •                                                  ⎪
    20                   •                                                  ⎭
```

Figure 2-97. Improved Microprogram to Implement Period C

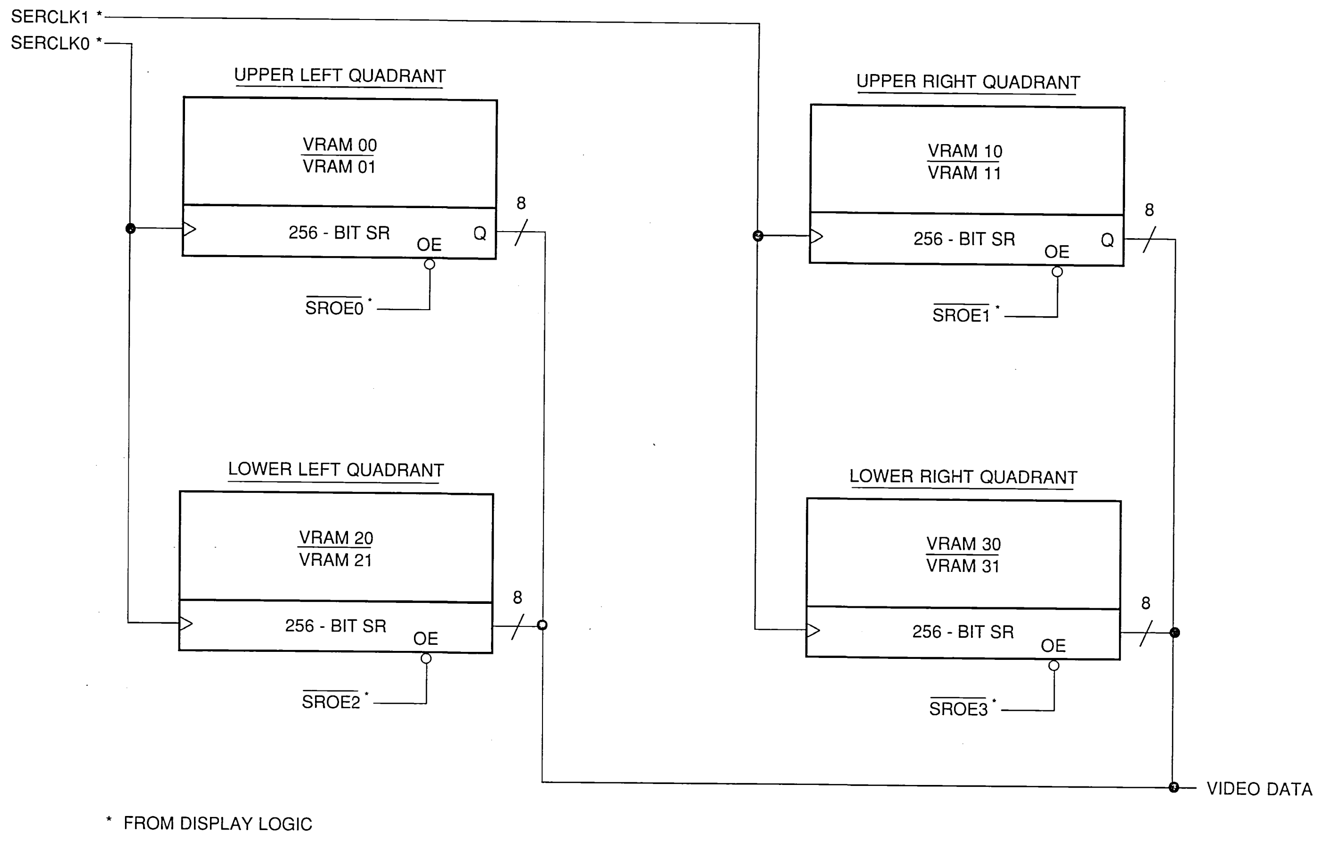


Figure 2-98. Organization of VRAM Shift Registers

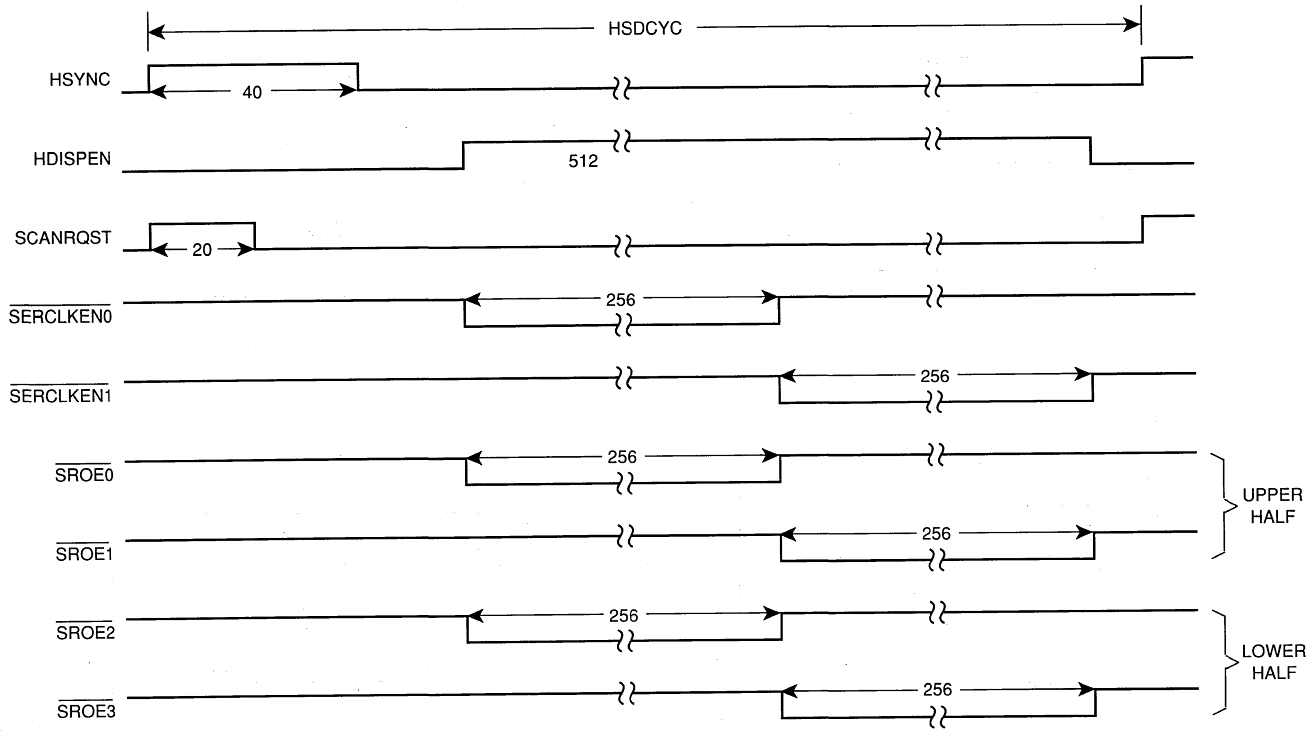

HSDCYC
HSYNC
40
HDISPEN
512
SCANRQST
20
$\overline{\text{SERCLKEN0}}$
256
$\overline{\text{SERCLKEN1}}$
256
$\overline{\text{SROE0}}$
256
$\overline{\text{SROE1}}$
256
UPPER HALF
$\overline{\text{SROE2}}$
256
$\overline{\text{SROE3}}$
256
LOWER HALF

Figure 2-99. Timing of Shift Register Output Enable During HSDCYC

controller to load the VRAM shift register with a new row of pixels. As shown in Figure 2-79, however, we need to specify a row address. This function is served by the row counter, which is used during the shift register load cycle.

The address bit labelled RAC8, also shown in Figure 2-110, is the top most address bit of the row address counter. When this bit is low, the upper half of the display is active, and when it is high, the lower half of the display is active. The memory controller uses this bit to determine which VRAMs to select during the shift register load cycle (see Figure 2-111).

The input signals to the memory controller fall into two groups. Four of the signals—RD, WR, SCANRQST, REFRRQST—are the request signals among which the memory controller arbitrates. The remaining signals are auxiliary signals used by the memory controller microprograms to implement the actions associated with each of the request signals.

In Example 2, we designed a memory controller that synthesized the timing for reading data from and writing data to the display buffer. In our current example, we make use of Example 2 microsubroutines since they perform the same functions we need now.

Recall, however, that in Example 2, we scanned the request lines to decide which request to honor. In our current example, we have four request lines. In scanning all four of these lines, we waste a considerable amount of time. A better procedure is to use the GOTOTM microinstruction in conjunction with a jump table, just as we did in Examples 3 and 4.

Aside from requesting different services, there is a major difference between the behavior of CPU and display logic request signals. As we discussed in Example 2, the memory controller generates the signal OPCOMP to interlock the memory controller to the CPU. In contrast, the display controller is not locked to the memory controller when generating the requests SCANRQST or REFRRQST.

To see the consequence of this, let's examine the events that take place when the display logic generates SCANRQST and no other requests are active. As shown in Figure 2-112, the memory controller uses the GOTOTM microinstruction and a jump table to establish the priorities among the incoming request signals. Now, assuming SCANRQST is active, the memory controller executes the SCAN microprogram and then returns to the idle loop. The problem is that SCANRQST is still active. As a result, the SCAN microprogram executes again and again until the display controller negates SCANRQST.

Figure 2-113 presents a solution to the above problem. In this state diagram, State 0 refers to the arbitration process that takes place using the GOTOTM microinstruction and jump table described above. Now, when SCANRQST is active, a transition takes place to State 1. In State 1, the SCAN microprogram executes and a new row of pixels is loaded into the VRAM shift register. When the SCAN microprogram completes its execution, it jumps to yet another state,

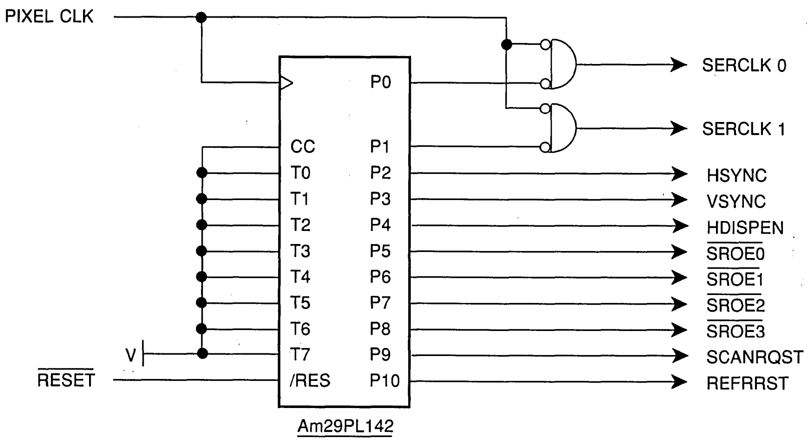


Figure 2-100. Adding Control Signals for HSDCYC

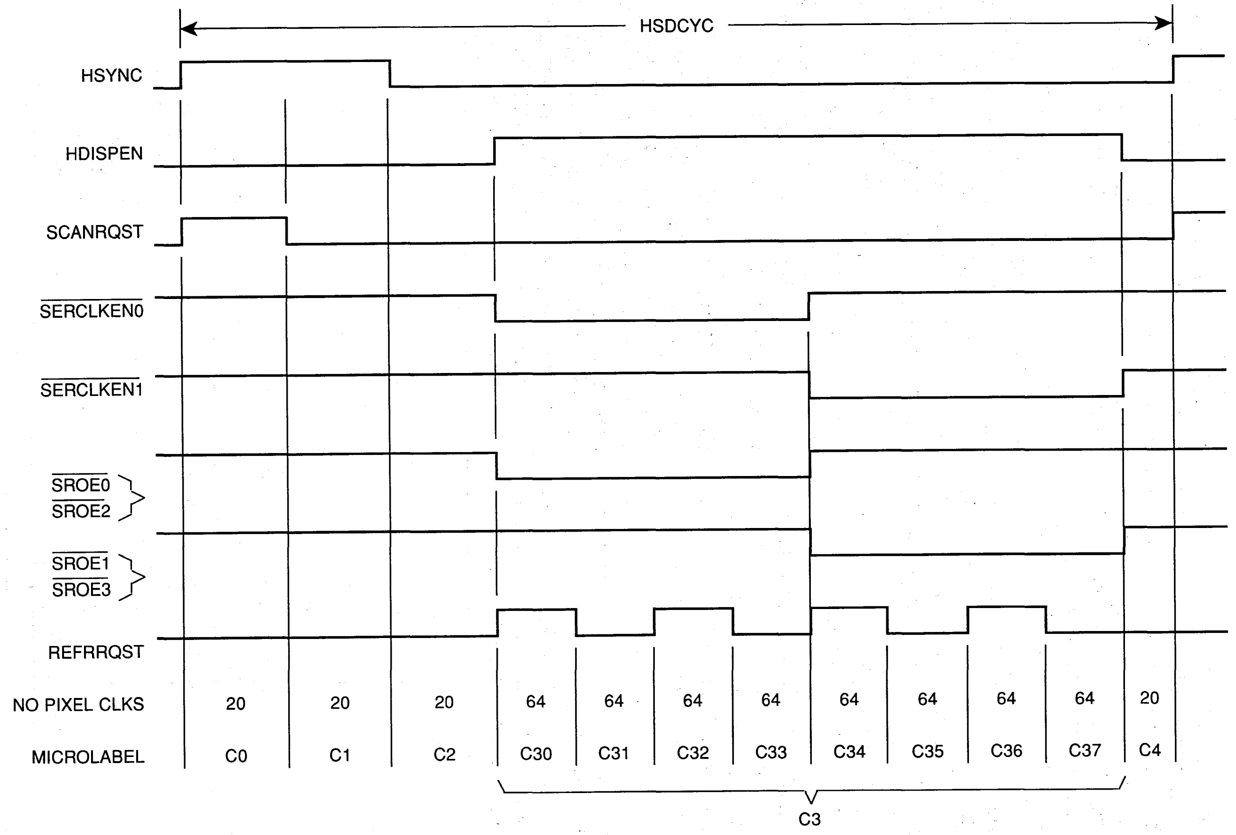

HSDCYC
HSYNC
HDISPEN
SCANRQST
$\overline{\text{SERCLKEN0}}$
$\overline{\text{SERCLKEN1}}$
$\overline{\text{SROE0}}$ $\overline{\text{SROE2}}$
$\overline{\text{SROE1}}$ $\overline{\text{SROE3}}$
REFRRQST
NO PIXEL CLKS
MICROLABEL
20 20 20 64 64 64 64 64 64 64 64 20
C0 C1 C2 C30 C31 C32 C33 C34 C35 C36 C37 C4
C3

Figure 2-101. Composite Timing Diagram for Final HSDCYC Timing Diagram

State 2, to continue arbitrating among the CPU request signals. In State 2, in contrast to State 0, the continued active level of SCANRQST does not result in another execution of the SCAN microprogram.

State 2 is implemented in the same manner as State 0, that is, the GOTOTM microinstruction is used in conjunction with a jump table (see Figure 2-114). It is the composition of the jump table, though, that allows us to ignore selectively the active level of SCANRQST. However, when SCANRQST goes low again, a transition takes place back to State 0.

Notice, also, that in State 2, the state of REFRRQST is ignored. The reason for this is that SCANRQST and REFRRQST are mutually exclusive just as RD and WR. As a result, SCANRQST goes low and the memory controller enters State 0 again before REFRRQST is asserted.

As mentioned above, some of the input signals to the memory controller are auxiliary signals. For example, in Figure 2-85 we used a decoder to route $\overline{CAS}$ to the appropriate VRAMs depending on the state of address bits A8 and A17. However, the Am29PL142 can perform this function as well. Consequently, if the CPU asserts the signal RD, the microprogram READ executes to implement the desired actions. We can modify the microprogram READ, however, to examine the state of A8 and A17 and then assert the appropriate $\overline{CAS}$ line.

The auxiliary signal RAC8, described above, is used in a similar fashion. When the display controller asserts SCANRQST and the memory controller honors this request, the SCAN microprogram executes. The SCAN microprogram then can examine the state of RAC8 and again activate the appropriate $\overline{CAS}$ lines to select the right VRAMs.

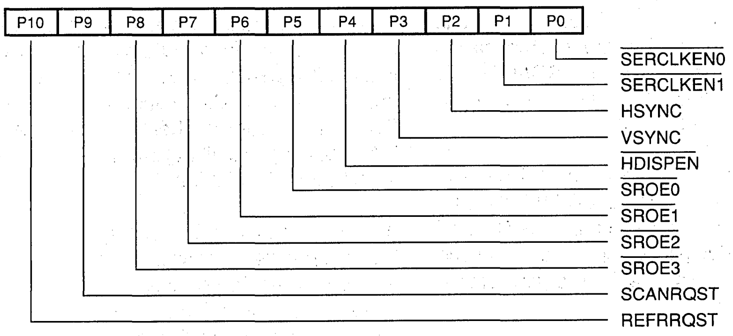
As shown in Figure 2-110, output P13 is fed back to TEST input T4. The reason for this is to allow the memory controller to select either the State 0 jump table or the State 2 jump table. Recall that we used this technique in Example 4 when we needed to select one of four priority tables for the VME bus arbiter.

This completes our description of the operation of the memory controller. Since the memory controller

```
                LABEL                         MICROINSTRUCTION
 " DISPLAY 1ST 128 ROWS "
                  C:              OUTC00, DEC;
                                  OUTC00, PUSHCNTR;
                                  OUTC00, CALL PL (UPPER);
                                  OUTC40, POPCNTR;
                                  OUTC40, IF (CREG < > Ø) THEN GOTO PL (C);

 " DISPLAY 2ND 128 ROWS "
            REPEAT 1:             OUTC00, DEC;
                                  OUTC00, PUSHCNTR;
                                  OUTC00, CALL PL (UPPER);
                                  OUTC40, POPCNTR;
                                  OUTC40, IF (CREG < > Ø) THEN GOTO PL (REPEAT 1);

 " DISPLAY 3RD 128 ROWS "
            REPEAT 2:             OUTC01, DEC;
                                  OUTC01, PUSHCNTR;
                                  OUTC01, CALL PL (LOWER);
                                  OUTC41, POPCNTR;
                                  OUTC41, IF (CREG < > Ø) THEN GOTO PL (REPEAT 2);

 " DISPLAY 4TH 128 ROWS "
            REPEAT 3:             OUTC01, DEC;
                                  OUTC01, PUSHCNTR;
                                  OUTC01, CALL PL (LOWER);
                                  OUTC41, POPCNTR;
                                  OUTC41, IF (CREG < > Ø) THEN GOTO PL (REPEAT 3);
```

Figure 2-102. Final Microprogram for Period C

OUTPUT FORMAT



											HEX EQUIV	NAME FOR OUTPUT FIELD VALUE
0	1	1	1	1	1	0	0	1	1	1	3E7	OUTC00
0	0	1	1	1	1	0	0	1	1	1	1E7	OUTC10
0	0	1	1	1	1	0	0	0	1	1	1E3	OUTC20
1	0	1	1	1	0	1	0	0	1	0	5D2	OUTC300
0	0	1	1	1	0	1	0	0	1	0	1D2	OUTC310
1	0	1	1	1	0	1	0	0	1	0	5D2	OUTC320
0	0	1	1	1	0	1	0	0	1	0	1D2	OUTC330
1	0	1	1	0	1	1	0	0	0	1	5D1	OUTC340
0	0	1	1	0	1	1	0	0	0	1	1D1	OUTC350
1	0	1	1	0	1	1	0	0	0	1	5D1	OUTC360
0	0	1	1	0	1	1	0	0	0	1	1D1	OUTC370
0	0	1	1	1	1	0	0	0	1	1	1E3	OUTC40
0	1	1	1	1	1	0	0	1	1	1	3E7	OUTC01
0	0	1	1	1	1	0	0	1	1	1	1E7	OUTC11
0	0	1	1	1	1	0	0	0	1	1	1E3	OUTC21
1	0	1	0	1	1	1	0	0	0	1	571	OUTC301
0	0	1	0	1	1	1	0	0	0	1	171	OUTC311
1	0	1	0	1	1	1	0	0	0	1	571	OUTC321
0	0	1	0	1	1	1	0	0	0	1	171	OUTC331
1	0	0	1	1	1	1	0	0	1	0	572	OUTC341
0	0	0	1	1	1	1	0	0	1	0	172	OUTC351
1	0	0	1	1	1	1	0	0	1	0	572	OUTC361
0	0	0	1	1	1	1	0	0	1	0	172	OUTC371
0	0	1	1	1	1	0	0	0	1	1	1E3	OUTC41

Figure 2-103. Output Values for Microprogram C and HSDCYC

incorporates many of the functions described in the previous examples, we do not present a detailed explanation of the implementation. However, Figure 2-115 presents the microprogram to implement the memory controller as a reference.

This example, the frame store, is an especially attractive example of the power of the Am29PL100 family. The particular advantage offered by each of the Am29PL142s is the ability to implement multiple functions in one chip. For the display logic, we implemented a full-scale CRT controller in addition to counters to generate refresh and scan requests. For the memory controller, we implemented both the random access and serial access modes of the VRAMs. An alternative implementation based upon discrete logic would use considerably more components.

2.3 CONTROL SIGNAL DESCRIPTIONS

This section contains the control signal descriptions for the control unit of the simple computer described in Section 2.1.1. The 11 control signals are described below:

LDIDR — This signal latches data from the memory into the input data register (IDR) on the rising edge.

LDODR — This signal latches data from the A port of the register file into the output data register (ODR) on the rising edge.

ENODR — This signal controls the three-state condition of the ODR. When ENODR is low, the ODR output is active, allowing data to be written to the memory (see description of READ signal). When the signal is high, the ODR output is three-stated, allowing the memory to be read.

LDMAR — This signal latches data into the memory address register (MAR) on the rising edge (see description of SELMAR below).

SELMAR — This signal selects the source of the address to be loaded into the MAR. When SELMAR is low, the incremented value of the program counter, R7, is selected. When the signal is high, the address contained in the IDR is selected.

READ — This signal controls the memory reads and writes. When READ is low, data from the ODR is

```
UPPER: OUTC00   ,  LOAD PL (16 #D);
       OUTC00   ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (19 #D);
C10:   OUTC10   ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (19 #D);
C20:   OUTC20   ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C300:  OUTC300  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C310:  OUTC310  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C320:  OUTC320  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C330:  OUTC330  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C340:  OUTC340  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C350:  OUTC350  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C360:  OUTC360  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C370:  OUTC370  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (16 #D);
C40:   OUTC40   ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (0 #H);
       OUTC40   ,  RETURN;

LOWER: OUTC01   ,  LOAD PL (16 #D);
       OUTC01   ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (19 #D);
C11:   OUTC11   ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (19 #D);
C21:   OUTC21   ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C301:  OUTC301  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C311:  OUTC311  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C321:  OUTC321  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C331:  OUTC331  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C341:  OUTC341  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C351:  OUTC351  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C361:  OUTC361  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (3F #H);
C371:  OUTC371  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (16 #D);
C41:   OUTC41   ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (0 #H);
       OUTC41   ,  RETURN;
```

Figure 2-104. Microprograms Upper and Lower

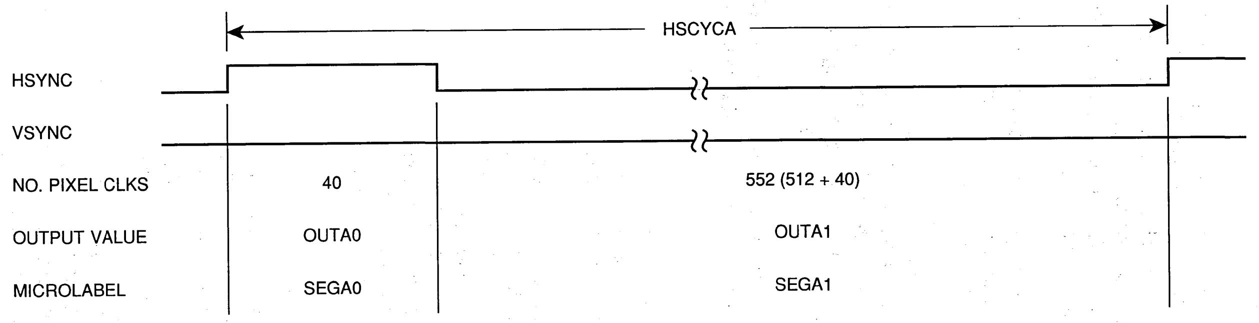


Figure 2-105. Timing for Horizontal Sync Cycle During Period A

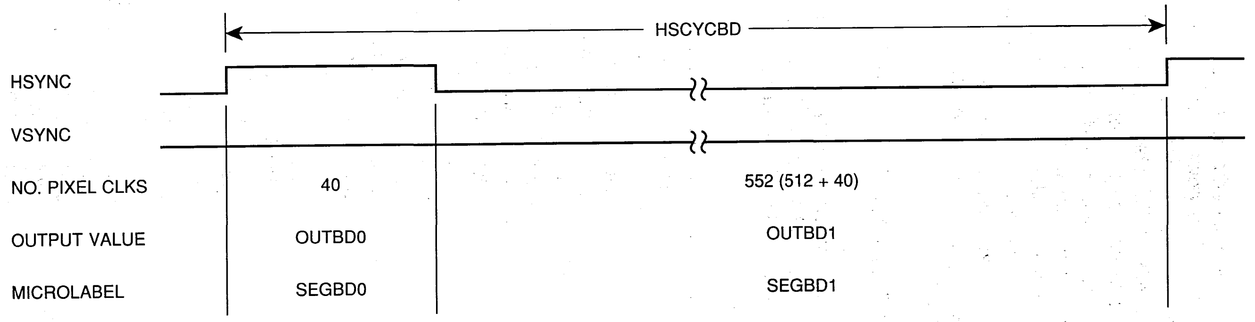


Figure 2-106. Timing for Horizontal Sync Cycle During Periods B and D

written to the memory at the location specified by the MAR. When the signal is high, data is read from the memory and latched into the IDR (see description of LDIDR above).

SELRFDATA — This signal selects the source of the data to be written into the register file. When SELRFDATA is low, data from the IDR is written into the register file. When the signal high, data from the ALU is written into the register file.

SELRFADDR — This signal selects the source of the Port A address of the register file. When SELRFADDR is low, the address comes from the IDR. When the signal is high, the address comes from the control unit (see description of PORTAADDR below).

PORTAADDR — These bits select the Port A address of the register file. When the signal low, the address comes from the IDR. When it is high, the address comes from the microcode (see description of SELRFADDR above).

$\overline{\text{WRRF}}$ — This signal writes data into the register file at the address specified in Port B.

ALUCODE — These bits select the ALU operation to be performed (see Table 2-1 in Section 2.1).

2.4 INSTRUCTION SET DESCRIPTIONS

This section contains the descriptions of the instruction set for the simple computer described in Section 2.1.

The 15 instructions are described below:

SHIFTL — Register R0 is shifted left 1–16 positions as specified in the COUNT field.

```
DEVICE  (PL142)
DEFINE

          OUTA0   = 1EF #H
          OUTA1   = 1EB #H
          OUTBD0 = 1E7 #H
          OUTBD1 = 1E3 #H;

BEGIN
                 " MICROPROGRAM FOR HSCYCA "
SEGA0:      OUTA0  ,  LOAD PL (36 #D);
                              " LOAD COUNTER WITH INITIAL COUNT "
            OUTA0  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
                              " GENERATE WAVEFORM FOR SEGA0 "

SEGA1:      OUTA1  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
            OUTA1  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
            OUTA1  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
            OUTA1  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (36 #D);
                              " GENERATE FIRST 512 CYCLES OF SEGA1 "
            OUTA1  ,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (0 #H);
            OUTA1  ,  RETURN;

                 " MICROPROGRAM FOR HSCYCBD "
SEGBD0:     OUTBD0,  LOAD PL (36 #D);
            OUTBD0,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
SEGBD1:     OUTBD1,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
            OUTBD1,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
            OUTBD1,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (7F #H);
            OUTBD1,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (36 #D);
            OUTBD1,  WHILE (CREG < > Ø) WAIT ELSE LOAD PL (0 #H);
            OUTBD1,  RETURN;
```

Figure 2-107. Microprograms to Synthesize HSCYCA and HSCYCBD Cycle

ADD — The content of the specified source register is added to the content of the destination register. The result is written back to the destination register.

SUB — The content of the destination register is subtracted from the content of the source field. The result is written back to the destination field.

AND — The content of the source register is ANDed with the content of the destination register. The result is written back to the destination register.

OR — The content of the source register is ORed with the content of the destination register. The result is written back to the destination register.

JMP — Transfer of control is passed to the address specified in the ADDRESS field.

CALL SUBROUTINE — Transfer of control is passed to the address specified in the ADDRESS field. The address of the next sequential instruction is stored on the stack for subsequent continuation of instruction processing.

RETURN — Transfer of control is passed to the return address contained on the stack. The stack is popped after the top of stack is used as return address.

BRANCHN — Transfer of control is passed to the address specified in the ADDRESS field if register R0 holds a negative two's complement number. Otherwise, processing continues with the next sequential instruction.

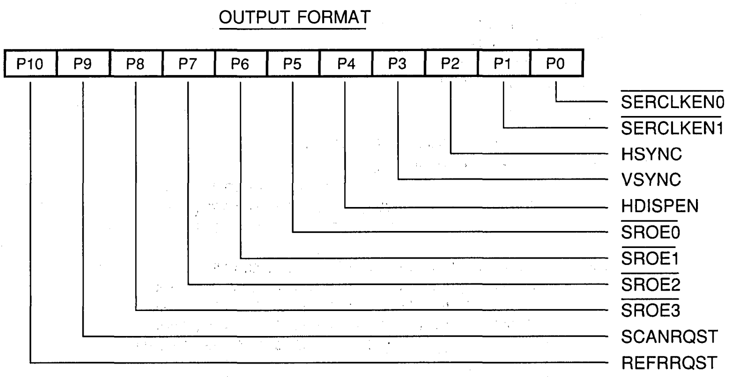
BRANCHZ — Transfer of control is passed to the address specified in the ADDRESS field if the content of register R0 is zero. Otherwise, processing continues with the next sequential instruction.

LOAD — Register R0 receives the data from the memory location specified in the ADDRESS field.

STORE — The content of register R0 is written to the memory location specified in the ADDRESS field.

INC — The content of the destination register is incremented by one.

DEC — The content of the destination register is decremented by one.


OUTPUT FORMAT
P10
P9
P8
P7
P6
P5
P4
P3
P2
P1
P0
$\overline{\text{SERCLKEN0}}$
$\overline{\text{SERCLKEN1}}$
HSYNC
VSYNC
HDISPEN
$\overline{\text{SROE0}}$
$\overline{\text{SROE1}}$
$\overline{\text{SROE2}}$
$\overline{\text{SROE3}}$
SCANRQST
REFRRQST

OUTPUT VALUES											HEX EQUIV	NAME FOR OUTPUT FIELD VALUE
0	0	1	1	1	1	0	1	1	1	1	1EF	OUTA0
0	0	1	1	1	1	0	1	0	1	1	1EB	OUTA1
0	0	1	1	1	1	0	0	1	1	1	1E7	OUTBD0
0	0	1	1	1	1	0	0	0	1	1	1E3	OUTBD1

Figure 2-108. Output Values for Segments of HSCYCA and HSCYCBD

STATEMENT NO.	LABEL	MICROINSTRUCTION	COMMENTS
1	A:	OUTA0, IF (CREG = 0) THEN LOAD PL (7 #D);	" INITIALIZE LOOP COUNTER FOR 6 ITERATIONS "
2		OUTA0, DEC;	" DECREMENT LOOP COUNTER "
3		OUTA0, PUSHCNTR;	" SAVE LOOP COUNT "
4		OUTA0, CALL PL (SEGA0);	" EXECUTE HSCYCA MICROPROGRAM "
5		OUTA1, POPCNTR;	" RESTORE LOOP COUNT "
6		OUTA1, IF (CREG = 0) THEN GOTO PL (EXIT A);	" TEST LOOP EXIT CONDITION "
7		OUTA1, GOTO PL (A);	" CONTINUE EXECUTING LOOP "
8	EXIT A:	OUTA1, GOTO PL (B);	" COMPENSATE FOR ONE CLOCK LOSS "
9	B:	OUTBD0, IF (CREG = 0) THEN LOAD PL (5 #D);	" INITIALIZE FOR 4 ITERATIONS "
10		OUTBD0, DEC;	
11		OUTBD0, PUSHCNTR;	
12		OUTBD0, CALL PL (SEGBD0);	
13		OUTBD1, POPCNTR;	
14		OUTBD1, IF (CREG = 0) THEN GOTO PL (EXIT B);	
15		OUTBD1, GOTO PL (B);	
16	EXIT B:	OUTBD1, GOTO PL (C);	
17	C:	SEE FIGURE 2-102	" EXECUTE MICROPROGRAM C"
18	D:	OUTBD0, IF (CREG = 0) THEN LOAD PL (4 #D);	" INITIALIZE FOR 3 ITERATIONS "
19		OUTBD0, DEC;	
20		OUTBD0, PUSHCNTR;	
21		OUTBD0, CALL PL (SEGBD0);	
22		OUTBD1, POPCNTR;	
23		OUTBD1, IF (CREG = 0) THEN GOTO PL (EXIT D);	
24		OUTBD1, GOTO PL (D);	
25	EXIT D:	OUTBD1, GOTO PL (A);	

Figure 2-109. Microprogram to Generate Periods A-D

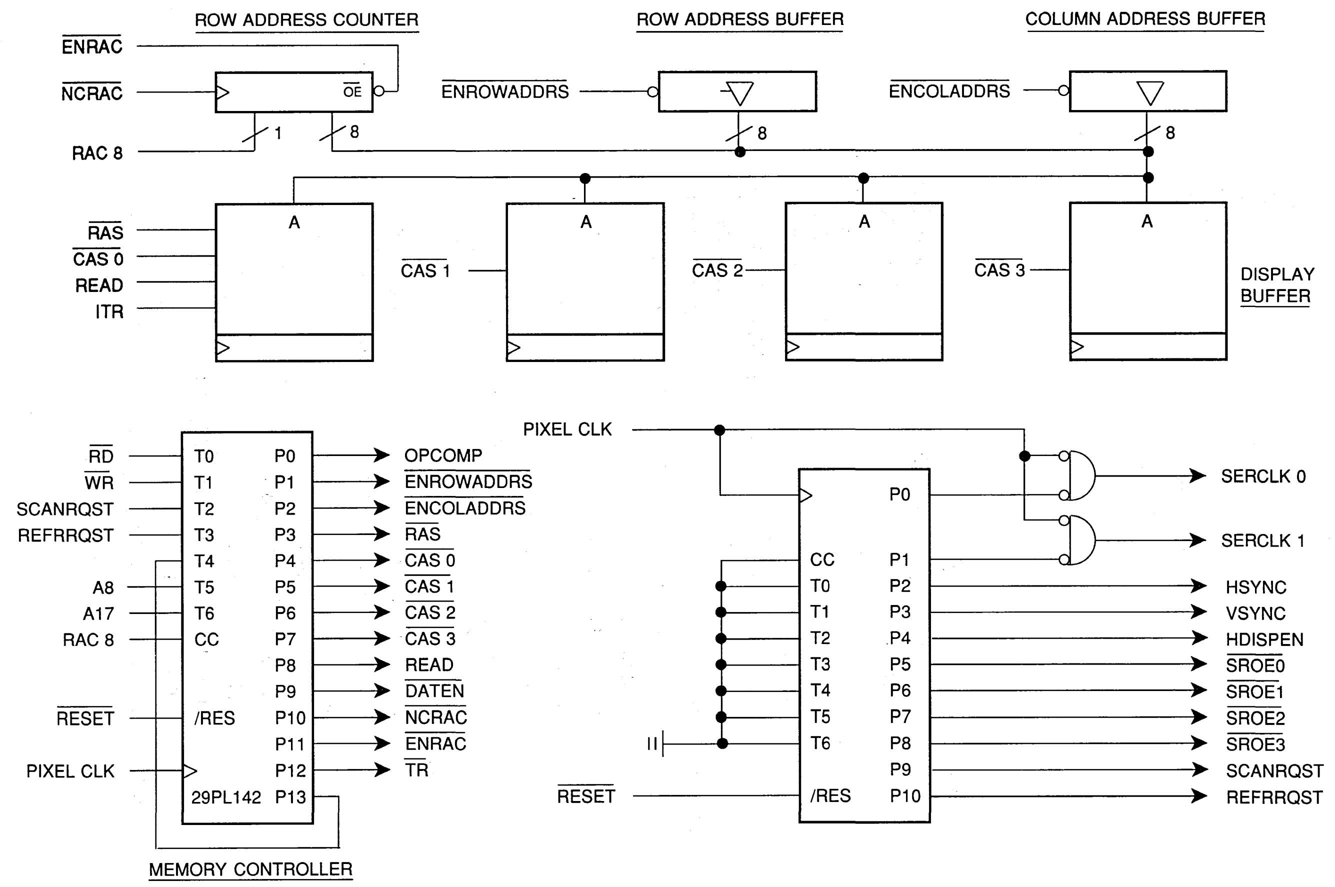

ROW ADDRESS COUNTER
ROW ADDRESS BUFFER
COLUMN ADDRESS BUFFER
$\overline{\text{ENRAC}}$
$\overline{\text{NCRAC}}$
RAC 8
$\overline{\text{OE}}$
1
8
$\overline{\text{ENROWADDRS}}$
$\overline{\text{ENCOLADDRS}}$
8
8
A
A
A
A
$\overline{\text{RAS}}$
$\overline{\text{CAS 0}}$
READ
ITR
$\overline{\text{CAS 1}}$
$\overline{\text{CAS 2}}$
$\overline{\text{CAS 3}}$
DISPLAY BUFFER
$\overline{\text{RD}}$
$\overline{\text{WR}}$
SCANRQST
REFRRQST
A8
A17
RAC 8
$\overline{\text{RESET}}$
PIXEL CLK
T0
T1
T2
T3
T4
T5
T6
CC
/RES
29PL142
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
OPCOMP
$\overline{\text{ENROWADDRS}}$
$\overline{\text{ENCOLADDRS}}$
$\overline{\text{RAS}}$
$\overline{\text{CAS 0}}$
$\overline{\text{CAS 1}}$
$\overline{\text{CAS 2}}$
$\overline{\text{CAS 3}}$
READ
$\overline{\text{DATEN}}$
$\overline{\text{NCRAC}}$
$\overline{\text{ENRAC}}$
$\overline{\text{TR}}$
MEMORY CONTROLLER
PIXEL CLK
$\overline{\text{RESET}}$
CC
T0
T1
T2
T3
T4
T5
T6
/RES
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
SERCLK 0
SERCLK 1
HSYNC
VSYNC
HDISPEN
$\overline{\text{SROE0}}$
$\overline{\text{SROE1}}$
$\overline{\text{SROE2}}$
$\overline{\text{SROE3}}$
SCANRQST
REFRRQST

Figure 2-110. Block Diagram of Display Buffer and Memory Controller

RAC 8	VRAM SELECTION
0	VRAM 00, VRAM 01, VRAM 10, VRAM 11
1	VRAM 20, VRAM 21, VRAM 30, VRAM 3

Figure 2-111. VRAM Selection Using RAC8 During Shift Register Load

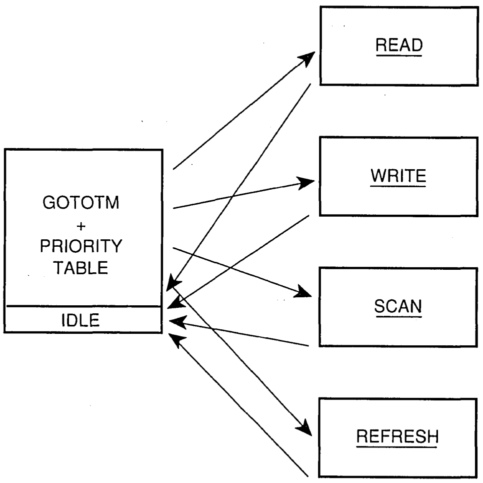


Figure 2-112. Memory Controller Arbitration

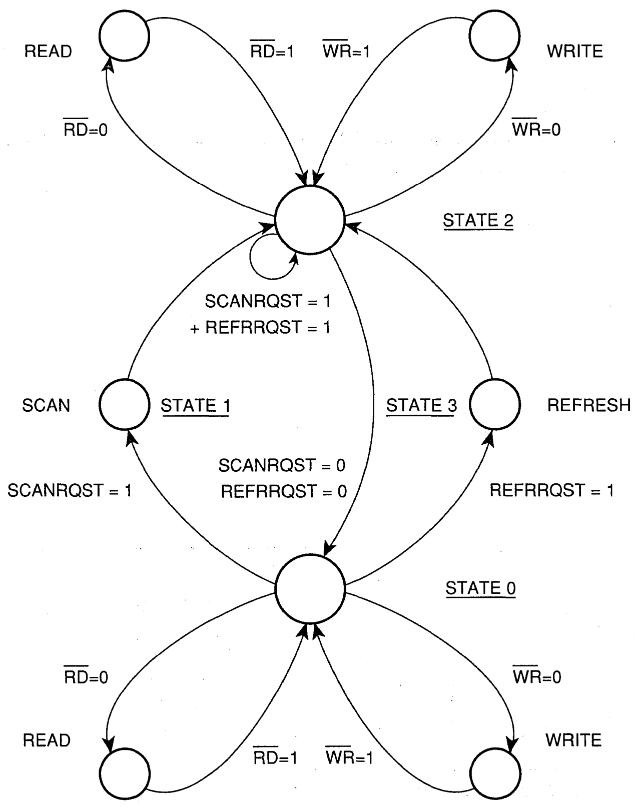


Figure 2-113. State Diagram for Memory Controller

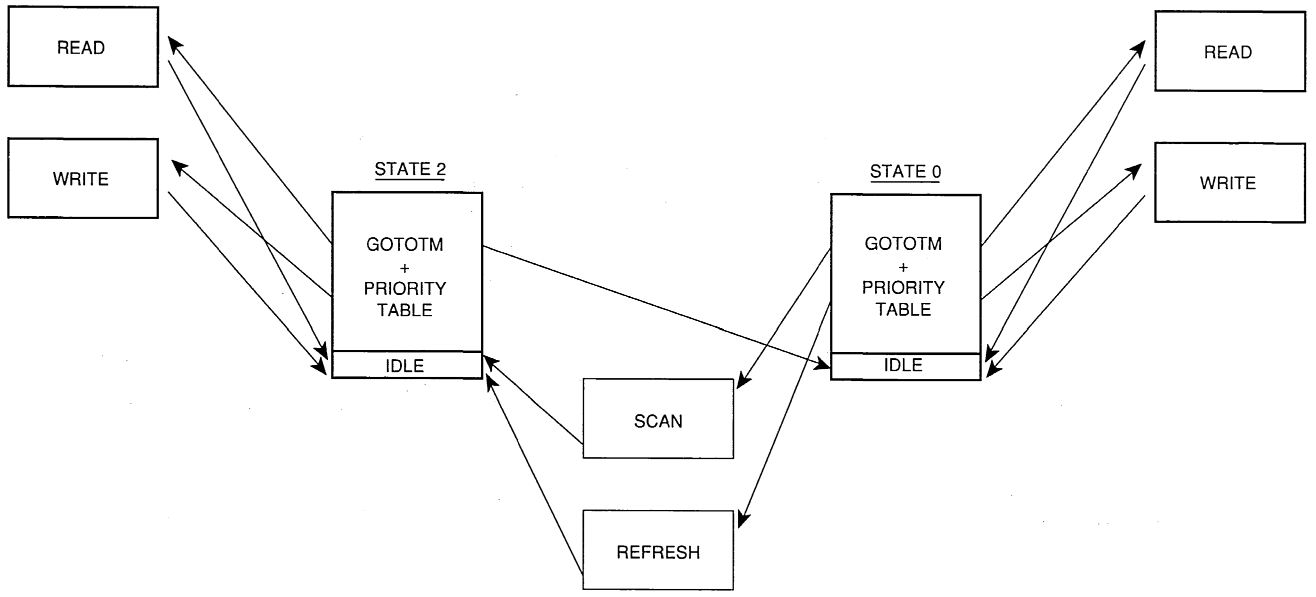


Figure 2-114. Revised Memory Controller Arbitration

```
DEVICE  (PL142)

DEFINE  $\overline{RD}$      =  T0
        $\overline{WR}$      =  T1
        SC      =  T2
        RF      =  T3
        A8      =  T5
        A17     =  T6
        RAC8    =  CC
        FAIL    =  EQ

        RD00    =  1BF4 #H     " OUTPUT VALUES DURING "
        RD01    =  19F2 #H     " READ CYCLE "
        RD020   =  1903 #H     " (FROM STATE 0) "
        RD021   =  1913 #H
        RD022   =  1923 #H
        RD023   =  1933 #H

        RD20    =  3BF4 #H     " OUTPUT VALUES DURING "
        RD21    =  39F2 #H     " READ CYCLE "
        RD220   =  3903 #H     " (FROM STATE 2) "
        RD221   =  3913 #H
        RD222   =  3923 #H
        RD223   =  3933 #H

        WR00    =  1AF4 #H     " OUTPUT VALUES DURING "
        WR01    =  18F2 #H     " WRITE CYCLE "
        WR020   =  1803 #H     " (FROM STATE 0) "
        WR021   =  1813 #H
        WR022   =  1823 #H
        WR023   =  1833 #H

        WR20    =  3AF4 #H     " OUTPUT VALUES DURING "
        WR21    =  38F2 #H     " WRITE CYCLE "
        WR220   =  3803 #H     " (FROM STATE 2) "
        WR221   =  3813 #H
        WR222   =  3823 #H
        WR223   =  3833 #H

        RF01    =  3B0C#H      " OUTPUT VALUES DURING "
        RF02    =  3B04 #H     " REFRESH CYCLE "
        RF03    =  3BF4 #H

        SC01    =  23FC #H     " OUTPUT VALUES DURING "
        SC02    =  23F4 #H     " SCAN REQUEST CYCLE "
        SC03    =  33F4 #H
        SC040   =  3334 #H
        SC050   =  3F34 #H
        SC041   =  33C4 #H
        SC051   =  3FC4 #H
        IDLE 0  =  1BFC#H      " IDLE VALUES IN STATE 0 "
        IDLE2   =  3BFC#H;     " IDLE VALUES  IN STATE 2 "
```

Figure 2-115. Memory Controller Microprogram for Example 5

```
BEGIN

          " STATE 0 JUMP TABLE "

          IDLE 0, IF (NOT FAIL) THEN GOTO ERROR;
          IDLE 0, IF (NOT FAIL) THEN GOTO WRITE00;
          IDLE 0, IF (NOT FAIL) THEN GOTO READ00;
ST0:      IDLE 0, IF (NOT FAIL) THEN GOTO TM (F #H);
          IDLE 0, IF (NOT FAIL) THEN GOTO ERROR;
          IDLE 0, IF (NOT FAIL) THEN GOTO SCAN;
          IDLE 0, IF (NOT FAIL) THEN GOTO SCAN;
          IDLE 0, IF (NOT FAIL) THEN GOTO SCAN;
          IDLE 0, IF (NOT FAIL) THEN GOTO ERROR;
          IDLE 0, IF (NOT FAIL) THEN GOTO REFR;
          IDLE 0, IF (NOT FAIL) THEN GOTO REFR;
          IDLE 0, IF (NOT FAIL) THEN GOTO REFR;
          IDLE 0, IF (NOT FAIL) THEN GOTO ERROR;
          IDLE 0, IF (NOT FAIL) THEN GOTO ERROR;
          IDLE 0, IF (NOT FAIL) THEN GOTO ERROR;
          IDLE 0, IF (NOT FAIL) THEN GOTO ERROR;

          " STATE 2 JUMP TABLE "

          IDLE 2, IF (NOT FAIL) THEN GOTO ERROR;
          IDLE 2, IF (NOT FAIL) THEN GOTO ST0;
          IDLE 2, IF (NOT FAIL) THEN GOTO ST0;
          IDLE 2, IF (NOT FAIL) THEN GOTO ST0;
          IDLE 2, IF (NOT FAIL) THEN GOTO ERROR;
          IDLE 2, IF (NOT FAIL) THEN GOTO WRITE20;
          IDLE 2, IF (NOT FAIL) THEN GOTO READ20;
ST2:      IDLE 2, IF (NOT FAIL) THEN GOTO TM (1F #H);
          IDLE 2, IF (NOT FAIL) THEN GOTO ERROR;
          IDLE 2, IF (NOT FAIL) THEN GOTO WRITE20;
          IDLE 2, IF (NOT FAIL) THEN GOTO READ20;
          IDLE 2, IF (NOT FAIL) THEN GOTO TM (#1F #H);
          IDLE 2, IF (NOT FAIL) THEN GOTO ERROR
          IDLE 2, IF (NOT FAIL) THEN GOTO ERROR
          IDLE 2, IF (NOT FAIL) THEN GOTO ERROR
          IDLE 2, IF (NOT FAIL) THEN GOTO ERROR
```

Figure 2-115 (Continued)

```
                " ROUTINE TO SYNTHESIZE READ CYCLE TIMING"
                " (SEE FIGURE 2-78) "
                " ENTRY IS FROM STATE 0 "

READ00:         RD00,     IF (A8) THEN GOTO PL (READ01);
                RD01,     IF (A17) THEN GOTO PL (READ02);
                RD020,    CONTINUE;
                RD020,    CONTINUE;
                RD020,    IF (/RD) THEN GOTO TM (F #H) ELSE WAIT;

                " A8 = 0 & A17 = 0 "

READ01:         RD01,     IF (A17) THEN GOTO PL (READ03);
                RD021,    CONTINUE;
                RD021,    CONTINUE;
                RD021,    IF (/RD) THEN GOTO TM (F #H) ELSE WAIT;

                " A8 = 1 & A17 = 0 "

READ02:         RD022,    CONTINUE;
                RD022,    CONTINUE;
                RD022,    IF (/RD) THEN GOTO TM (F #H) ELSE WAIT;

                " A8 = 0 & A17 = 1 "

READ03:         RD023,    CONTINUE;
                RD023,    CONTINUE;
                RD023,    IF (/RD) THEN GOTO TM (F #H) ELSE WAIT;

                " A8 = 1 & A17 = 1 "
```

Figure 2-115 (Continued)

```
           " ROUTINE TO SYNTHESIZE READ CYCLE TIMING"
           " (SEE FIGURE 2-78) "
           " ENTRY IS FROM STATE 2 "

READ20:    RD20,    IF (A8) THEN GOTO PL (READ21);
           RD21,    IF (A17) THEN GOTO PL (READ22);
           RD220,   CONTINUE;
           RD220,   CONTINUE;
           RD220,   IF (/RD) THEN GOTO TM (1F #H) ELSE WAIT;

           " A8 = 0 & A17 = 0 "

READ21:    RD21,    IF (A17) THEN GOTO PL (READ23);
           RD221,   CONTINUE;
           RD221,   CONTINUE;
           RD221,   IF (/RD) THEN GOTO TM (1F #H) ELSE WAIT;

           " A8 = 1 & A17 = 0 "

READ22:    RD222,   CONTINUE;
           RD222,   CONTINUE;
           RD222,   IF (/RD) THEN GOTO TM (1F #H) ELSE WAIT;

           " A8 = 0 & A17 = 1 "

READ23:    RD223,   CONTINUE;
           RD223,   CONTINUE;
           RD223,   IF (/RD) THEN GOTO TM (1F #H) ELSE WAIT;

           " A8 = 1 & A17 = 1 "
```

Figure 2-115 (Continued)

```
            " ROUTINE TO SYNTHESIZE WRITE CYCLE TIMING"
            " (SEE FIGURE 2-78) "
            " ENTRY IS FROM STATE 0 "

WRITE00:    WR00,    IF (A8) THEN GOTO PL (WRITE01);
            WR01,    IF (A17) THEN GOTO PL (WRITE02);
            WR020,   CONTINUE;
            WR020,   CONTINUE;
            WR020,   IF ($\overline{WR}$) THEN GOTO TM (F #H) ELSE WAIT;

            " A8 = 0 & A17 = 0 "

WRITE01:    WR01,    IF (A17) THEN GOTO PL (WRITE03);
            WR021,   CONTINUE;
            WR021,   CONTINUE;
            WR021,   IF ($\overline{WR}$) THEN GOTO TM (F #H) ELSE WAIT;

WRITE02:    WR022,   CONTINUE;
            WR022,   CONTINUE;
            WR022,   IF ($\overline{WR}$) THEN GOTO TM (F #H) ELSE WAIT;

            " A8 = 0 & A17 = 1 "

WRITE03:    WR023,   CONTINUE;
            WR023,   CONTINUE;
            WR023,   IF ($\overline{WR}$) THEN GOTO TM (F #H) ELSE WAIT;

            " A8 = 1 & A17 = 1 "
```

Figure 2-115 (Continued)

```
           " ROUTINE TO SYNTHESIZE WRITE CYCLE TIMING"
           " (SEE FIGURE 2-78) "
           " ENTRY IS FROM STATE 2 "

WRITE20:   WR20,    IF (A8) THEN GOTO PL (WRITE21);
           WR21,    IF (A17) THEN GOTO PL (WRITE22);
           WR220,   CONTINUE;
           WR220,   CONTINUE;
           WR220,   IF (/WR) THEN GOTO TM (1F #H) ELSE WAIT;

           " A8 = 0 & A17 = 0 "

WRITE21:   WR21,    IF (A17) THEN GOTO PL (WRITE23);
           WR221,   CONTINUE;
           WR221,   CONTINUE;
           WR221,   IF (/WR) THEN GOTO TM (1F #H) ELSE WAIT;

           " A8 = 1 & A17 = 0 "

WRITE22:   WR222,   CONTINUE;
           WR222,   CONTINUE;
           WR222,   IF (/WR) THEN GOTO TM (1F #H) ELSE WAIT;

           " A8 = 0 & A17 = 1 "

WRITE23:   WR223,   CONTINUE;
           WR223,   CONTINUE;
           WR223,   IF (/WR) THEN GOTO TM (1F #H) ELSE WAIT;

           " A8 = 1 & A17 = 1 "
```

Figure 2-115 (Continued)

```
          " ROUTINE TO SYNTHESIZE REFRESH CYCLE TIMING"
          " (SEE FIGURE 2-79) "

REFR:     RF01,    CONTINUE;
          RF02,    CONTINUE;
          RF03,    CONTINUE;
          RF03,    IF (NOT FAIL) THEN GOTO PL (ST2);

          " ROUTINE TO SYNTHESIZE SCAN CYCLE TIMING "
          " (SEE FIGURE 2-79) "

SCAN:     SC01,    CONTINUE;
          SC02,    CONTINUE;
          SC03,    IF (RAC8) THEN GOTO PL (SCAN0);
          SC040,   CONTINUE;
          SC050,   IF (NOT FAIL) THEN GOTO PL (ST2);

SCAN0:    SC041,   CONTINUE;
          SC051,   IF (NOT FAIL) THEN GOTO PL (ST2);

ERROR:    " THIS ROUTINE IS ENTERED WHENEVER "
          " BOTH RD AND WR ARE ACTIVE "
          " OR BOTH SCANRQST AND REFRRQST ARE ACTIVE "
```

Figure 2-115 (Continued)

CHAPTER 3

ARTICLE REPRINTS

Fuse-programmable chip takes command of distributed systems

The first fuse-programmable controller eliminates bulky and expensive designs, freeing distributed intelligence to carve out a greater niche for itself.

In much the same way that cars and highways spawned the suburbs, standard microprocessor buses and add-on boards have distributed processing intelligence, revolutionizing the design of digital systems. Breaking systems into independent modules shortens the design cycle, eases upgrades, and accelerates fault diagnosis. But despite these advantages, an essential element has been missing: a one-chip controller geared specifically to the needs of distributed intelligence.

Without that critical ingredient, engineers have been forced to turn to less than optimal solutions. One approach relies on boards packed with as many as 35 SSI and MSI devices. Another tack is to go with a powerful—yet costly—VLSI chip. Alternatively, a programmable logic device can be pressed into service, but such circuitry lacks the computing power to control peripherals.

The missing element, a fuse-programmable controller chip, is now here. By mixing intelligence and control, the Am29PL141 stakes out new territory for distributed systems. The 20-MHz IC combines for the first time all of the elements of an intelligent microcode controller. Its powerful sequencing logic steps through the controller's 64-by-32-bit pipelined PROM. That fuse-programmable memory stores a user-defined microprogram drawn from a set of 29 microinstructions, including a repertoire of jumps, multiple branches (or case statements), and subroutine calls. All can be executed conditionally, depending upon the outcome of one of eight tests. In addition, a serial shadow register on the 28-pin chip helps designers diagnose system troubles right down to a particular IC. (In the past, expediency often dictated that complex trouble-shooting be avoided for as long as possible.)

Om Agrawal and **Deepak Mithani**
Advanced Micro Devices Inc., 901 Thompson Pl., P.O. Box 3453, MS 47, Sunnyvale, CA 94088; (408) 749-2903.

Four basic blocks

The controller comprises four main functional blocks. Three of them—the microaddress control logic, condition code selector, and microinstruction decoder—form the cornerstone of the controller, the address sequencer. The fourth is a microprogram memory (64 by 32 bits) with a pipelined register and serial shadow register (Fig. 1).

For the most part, the elements of the address sequencer are fairly typical. Nevertheless, the way in which they are organized and connected, as well as the instruction set, make the chip unique. For example, the microaddress control portion of the sequencer contains one register for counting loops and another for stacking subroutine return addresses. Yet either register can be employed to double the capacity of the other. Consequently, the chip can nest two levels of loop counting or two levels of subroutine branching (the instruction set reflects those abilities). And the high degree of interaction between elements, particularly within the microaddress control logic, makes necessary a highly sophisticated microinstruction set.

The microaddress control logic is the brain of the address sequencer, since it generates the addresses that access the microinstructions. At any time, the address that is called depends on the preceding instruction and the outcome of any conditional tests.

Within the control logic, a program-counter multiplexer supplies the PROM's 6-bit address. The multiplexer takes the address from a microprogram counter, incremented program counter, branch control logic, or subroutine register. Because the program counter contains the address of the currently executing instruction, that instruction is executed again when the program counter is selected as the address source. As a result, the counter plays a fundamental role in tallying loops and executing "wait until true" instructions.

The incrementer holds the next address in the sequence, and is the expected source when no jumps are executed and no branch or subroutine conditions exist. When conditional statements like "if . . . then . . . else" and multiple branches pass the required tests, or when unconditional jumps are executed, the branch control logic supplies the address. Finally, when the program calls a subroutine, the subroutine register supplies the necessary address.

A multiplexer selects one of three address sources. If only one stack level is needed, the value stored in the subroutine register is chosen. When the count register feeds the subroutine register, however, it furnishes an additional stack level. The third source, the incrementer, supplies the subroutine's return addresses.

Doing double duty

If not needed for a second subroutine level, the count register can, among other functions, execute iterative loops and time external events. To accomplish the former, the controller loads the register with the number of iterations to be run. Each iteration decrements the register until it reaches zero. The zero-detection logic associated with the counter informs the chip's microinstruction decoding logic when the register "bottoms out."

Using the same logic, an instruction can be repeated a set number of times. Repeated executions of the same instruction is a simple way to insert wait states and, therefore, build an interface to different microprocessors and peripherals.

The count register is loaded from any of four sources: a decrementer, for normal loops; an instruction field; the subroutine register; and the branch-control logic. The last derives a 6-bit value from a data field in a microinstruction.

The branch-control logic, a powerful block within the sequencer, calculates the 6-bit value either by applying the microinstruction data field directly or by using it to mask the chip's six test inputs, T_0 through T_5. In the second case, the masked input actually becomes the branch address. Moreover, either the data field or masked test value serves as both a branch address and a count value.

The same control logic also compares the masked test inputs to a constant in a microinstruction field. The outcome of this check affects a flip-flop. The latter's condition itself becomes a factor in deciding conditional branch and subroutine instructions. If a match occurs, the flip-flop is set. Alternatively, the flip-flop remains unchanged if there is no match. Because the flip-flop does not change when there is no match, it is particularly useful for comparing ASCII characters and other 6-bit fields, as well as for successively checking the chip's test inputs.

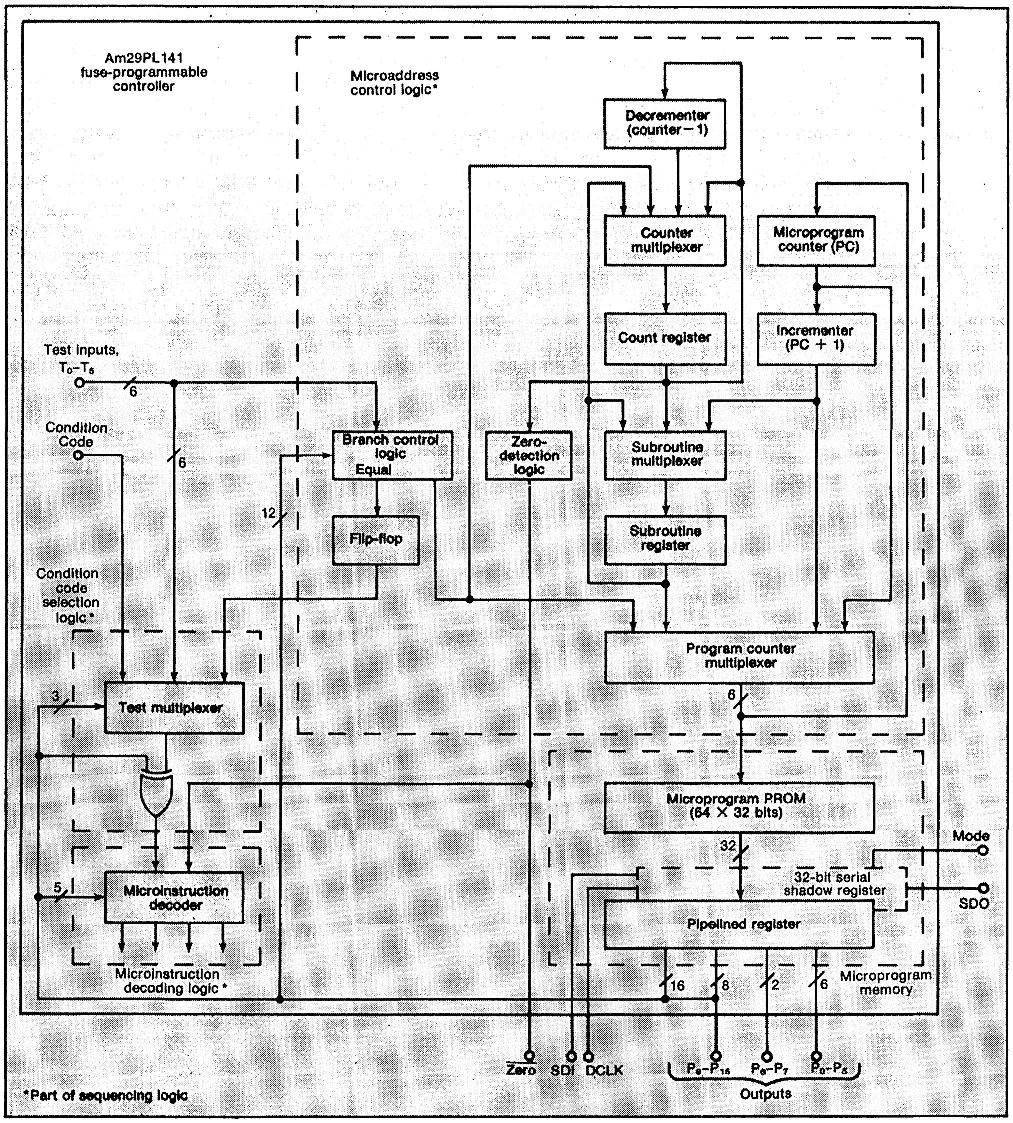
Controlling conditions

A set flip-flop is one of the eight aforementioned tests that fulfills a conditional branch or subroutine. Through its condition-code selection logic, the controller is able to check each of its six test input lines, as well as the Condition Code input. Further, an exclusive-OR gate within the selection logic switches the meaning, or interpretation, of a test result. In other words, with no external hardware, a test condition can be asserted either when a match occurs or when one does not.

The final component of the chip's sequencer section is the microinstruction decoder. That programmable logic array generates the IC's internal control signals based on the microinstruction being executed and the test results reported by the condition-code logic.

The IC's fourth functional block comprises the fuse-programmable microprogram memory, pipelined register, and serial shadow register. The pipeline register is 32 bits wide, and stores the microinstruction being run. The next address is calculated by the sequencer and its contents is fetched from the microprogram memory. The upper 16 bits of the pipeline's output remain within the chip to sequence addresses and control internal functions.

Only the 16 low-order bits link to the outside, as user-defined control lines. Of these, the upper byte is



1. The 20-MHz Am29PL141 is the first complete microprogrammable controller chip, making it an important building block for distributed processing systems. Its powerful sequencing logic steps the controller through its pipelined PROM. The fuse-programmable memory is 64 by 32 bits.

Fuse-programmable controller

put in the high-impedance state by setting a microinstruction's Output Enable bit to 0. Moreover, chips can be cascaded readily if more than 16 control bits are needed (Fig. 2).

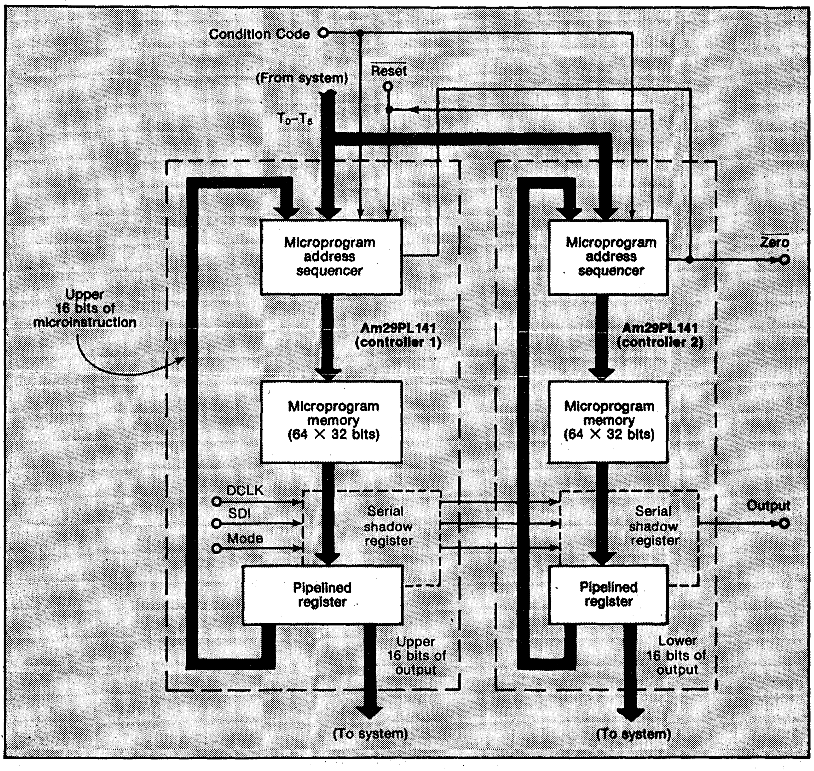
The serial shadow register, also 32-bits wide, simplifies device- and system-level diagnostics. It can be loaded in parallel with the contents of the pipelined register or loaded serially from the Serial Data Input pin. On the other hand, the serial register can also load the pipeline or shift data out serially. It also may simply hold the data sent to it.

To check out the chip, an instruction is shifted serially into the shadow register and then loaded in parallel into the pipeline. Doing so forces the instruction to be executed, and its results transferred back from the pipeline into the shadow register. From there it is shifted out for diagnosis. If all the shadow registers in a system are tied together, a series diagnostic loop is created that isolates a problem down to a single chip.

The shadow fuse

A separate fuse must be blown to set up the serial shadow register. When that is done, four pins are redefined to handle diagnostics. Specifically, the Condition Code and Zero lines and Output Data Bits 6 and 7 become, respectively, the Serial Data In (SDI), Serial Data Out (SDO), Diagnostic Clock (DCLK), and Mode control lines.

The strength of any controller—and the advantage



2. When an application calls for more than 16 control bits, two or more chips can be cascaded horizontally. The lower 16 bits of each of the chip's 32-bit microinstructions (of which there are 29) serve to control a system's components. Eight of these 16 control bits can be put into a high-impedance state under microinstruction control; the other 8 bits are always enabled.

Fuse-programmable controller

of a microprogrammed system that employs it—lies in an engineer's ability to specify the sequence in which microinstructions are executed. To ensure that ability, the controller executes all the basic high-level constructs required for structured microprogramming. Its 29 op codes include sequential instructions, conditional instructions, dual branching forks, and multibranching case statements. Iterative executions, like For, While, When, and Until, round out the set. In addition, Jump, Jump to Subroutine, Loop, and Compare instructions allow designers to store very complex algorithms in the chip's 64-word memory.

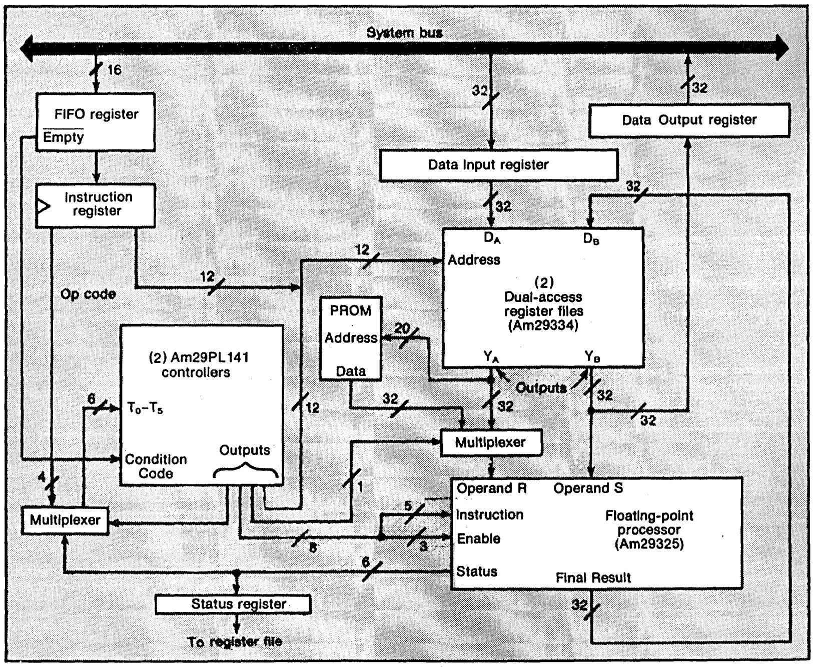
Instruction formats fall into two categories. The first is for general microinstructions; the second is for the chip's Compare instructions. The latter compare a 6-bit test input to a masked constant. The Compare instructions are well-suited for character searches, as well as key searches in a look-up table.

A single-precision, floating-point peripheral board (Fig. 3) presents a good example of the part that the controller plays in a distributed system. As a microprogrammed design, the peripheral serves as an add-on math accelerator card that plays with different hosts and buses. The controller orchestrates the actions of the floating-point processor, and various registers, register files, and memory chips.

Simple arithmetic

The processor is simple to use, partly because it incurs no pipeline delays. It conforms to IEEE and other industry standards, and takes only a single clock cycle to add, subtract, or multiply. It needs five cycles to divide, using the Newton-Raphson method that inverts one of the factors and multiplies. In operation, to divide X by Y the chip fetches the approximate inverse of Y from a PROM-based table and multiplies it by X. One or two iterations of this method increase the initial accuracy.

The floating-point board works with a microword

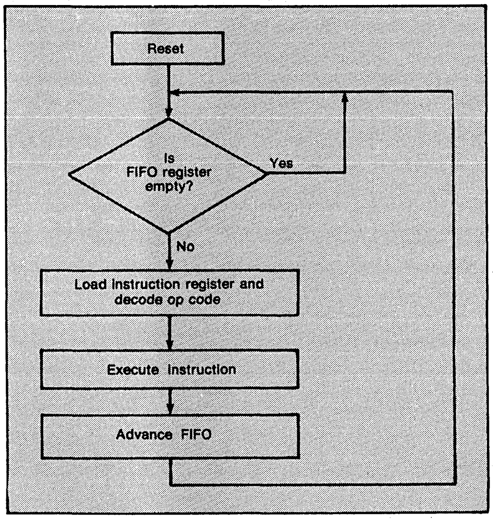


3. In a typical application, the controller oversees the workings of a floating-point processor board. Host instructions sent to the FIFO and instruction registers initiate subroutines in the controller that generate the signals that run the board. Two controllers are employed to supply the necessary number of control signals.

of at least 25 bits, 9 more than available with one controller. Thus the design employs two controller chips. The floating-point processor requires five command bits. Three are instructions and two select the input source. It also needs three control bits to enable its trio of data registers. Two other chips (each a dual-access four-port register of 64 words by 18 bits) temporarily store commands. They accept 12 register address bits (6 for source and 6 for destination) from the host or the controller's microprogram memory.

Data passes to and from the host through input and output registers on the board, which call for their own enable signals. Another bit is needed to advance the FIFO instruction register. One is necessary to enable and another to select a status word. (The floating-point chip supplies status information, which is available to the controller through its test inputs as well as to the host through the register file.) Seven control bits are left for miscellaneous tasks.

Operation begins when at least one 16-bit instruction is loaded from the host into the peripheral's FIFO register. The instruction consists of a 4-bit op code, a 6-bit source-register address, and a 6-bit address for the second source register, which also stores



4. Loading an instruction into the FIFO starts the peripheral and activates the controller, which loads an external instruction register and decodes the op code field. The op code initiates a subroutine in the controller, issuing the proper control signals, advancing the FIFO register, and loading the next instruction.

the results. Until an instruction is received, the controller is in the wait state (Fig. 4). When an instruction arrives, however, the FIFO's $\overline{\text{Empty}}$ signal activates the controller, which then reads the command from the FIFO into a separate instruction register. The controller also loads the instruction's op code into its test inputs. It then masks the two unused test bits and jumps to a subroutine that performs the operation specified by the op code. After completing it, the controller advances the FIFO register to load the next instruction.

The peripheral executes up to 16 op codes. The first eight are single-cycle operations and identical to those of the floating-point processor. They consist of addition, subtraction, multiplication, and format conversion instructions. The remaining op codes are used to load, store, and divide data, and a multiple cycle instruction multiplies and accumulates values. The four remaining op codes can be defined by the user to implement application-related operations.

Software and hardware tools are a necessary part of such projects as the foregoing peripheral. The software assembles high-level microprograms and a JEDEC output file that specifies the fuse pattern to be burned into the PROM array. Currently, a program called Fuse Formatter, which runs on the IBM PC personal computer, lets designers enter hexadecimal code that corresponds to PROM data. From that code, the program creates a file that is downloaded directly to one of several PROM programmers. The latter blow the corresponding fuses in the microprogram memory and are the only required hardware tools. □

Om Agrawal is the product planning manager for programmable logic devices at AMD. He has designed 16- and 32-bit minicomputers, and is the coauthor of a book on high-speed memory systems. Agrawal holds a PhD in electrical engineering and computer science from Iowa State University. He also received an MBA from the University of Santa Clara.

As a senior product marketing engineer for the company's microprocessor division, Deepak Mithani, designs and markets bipolar microprocessors. He earned a BSEE from India's Maharaja Sayajirao University and an MSEE from the University of Wisconsin.

FPCs and PLDs simplify VME Bus control

By using fuse-programmable controllers (FPCs) and PLDs, you can implement VME Bus control in your computer system with a minimum of hardware. Just a few chips per plug-in module, and a bus arbiter, can perform all bus-control functions. This article, Part 1 of a 2-part series, describes the bus-arbitration process and control functions and shows you how to implement the VME Bus protocol in an FPC. Part 2 will describe the implementation of slave controllers, discuss interrupt handling, and provide tools for programming the FPC.

Arthur Khu, *Advanced Micro Devices*

Until recently, you needed many ICs—sequencers, microprogrammed control stores, and other MSI/LSI chips—to implement VME Bus control in a computer system. Now, however, you can use a minimum of hardware to handle the bus's intermodule communication functions. Just a few fuse-programmable controllers (FPCs) and programmable-logic devices (PLDs) relieve the CPU of all bus-control functions.

A typical VME Bus-based computer system comprises one or more master modules (eg, CPU boards), one or more slave modules (eg, cache/memory boards), a bus arbiter, and interrupt-handling circuitry. A master initiates a data transfer to a slave by requesting control of the data bus from the bus arbiter. Once bus control has been granted to a master, the master and slave exchange control signals according to predefined protocols that guarantee an orderly transfer of data between the communicating modules. The interrupt-handling circuitry services all interrupt requests.

By using the streamlined design in **Fig 1**, you can implement most intermodule communication in a VME Bus-based system by using just two types of VLSI devices: the Am29PL141 fuse-programmable device and the AmPAL22V10 programmable-logic device. For the remainder of the bus-control functions, you'll need some bus-arbitration circuitry, which must occupy a particular slot on the VME Bus backplane, but which can reside on the same board with the bus master of highest priority.

When you design VME Bus control into your system,

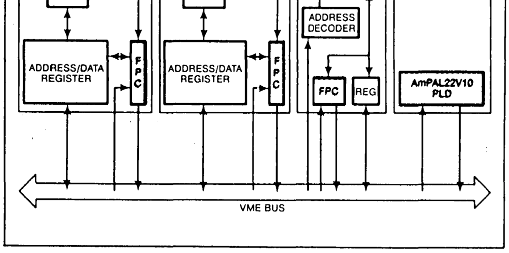


Fig 1—You can implement VME Bus control *in a computer system by using just a few fuse-programmable controllers (FPCs) and programmable-logic devices.*

You can implement VME Bus control in your computer system by embedding the bus-control functions in state machines that comply with the VME Bus protocol.

your first step is to consider the VME Bus protocols. These protocols specify the steps your circuitry must take to perform any bus-related operation, such as the transfer of data between two modules. You can describe these protocols by means of flow diagrams that show how the various interface signals reflect the interactions between the communicating modules. **Fig 2,** for example, shows how the priority bus arbiter resolves simultaneous bus requests from two modules that use the same request line.

Next, you analyze the flow diagrams to pick out the functions that can be incorporated into FPCs or PLDs, and you define state machines for those functions. You can describe the state machines abstractly, by Boolean logic equations, or diagrammatically, by means of standard flow-chart symbols (eg, rectangles to represent processes and diamonds to represent control-flow decisions). Finally, you must write programs for the state machines in a high-level or assembly language. You repeat this process to design each of the four main types of VME Bus controllers: bus arbiters, masters, slaves, and interrupt handlers.

Before any master module can perform a data transfer, it must request control of the data bus from the bus arbiter; the arbiter must check to see whether the bus is free and must resolve any contention between two masters of equal priority. For example, consider a priority-option bus arbiter implemented on a single PLD. The arbiter will monitor the four bus-request lines ($\overline{BR_{0\text{-}3}}$) and assign the highest priority to $\overline{BR_3}$.

As the flow diagram **(Fig 2)** shows, the arbiter grants the bus to the requesting module that's using the highest active request line, which is $\overline{BR_1}$ in this example. To enable your bus arbiter to resolve simultaneous bus requests from two or more modules that use the same request line, you must daisy-chain the associated bus-grant signal to all the devices using that request line. Therefore, the arbiter must be in the first slot of the VME Bus system. The module that's closest to the bus arbiter will have the highest priority: If it requests the bus, it will receive the bus grant and lock out any modules farther down the chain **(Listing 1).** The AND/OR array of the PLD processes all bus requests in parallel, so that priority arbitration is complete in only one clock cycle.

LISTING 1

```
IF (/BBSY*(BR0+BR1+BR2+BR3)) THEN  "if bus not busy and a      "
   BEGIN                           "  request line is active   "
   IF (BR3) THEN                   "if BR3 is active, grant    "
        BG3IN := 1 ;               "  bus to device on BR3     "
   IF (/BR3*BR2) THEN              "activate bus grant         "
        BG2IN := 1;                "  daisy chain 2            "
   IF (/BR3*/BR2*BR1) THEN         "if BR1 is active and BR3"
        BG1IN := 1;                "  and BR2 are not, then    "
                                   "  grant bus to device      "
                                   "  using request line 1     "
   IF (/BR3*/BR2*/BR1*BR0) THEN    "BR0-3 and BG0IN to
        BG0IN := 1 ;               "  BG3IN are active low     "
   END ;

   IF (/BBSY*BG1IN) THEN "if BG1IN is asserted in response to a request "
       BG1IN := 1;       "over line BR1, then continue asserting BG1IN "
                         "until requesting device drives BBSY active    "
```

LISTING 2

```
   IF (BBSY*(BR0+BR1+BR2+BR3)) THEN
       BEGIN
       IF (BR3*(/(MASTER[1:0] = 3) ) ) THEN
           BCLR := 1;     "assert BCLR if MASTER < > 3"
       IF (/BR3*BR2*(/(MASTER[1:0] = 3) ) ) THEN
           BCLR := 1;
       IF (MASTER[1:0] = 0) THEN
           BCLR := 1;
       END;
```

Priority arbitration options

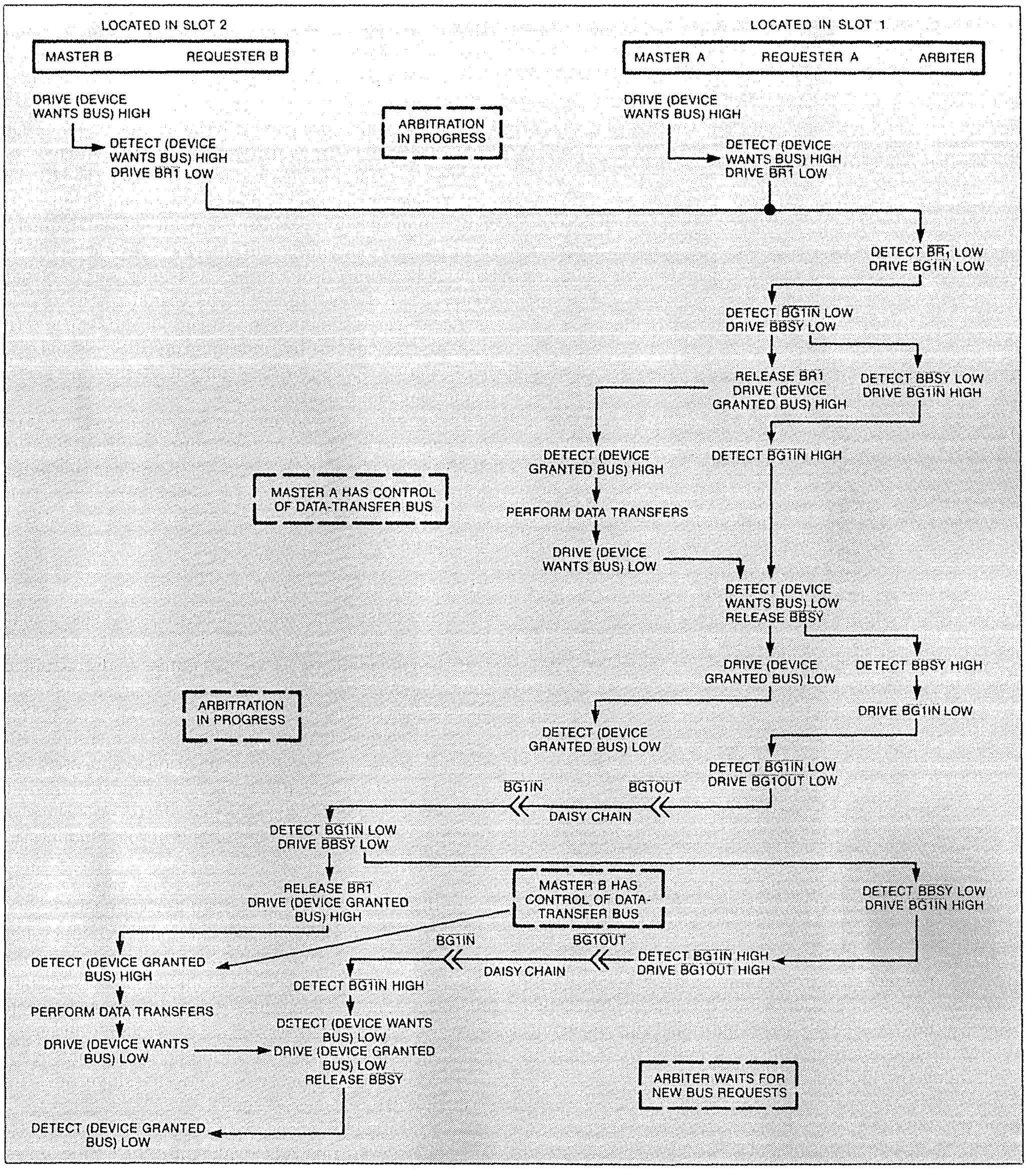
Sometimes, the arbiter must force the current bus master to relinquish control of the bus; this situation occurs, for example, when a bus master of higher priority initiates a bus request. The arbiter examines the priorities of both the current bus master and the new requester. If these conditions meet the predefined bus-clear conditions, the arbiter asserts the bus-clear signal ($\overline{BCLR}$).

In the design in **Fig 1,** the bus arbiter keeps track of the current bus master's priority by recording the bus-request line that was used to gain control of the bus. For example, if the current bus master used $\overline{BR_2}$ to gain control of the bus, it sets two output registers called Bus_Master to the binary value 2. Either of the two following conditions will activate $\overline{BCLR}$:

- The value in Bus_Master is 2, 1, or 0 and the active bus request line is 3 or 2.
- The value in Bus_Master is 0, and any bus-request line is active.

When the value in Bus_Master is 3, the arbiter will not honor any bus requests until the Bus Busy line ($\overline{BBSY}$) is high. **Listing 2** gives the logical expression of these conditions. Once having asserted $\overline{BCLR}$, the arbiter holds this line in the active state until the current bus master releases $\overline{BBSY}$.

Be sure to define the $\overline{BCLR}$ in such a way that uninterruptible devices can use bus-request line 3, which has the highest priority. Devices that temporari-



LOCATED IN SLOT 2
MASTER B
REQUESTER B
LOCATED IN SLOT 1
MASTER A
REQUESTER A
ARBITER
DRIVE (DEVICE WANTS BUS) HIGH
DETECT (DEVICE WANTS BUS) HIGH DRIVE BR1 LOW
ARBITRATION IN PROGRESS
DRIVE (DEVICE WANTS BUS) HIGH
DETECT (DEVICE WANTS BUS) HIGH DRIVE BR1 LOW
DETECT BR1 LOW DRIVE BG1IN LOW
DETECT BG1IN LOW DRIVE BBSY LOW
RELEASE BR1 DRIVE (DEVICE GRANTED BUS) HIGH
DETECT BBSY LOW DRIVE BG1IN HIGH
DETECT (DEVICE GRANTED BUS) HIGH
DETECT BG1IN HIGH
MASTER A HAS CONTROL OF DATA-TRANSFER BUS
PERFORM DATA TRANSFERS
DRIVE (DEVICE WANTS BUS) LOW
DETECT (DEVICE WANTS BUS) LOW RELEASE BBSY
DRIVE (DEVICE GRANTED BUS) LOW
DETECT BBSY HIGH
ARBITRATION IN PROGRESS
DRIVE BG1IN LOW
DETECT (DEVICE GRANTED BUS) LOW
DETECT BG1IN LOW DRIVE BG1OUT LOW
BG1IN
BG1OUT
DAISY CHAIN
DETECT BG1IN LOW DRIVE BBSY LOW
RELEASE BR1 DRIVE (DEVICE GRANTED BUS) HIGH
MASTER B HAS CONTROL OF DATA-TRANSFER BUS
DETECT BBSY LOW DRIVE BG1IN HIGH
DETECT (DEVICE GRANTED BUS) HIGH
BG1IN
BG1OUT
DAISY CHAIN
DETECT BG1IN HIGH DRIVE BG1OUT HIGH
DETECT BG1IN HIGH
PERFORM DATA TRANSFERS
DETECT (DEVICE WANTS BUS) LOW DRIVE (DEVICE GRANTED BUS) LOW RELEASE BBSY
DRIVE (DEVICE WANTS BUS) LOW
ARBITER WAITS FOR NEW BUS REQUESTS
DETECT (DEVICE GRANTED BUS) LOW

Fig 2—Flow diagrams help you define the VME Bus protocols. *This flow diagram illustrates how the bus arbiter chooses between requesters that have the same priority level.*

LISTING 3

```
DEVICE "MASTER" (Am29PL141)     "specify the device to use"
                                DEFINE
                                "define the test conditions"
      bus__request    = t0      "master request input              "
      bus__grant      = t1      "bus grant in line                 "
      dtack           = t2      "data transfer acknowledge line    "
      addr__strobe    = t3      "check address strobe bus line     "
      : : : : : : :             "other test conditions             "
"define the control outputs"
      off__state  = 0000#h      "off state                         "
      as          = 8000#h      "16th bit = 1 : address strobe     "
      busmaster   = 4000#h      "15th bit = 1 : inform master      "
      arb__req    = 2000#h      "14th bit = 1 : drive BRn line     "
      bbsy        = 1000#h      "13th bit = 1 : drive BBSY line    "
"signals going to the bus are gated thru inverting high-current
 bus driver; these include AS,ARB__REQUEST, and BBSY               "
        :
      signal1     = 0001#h ;    "other control signals             "
"FPC assembler format is = <label:output,statement;>, with label
 optional"
BEGIN
"wait for a bus request to be asserted high"
(A) start : off__state        , while (not bus__request) wait
                                 else goto pl(get__bus) ;
(C) bus__granted  "assert BBSY and BUSMASTER signals"
         : bbsy+busmaster
                              , if (addr__strobe = 1) then
                                 goto pl(bus__granted) ;
(D) "AS(L) must be HIGH before continuing; this means previous bus
 master is not driving the bus anymore                             "
       :     "other statements"
       :
       :
"BUS ACQUISITION microcode subroutine: the control output
 ARB__REQUEST is asserted (BRn line) until the /BGINn signal
 is received active LOW; if the bus request line from the master
 goes LOW before the bus is granted (e.g., master cancels bus
 request), then turn off the ARB__REQUEST output and return
 to the wait state START                                           "
(B) get__bus: arb__request    , if (bus__grant = 0) then
                                   goto pl(bus__granted) ;
              arb__request    , if (bus__request) then
                                   goto pl(get__bus) ;
              off__state      , if (not bus__request) then
                                   goto pl(start) ;
END. "end of source code"
```

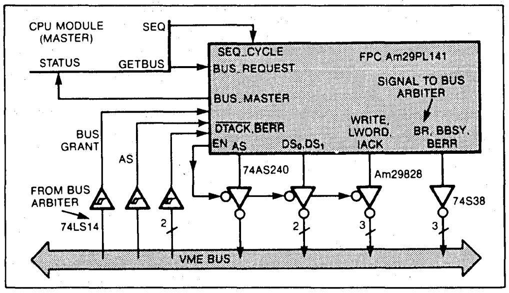


*Fig 3—**You can implement the requester logic** for a master module in an FPC. The FPC handles all bus-acquisition and data-transfer protocols.*

ly can be suspended to accommodate interrupts and higher-priority operations should be assigned to bus-request line 0. The BCLR conditions will vary with your application, but you can modify them just by redefining the high-level logic expressions.

Master modules

The master modules of VME Bus systems initiate all data transfers over the bus. A system can have one master (the CPU unit) or several (multiple processors, DMA controllers). A master module must control the bus before it can perform any data transfers. Most of the bus-control logic resides in the requester section of the module; this section handles bus acquisition and communication with other master or slave devices in the system.

You can microprogram all the requester functions into a single FPC, which serves as the interface between the master device and the VME Bus (**Fig 3**). The FPC's microprogram handles the bus-acquisition protocol. Upon receiving the bus-grant signal from the arbiter, the microprogram informs the master device that it has control of the bus and may initiate a data transfer. After completing the transfer, the master device releases the bus-request signal, thus causing the FPC to free the bus by releasing the $\overline{\text{BBSY}}$ signal. The FPC may also relinquish control of the bus at the request of the bus arbiter.

In designing your application, you need to extract the bus-acquisition phase of the requester from the flow diagram in **Fig 2** and translate this information into a state diagram (**Fig 4**), from which you must write the corresponding FPC assembler source code. **Listing 3** gives an example of this code.

FPC-controlled data transfers

Once a master module becomes the bus master (that is, once it gains control of the bus), it can begin transferring data to a slave module. A data-transfer flow diagram (**Fig 5**) shows the necessary handshaking signals for transferring 32-bit data between master and slave. The state diagram for the master module's FPC (**Fig 4**) combines the bus-acquisition (requester) and data-transfer-control functions of the master module. An FPC assembler (which the manufacturer of the FPC provides) simplifies the task of microprogramming this state machine into an FPC.

Handling unanswered data-transfer requests

To prevent bus lockups, which malfunctioning slave devices may cause, you should implement a bus-time-out (BTO_N) option in the master module by writing a microcode routine that uses the 6-bit counter in the master's FPC. (**Listing 4** gives an example of such a routine.) The bus-time-out option permits a bus master to abort a data-transfer cycle if the slave does not respond within n microseconds.

At the start of every data-transfer operation, the FPC microprogram loads a value into the 6-bit counter, tests for the data-transfer acknowledge signal ($\overline{\text{DTACK}}$), decrements the counter and tests it for zero,

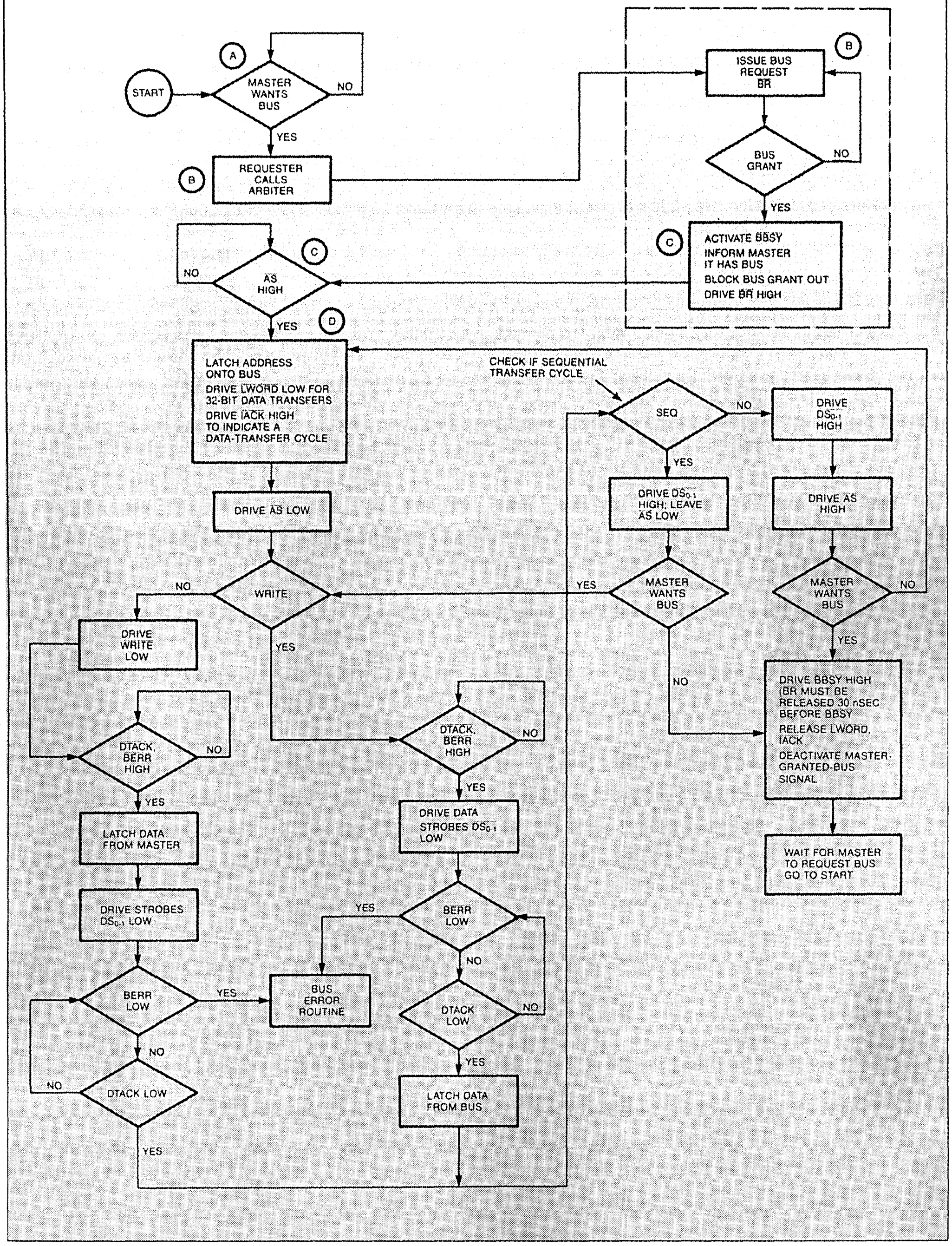


Fig 4—You can derive detailed state-machine diagrams *from your flow diagrams. In this diagram for a master requester, the bus-acquisition phase is surrounded by a dashed line. Note the close correspondence between the state diagram and the assembly-language program in **Listing 3.***

The bus-time-out option permits a bus master to abort a data-transfer cycle if a slave does not respond within a specified time.

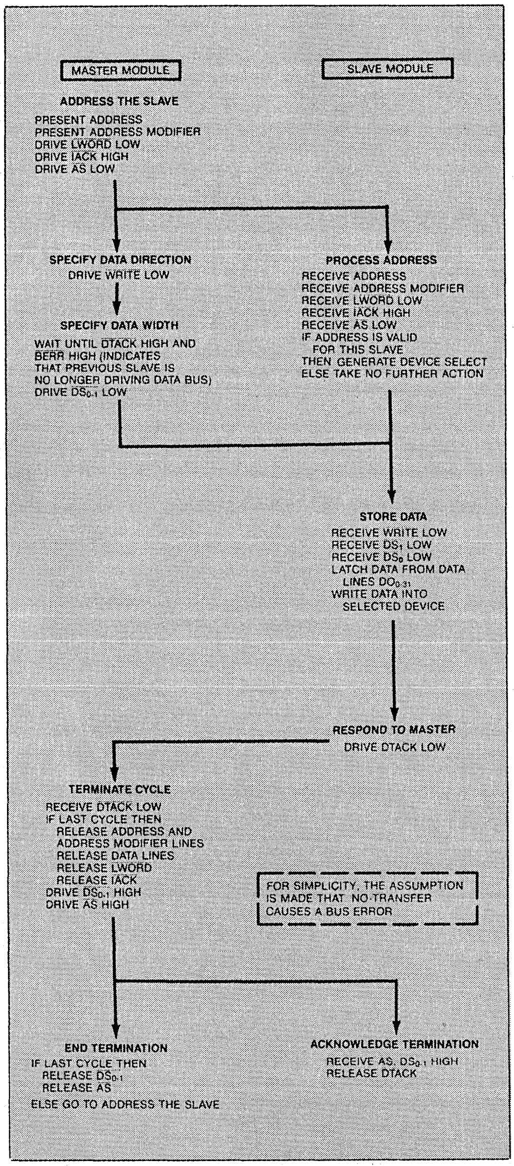


***Fig 5—This flow diagram shows the signals you'll need** to perform a 32-bit data transfer once a master has acquired control of the bus.*

LISTING 4

```
"Begin transfer : Bus time out loop contains instructions 1 and
 2. If count reg is loaded with 63 and FPC is running at 20 MHz,
 then total time before bus times out is (63+1)*2*50 ns          "
wait__loop: DS0 + DS1 , if (DTACK = 0) then                   "1"
                           goto pl(transfer__complete) ;
            DS0 + DS1 , while (creg < > 0)                    "2"
                           loop to pl(wait__loop) ;
            DS0 + DS1 , if (creg = 0) then
                           goto pl(bus__time__out)
                      :
                      :
transfer__complete:  "goto to this section if handshaking signal
                      was received"
                      :
                      :
bus__time__out: "execute this section when counter is 0"
               BTO__signal, . . . . . . . "activate bus time out signal"
```

and finally loops back to the instruction that tests $\overline{\text{DTACK}}$. As soon as $\overline{\text{DTACK}}$ is detected, the program branches to the section of code that handles a normal data-transfer operation.

If the counter value reaches zero, however, the microprogram must branch to a section of code that can handle situations in which data-transfer requests are not completed; such situations are, of course, entirely dependent on the user and the application. To calculate the time that will elapse before the program generates a bus-time-out signal, multiply the number of instructions in the time-out loop by the system cycle time by the initial value (plus 1) that the program has loaded into the count register. **EDN**

Author's biography

Arthur Khu, a product planning engineer for Advanced Micro Devices (Sunnyvale, CA), is responsible for research and definition of advanced programmable-logic-device architectures. He holds a BS in Math/Computer Science and an MS in Computer Science from Santa Clara University. In his spare time, Art enjoys racquetball and astronomy.

Designer's Guide to
VME Bus Control—Part 2

FPCs and PLDs implement VME Bus slave controllers

By using fuse-programmable controllers (FPCs) and PLDs, you can implement VME Bus control in your computer system with a minimum of hardware. Part 1 (October 2, pg 187) of this 2-part series described the bus-arbitration process and control functions and showed how to implement the VME Bus data-transfer protocol in an FPC. This second and final part describes the implementation of similar controllers for slave modules and interrupt handlers, and it provides tools for programming the FPC.

Arthur Khu, *Advanced Micro Devices*

When you offload a system's bus-control functions from the CPU to a hardware bus controller, you considerably reduce the time these functions require, thereby improving the data-transfer rate over the bus. You can substantially reduce the cost and chip count of such a bus controller by implementing the bus-control logic in VLSI devices such as fuse-programmable controllers (FPCs) and programmable logic devices (PLDs).

The VME Bus protocol requires that all data transfers over the bus be initiated by a master module. Your system can include several master modules, such as CPUs or DMA controllers; a bus-arbiter module arbitrates simultaneous data-transfer requests from two or more masters. No data transfer can take place until the requesting master has been given control of the bus by the bus arbiter.

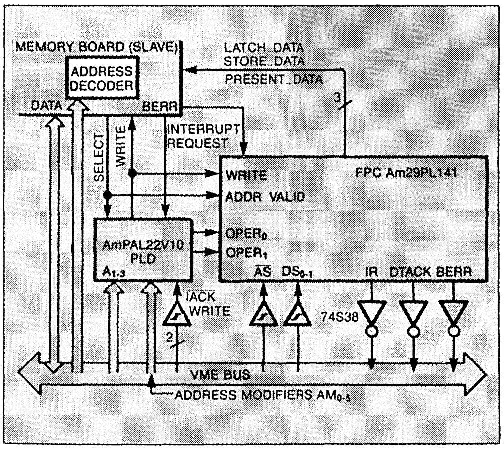


*Fig 1—**This device controller,** which is implemented with an FPC and a PLD, handles data-transfer and interrupt protocols for a slave device in a VME Bus system.*

Because a slave can't initiate a data transfer, your system must include an interrupt mechanism that allows a slave to request service from a master. You can implement such a mechanism in your system by designing a slave subsystem like the one in **Fig 1.** This slave subsystem comprises a slave module and a slave interrupt controller. The slave interrupt controller, which consists of an Am29PL141 FPC and an AmPAL22V10 PLD, serves as the interface between the slave subsys-

The interrupt controller for a slave device implements the interrupt protocol in hardware.

tem and the VME Bus structure. Once the master has initialized the slave, it can issue commands to the slave. The slave performs these tasks in the background, leaving the master free to continue its own operations. When the slave is ready to return the status or result of a task to the master, it issues an interrupt request.

The VME Bus single-cycle data-transfer protocol defines the interactions between master and slave controllers **(Fig 2)**. When the master drives the address strobe ($\overline{AS}$) low, the slave controller latches the address presented on the bus, and a separate address-decoding unit on the slave board decodes the address. If the address is not valid (ie, if it's not in the range associated with this slave), the slave takes no further action. However, if the address selects this slave subsystem,

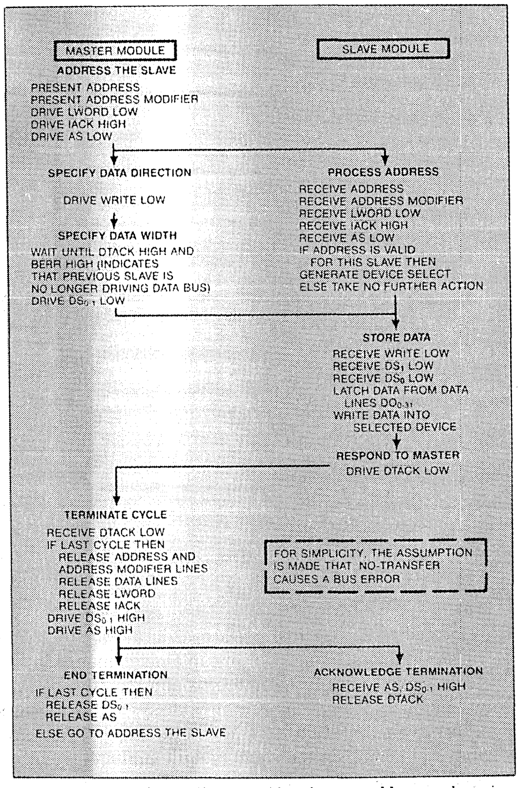
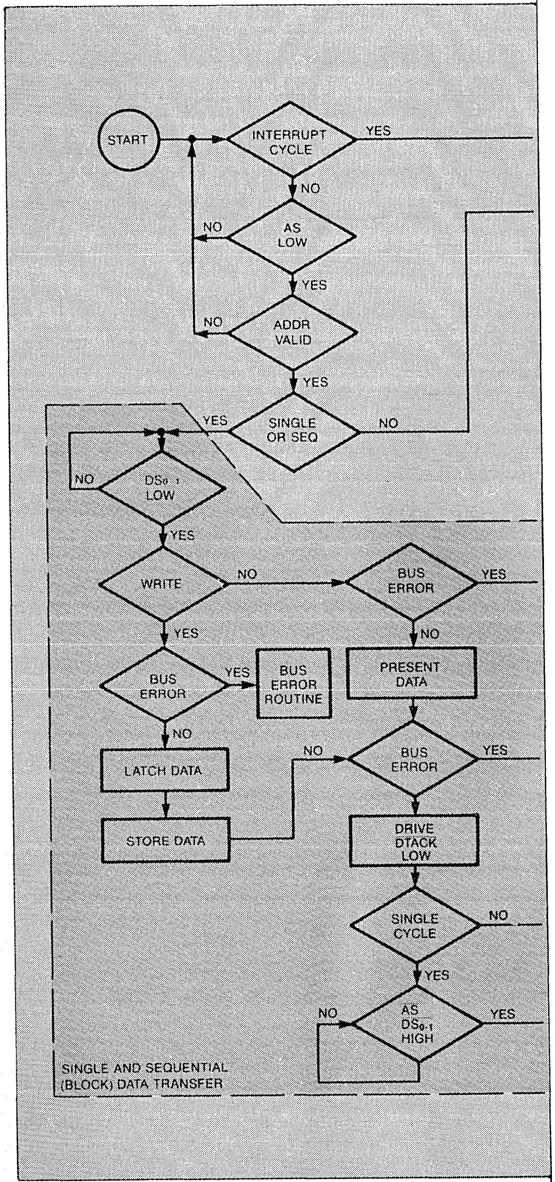


Fig 2—The slave controller recognizes its own address *and receives a data word from the bus master in this flow diagram for single-cycle transfer.*



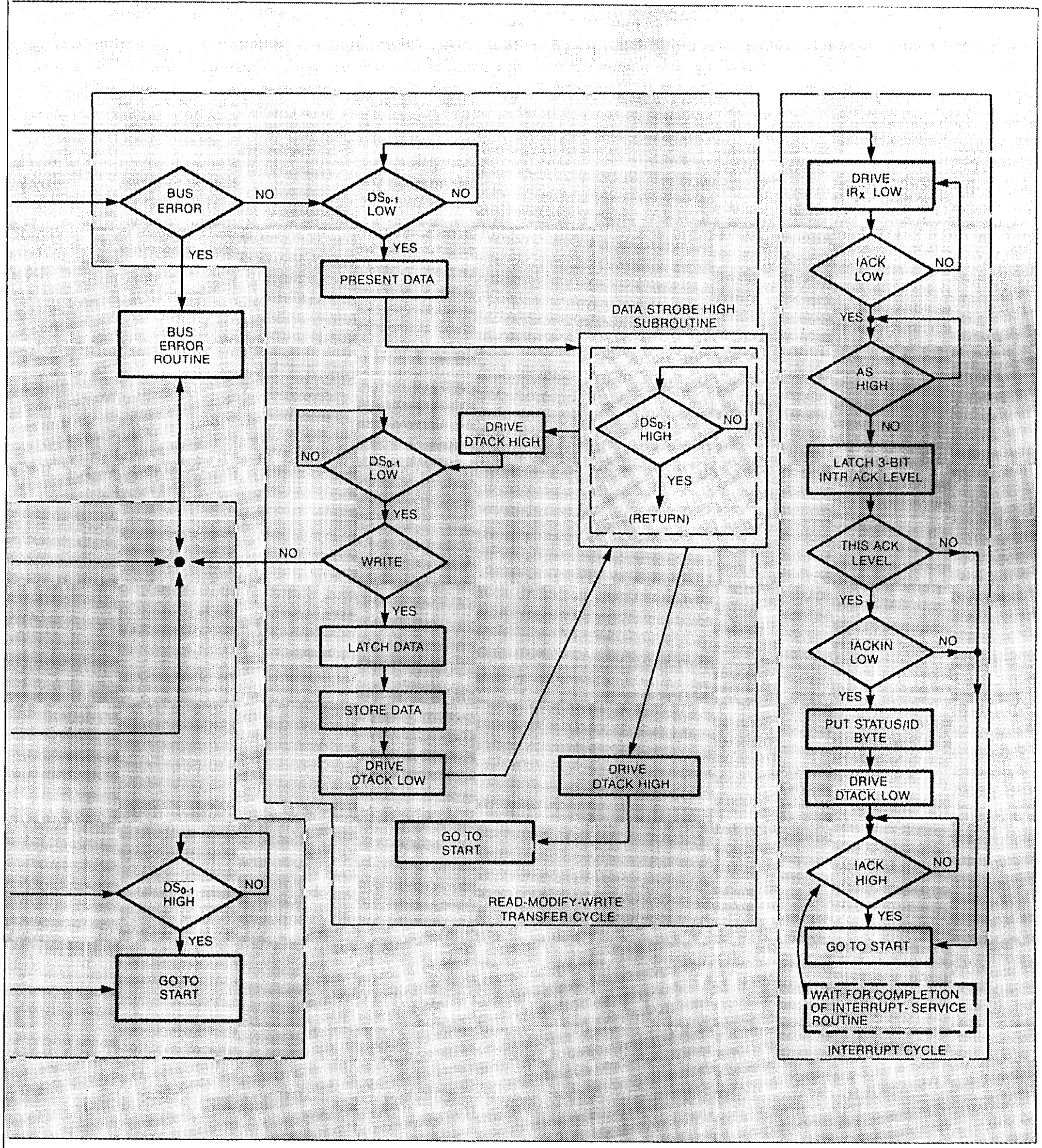


*Fig 3—**This state-machine diagram** for **Fig 1's** slave controller is derived from the flow diagram in **Fig 2**. The state machine has four modes, which handle interrupts as well as single, block, and read-modify-write data transfers.*

The VME Bus protocol requires that all data transfers over the bus be initiated by a master module.

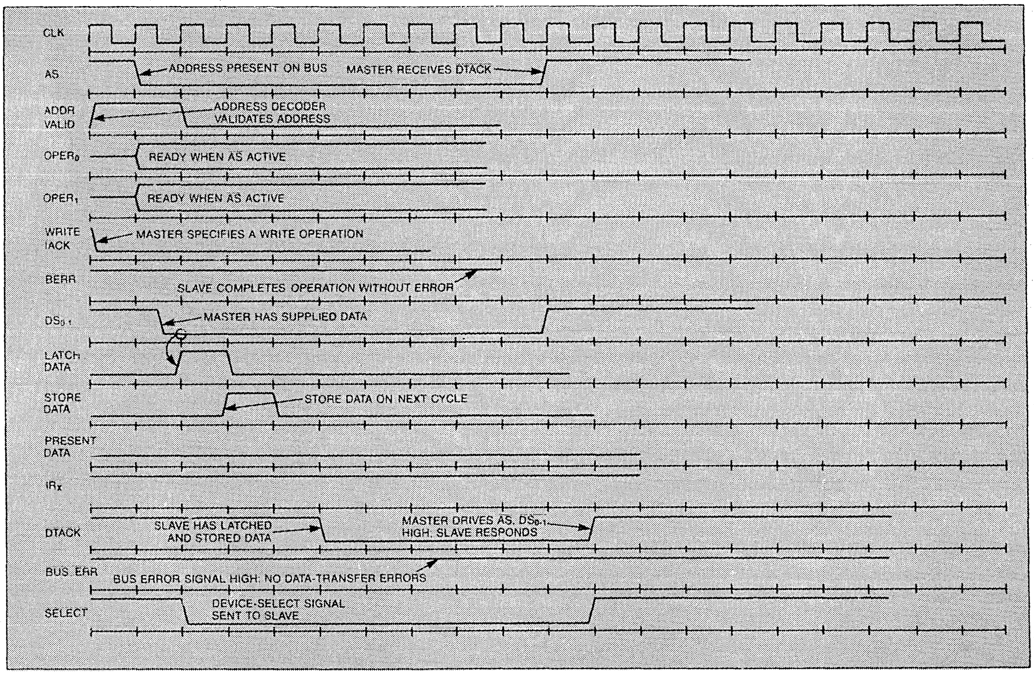
the slave looks for signals from the master that specify whether a read or a write operation should take place. After presenting or storing data, the slave controller drives the data-transfer-acknowledge signal ($\overline{DTACK}$) low to inform the master of the successful transfer.

From the VME Bus-protocol flow diagrams (**Fig 2**), you can derive a state diagram (**Fig 3**) for the slave-controller state machines; **Fig 4** shows the resulting timing pattern for the slave controller. You can develop microcode for the slave controller's single-cycle transfer mode from the state diagram and the timing pattern. Use the address-modifier lines ($AM_{0\text{-}5}$) to specify the other three slave operating modes (sequential, or block, transfers; read-modify-write transfers; and interrupt cycle). To recognize these special modes, the slave controllers on each slave board constantly monitor the six address-modifier lines.

When the code presented on the address-modifier lines specifies a block-transfer operation, the master retains control of the bus throughout the operation by holding $\overline{AS}$ and $\overline{BBSY}$ low. For a block transfer, bus arbitration takes place only once, before the start of the operation; for single-cycle transfers, bus arbitration takes place before the transfer of each word. A block transfer, therefore, takes less time than does the corresponding number of single-word transfers.

At the start of a block-transfer operation, all the slaves load the address presented on the bus into their address counters and decoders, but only the slave whose memory range encompasses the decoded address responds to the data-transfer request. As each word transfer is completed, all the slaves increment (or decrement) their address counters and decode the new address. This procedure is necessary because the memory block being transferred may reside on more than one slave memory board.

In the slave subsystem in **Fig 1**, the PLD decodes control signals from the bus and slave board and sends two signals, $OPER_0$ and $OPER_1$, to the slave's FPC. The FPC decodes these two signals, along with inputs from



CLK
AS
ADDRESS PRESENT ON BUS
MASTER RECEIVES DTACK
ADDR VALID
ADDRESS DECODER VALIDATES ADDRESS
OPER0
READY WHEN AS ACTIVE
OPER1
READY WHEN AS ACTIVE
WRITE IACK
MASTER SPECIFIES A WRITE OPERATION
BERR
SLAVE COMPLETES OPERATION WITHOUT ERROR
DS0-1
MASTER HAS SUPPLIED DATA
LATCH DATA
STORE DATA
STORE DATA ON NEXT CYCLE
PRESENT DATA
IRx
DTACK
SLAVE HAS LATCHED AND STORED DATA
MASTER DRIVES AS, DS0-1, HIGH; SLAVE RESPONDS
BUS ERR
BUS ERROR SIGNAL HIGH; NO DATA-TRANSFER ERRORS
SELECT
DEVICE-SELECT SIGNAL SENT TO SLAVE

Fig 4—You'll need to generate a timing diagram *for each of the slave controller's operating modes. This diagram shows the timing for a single-cycle data transfer.*

the slave, bus, and address-decoding units on the slave board, to determine which of the four possible slave operating modes to execute. The four modes, which are designated by binary codes, specify the following operations:

- (00) Perform a single-cycle transfer
- (01) Perform a sequential-cycle transfer
- (10) Perform a read-modify-write cycle
- (11) Perform an interrupt cycle.

If $OPER_0$ and $OPER_1$ are both high, the FPC operates as an interrupter by branching to an interrupt subroutine (**Fig 3**).

When the slave requests an interrupt, the FPC generates an interrupt-request signal and waits for the interrupt-acknowledge signal ($\overline{IACK}$) and daisy-chain signal ($\overline{IACKIN}$). When the data strobes become active, the PLD reads a 3-bit value from the address bus (A_{1-3}); this value indicates which interrupt-request line was acknowledged. The PLD decodes these three bits to determine whether their value matches its own request level; if it finds no match, no further action occurs. If it does find a match, however, the PLD routes a valid signal to the FPC, indicating that the interrupt handler has acknowledged the slave's interrupt request. The FPC then signals the slave board to put its status or identification byte on the data bus for the interrupt handler to use as an interrupt vector. The slave FPC waits in a loop until the interrupt handler drives the $\overline{IACK}$ signal high to signify that interrupt service is complete.

Besides containing master, slave, and bus-arbiter modules, a VME Bus system usually has an interrupt-handler module that handles external I/O or special system events (time-out or overflow errors, for example). You can reduce the logic complexity of the interrupt-handler module in your system by offloading some of the initial interrupt-recognition tasks to an interrupt-handling preprocessor (IHP). You can use a PLD as the IHP, programming it to preprocess interrupt requests, obtain control of the bus, and handle hand-

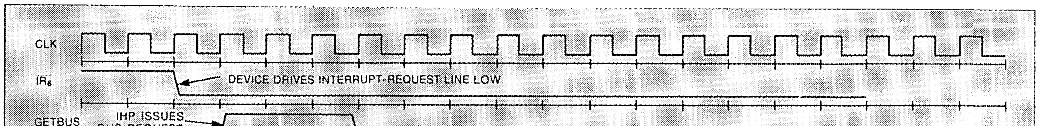

CLK
$\overline{IR_6}$
DEVICE DRIVES INTERRUPT-REQUEST LINE LOW
GETBUS
IHP ISSUES BUS REQUEST
$\overline{DTACK}$
INTERRUPTER PUTS STATUS BYTE ON BUS
$\overline{BG3OUT}$
BUS-GRANT SIGNAL IS NOT DAISY-CHAINED OUT
$\overline{BG3IN}$
$\overline{BR_3}$
$\overline{BBSY}$
SERVICE-DONE SIGNAL FORCES IHP TO RELEASE BBSY SIGNAL
INTR[2.0]
ACKNOWLEDGE INTERRUPT LEVEL 6 (110)
$\overline{IACK}$
$\overline{AS}$
INFORM INTERRUPTER THAT 3-BIT INTERRUPT-ACKNOWLEDGE LEVEL IS ON THE BUS
LATCH STATUS
LATCH STATUS/IDENTIFICATION BYTE
START SERVICE
IHP TELLS INTERRUPT HANDLER TO BEGIN INTERRUPT SERVICE
SERVICE DONE
INTERRUPT HANDLER TELLS IHP THAT INTERRUPT SERVICE IS COMPLETED
NOTE:
INTERRUPT-HANDLING PREPROCESSOR (IHP) MONITORS LINES IR_{1-7}

Fig 5—The interrupt-handling preprocessor *monitors the seven VME Bus interrupt-request lines ($\overline{IR_{1-7}}$), identifies the interrupting device, and tells the interrupt handler when to begin servicing the interrupt. This timing diagram shows the signal states that exist before and during the transfer of the identification and status bytes from the slave-interrupt subsystem to the interrupt handler.*

When the slave module is in block-transfer mode, bus arbitration takes place only once before the start of the operation.

shaking signals (such as interrupt-acknowledge signals). Only when the IHP latches the interrupt vector will control pass to the interrupt-handler module.

To define interrupt-request processing, bus acquisition, and the interrupt vector's transfer phase, you'll have to use logic equations written in high-level Boolean notation. In the design in **Fig 1**, the PLD monitors seven interrupt-request lines and four data-transfer control inputs, and it sends 10 control signals to the interrupt handler and the VME Bus drivers. The PLD monitors all interrupt-request lines according to the following logic equation:

```
IF (IR1 + IR2 + IR3 + IR4 + IR5 +          (1)
  IR6 + IR7) THEN
BR3 : = 1 ;
      "THIS INTERRUPT HANDLER USES"
      "THE BR3 REQUEST LINE"
```

If any interrupt-request line is active, the PLD asserts $\overline{\text{BR3}}$ to initiate the bus-acquisition phase.

The next step is to wait for the bus-grant-in signal. Only when $\overline{\text{BGIN3}}$ is active will $\overline{\text{BBSY}}$ be active. The logical expression of this condition is given in the following equations:

```
IF (BR3*/BG3IN) THEN                              (2)
         BR3 : = 1 ;

IF (BR3*BG3IN + BBSY*/SERVICE_DONE) THEN (3)
         BBSY : =  1 ;
```

Eq 2 continually asserts $\overline{\text{BR3}}$ as long as $\overline{\text{BG3IN}}$ is not active; **Eq 3** asserts $\overline{\text{BBSY}}$ only when request line $\overline{\text{BR3}}$ and $\overline{\text{BG3IN}}$ are active, or if service is not complete after the IHP asserts $\overline{\text{BBSY}}$.

When the IHP is the bus master, it puts the 3-bit

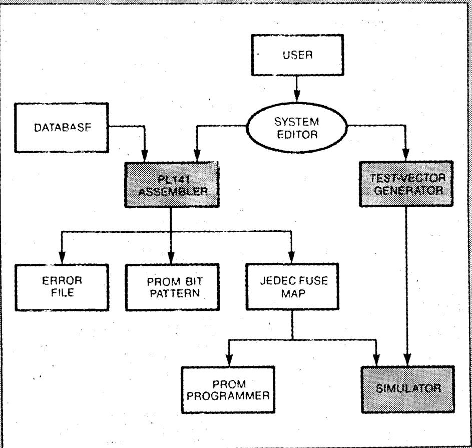
Development tools help you program FPCs

The Am29PL141 is a single-chip fuse-programmable controller (FPC) that can implement state machines and distributed control functions. You can express and functionally verify these state machines and functions by using three FPC-development tools provided by the manufacturer: an assembler, a test-vector generator, and a simulator. These development tools, which are written in C, run on an IBM PC or compatible computer under MS-DOS. Source code for the programs is available, so you can port the software to other systems.

The FPC has a control store that's resident in its PROM; the store is 64 words long and 32 bits wide. Instructions such as jumps, loops, and subroutine calls in the PROM control store are conditionally executed by the 20-MHz internal sequencer. Each 32-bit instruction is partitioned into the following format:

```
BITS:   1     5      1    3     6       16
FIELD: OE OPCODE POL TEST DATA OUTPUT
```

The output field contains the 16 control outputs that the FPC generates when it executes an instruction. The upper eight output bits are controlled by the output enable (OE) bit; they may be high, low, or disabled. The lower eight output bits are always enabled. The op-code field specifies which of the 29 possible instructions the FPC will execute if the condition selected by the test



To program the Am29PL141 FPC to implement VME Bus-control functions, *you can use the assembler, test-vector generator, and simulator provided by the FPC's manufacturer. These development tools, which are written in C, run on an IBM PC or compatible computer under MS-DOS.*

interrupt-line-acknowledge value on the bus and asserts the data strobes. Upon receipt of the $\overline{\text{DTACK}}$ signal from the interrupter, the IHP strobes the status byte from the bus into a register on the interrupt-handler card. The IHP then informs the interrupt handler that a status/identification byte is ready and that the handler should begin servicing the interrupt. The IHP resolves interrupt priorities within a single clock cycle because all interrupt/input signal lines are processed in parallel by the logic array in the PLD. **Listing 1** shows how you'd express the procedure in a logic-description language.

As you can see from **Listing 1,** if both the $\overline{\text{IR7}}$ and the $\overline{\text{BBSY}}$ signals are active in section 1, then the 3-bit value generated by the IHP will be binary 111 (decimal 7), regardless of the state of the other interrupt lines. In section 2, if $\overline{\text{IR1}}$ and $\overline{\text{BBSY}}$ are active and the other six control signals are inactive, then the 3-bit output value will be binary 001 (decimal 1). In section 2, $\overline{\text{IR1}}$ is the highest priority line that is active.

LISTING 1

```
IF  (BBSY) THEN
    BEGIN
(1) IF (IR7) THEN                 "IR7 was active                  "
        INTR[2:0] : = 7 ;         "3-bit value acknowledging
                                   interrupt line 7                "
    IF (/IR7*IR6) THEN            "IR6 active, IR7 inactive        "
        INTR[2:)] : = 6 ;         "acknowledge interrupt line 6"
    IF (/IR7*/IR6*IR5) THEN       "IR5 highest priority            "
        INTR[2:0] : = 5 ;         "interrupt line active           "
              :
              :
              :
(2) IF (/IR7*/IR6*/IR5*/IR4*/IR3*/IR2*IR1) THEN
        INTR[2:0] : = 1 :         "acknowledge interrupt line 1"
END;
```

The remaining interrupt-preprocessing steps complete the transfer of the interrupt-vector byte; this transfer is described logically in the following

```
IF  (BBSY) THEN                                                         (4)
    BEGIN
    IACK := 1;                    "begin interrupt acknowledge daisy"
(X) IF (DTACK) THEN               "chain; if device sends data       "
       LATCH. STATUS := 1;.       "transfer acknowledge (DTACK)      "
                                  "signal, then latch the status     "
    END:
IF  (IACK) THEN
    AS := 1;            "assert the address strobe signal to inform the
                         interrupter that the interrupt acknowledge
                         level is ready"
```

field is true. The polarity bit (POL) determines whether a high input or a low input represents true for the test input selected. You can use the data field as an argument for the op-code field. For example, if you want the FPC to jump (op code 25) to location 34 (22_{HEX}) when the second bit in the test field is high (true), you put the following 32-bit word into the control store (though you'll also have to define the OE bit and the output bits):

OE 11001 0 010 100010 OUTPUT

The assembler simplifies system design by allowing you to use a high-level language. Instead of specifying the bit patterns shown above, you can write the following section of code:

```
OUTPUT , IF (T2 = 1) THEN
            GOTO PL (34);
```

The assembler translates this statement into the appropriate bit patterns; when the FPC executes the instruction, it generates the bit pattern defined by the symbol OUTPUT.

Every microinstruction written in assembly language follows this format:

< LABEL : > OUTPUT , STATEMENT ;

The label field, which is optional, simplifies the writing of subroutine calls and conditional branch instructions, which you can express in IF-THEN-ELSE, WHILE-DO, or COMPARE forms. Once you've written all the instructions with the editor and translated them to executable form with the assembler, you can cause the assembler to generate a JEDEC fuse map.

Before physically programming the PROM control store that you've designed, you can test your design with the help of the test-vector generator and simulator. The test-vector generator produces test vectors, from a user-generated truth table, and converts them into a JEDEC-standard test-vector file that the simulator can use.

The simulator performs two important functions: It verifies the control flow of your design, and it checks the fuse map that the assembler generates. By using the simulator's interactive mode, you can inspect, modify, or preset any of the internal registers in the FPC. The simulator also has single-stepping and break-point features so that you can trace the operation of your design in detail.

An interrupt-handling preprocessor can offload some of the initial interrupt-recognition tasks of your system's interrupt-handler module.

Once the assertion of the $\overline{\text{DTACK}}$ signal indicates the successful transfer of the 8-bit status or identification byte, the IHP instructs the interrupt-handler module to begin the interrupt-service routine. This instruction from the IHP is logically defined as follows:

```
IF (BBSY*LATCH_STATUS + BBSY*                    (5)
  START_SERVICE*/SERVICE_DONE) THEN
START_SERVICE : = 1;
```

This definition states that the IHP constantly asserts the START_SERVICE signal until the interrupt handler generates a SERVICE_DONE signal, at which point the IHP drives START_SERVICE low. The logic described above generates the timing diagram shown in **Fig 5.**

Development tools simplify controller design

Development tools provided by FPC and PLD manufacturers simplify the task of programming these devices as VME Bus controllers. Once you've analyzed the bus protocols and converted these into state-machine diagrams, you can write assembly-language programs and high-level logic equations to describe the state machines. The assembler and logic software will then process these programs and equations to fit into the FPC or PLD. For a summary of the programming tools available for the Am29PL141 FPC, see **box,** "Development tools help you program FPCs." **EDN**

Author's biography

Arthur Khu, a product planning engineer for Advanced Micro Devices (Sunnyvale, CA), is responsible for research and definition of advanced programmable-logic-device architectures. He holds a BS in Math/Computer Science and an MS in Computer Science from Santa Clara University. In his spare time, Art enjoys racquetball and astronomy.

CHAPTER 4

COFFEE MACHINE CONTROLLER USING Am29PL141

This section is a tutorial to show designers how to go from a design requirement to Am29PL141 microcode. The coffee machine application was chosen because it is easy to understand.

The following example describes the hardware and the programming required. A flow diagram of the program is included. The assembler program for the coffee vending machine example is called COFFEE.EXP. The PL14x assembler produces two outputs, the JEDEC fuse map output file (COFFEE.JED) and the PROM bit pattern output file (COFFEE.BIT). First, the problem is defined.

The coffee machine controller waits for a coin before dispensing the beverage selected by the customer. The choices are indicated as combinations of buttons.

Design requirement:

Design a coffee machine controller that works as follows:

1. Do nothing until a coin is detected.
2. On coin detection turn on busy light and wait for selection:

   i. coffee
   ii. chocolate
   iii. soup
   iv. coin return

3. If coin return is detected, return coin, turn off busy light and wait for next coin.
4. If coffee, chocolate or soup is detected, drop a cup.
5. The cup has 1.5 seconds to get into place.
6. Turn on water for 1 second prior to release of powders.
7. Water will remain on continuously for a total of 10 seconds.
8. Busy light will remain on until end of sequence.
9. Depending on selection, either coffee, soup or chocolate will be dispensed:

coffee	2.5 seconds
soup	2.0 seconds
chocolate	3.5 seconds

10. If coffee was selected, check to see if cream and/or sugar are selected. If yes, cream 2.0 seconds, sugar 1.5 seconds.
11. After water has completed filling the cup, allow 3.5 seconds for cup removal before testing for presence of next coin.
12. Clock rate is 10 Hz.

As can be seen, there are six possible beverages:

i. coffee black
ii. coffee with sugar
iii. coffee with cream
iv. coffee with cream and sugar
v. chocolate
vi. soup

The conditions that need to be tested are:

i. coin drop
ii. coffee
iii. cream
iv. sugar
v. chocolate
vi. soup
vii. return (coin return)

Control signals that need to be generated from the controller are:

i. busy light on (busy)
ii. cup drop (cup)
iii. water on (water)
iv. coffee on (coffee)
v. cream on (cream)
vi. sugar on (sugar)
vii. chocolate on (choclat)
viii. soup on (soup)
ix. coin return (coin_return)
x. clear inputs (clr_inp)

Figure 4-1 represents the hardware required for the controller. The inputs need to be synchronized and latched, hence the PAL device (16R8). Once latched, the clr_inp signal from the Am29PL141 clears the external registers within the PAL at the end of each sequence. The Am29PL141 has seven external test inputs. These are used to test the seven conditions. Since all but one of the Am29PL141 instructions are conditional, unconditional jumps must be implemented by a 'forced pass'. The 'EQ' flag internal to the Am29PL141 is a test condition not being used in this design. It can therefore be used to allow 'unconditional' instructions. The state of the 'EQ' flag is always known since it is unused for any other purpose. (The 'EQ' flag is cleared on reset.)

Figure 4-2, the flow diagram for the program, describes the logical flow of events required by the design. The rectangular boxes in the flow diagram show the value of the control field for that state. The diamond shaped boxes imply a conditional test to decide the next state. A pair of rectangular and diamond shaped boxes indicate a conditional microcode line. A rectangular box not followed by a diamond shaped box implies that the instruction is a continue or an unconditional branch.

The Am29PL141 is used to develop the microcode. Figure 4-3 is a listing of the assembler source code used. It is assumed that the reader is familiar with the PL14x assembler supplied by Advanced Micro Devices. Note that all timing is in 0.5 second increments. At 10 Hz, 0.5 second corresponds to 5 clocks.

Each box in the flow diagram can be directly translated into one or more lines of microcode. One important convention needs to be remembered. Each microcode line specifies the state of the control outputs and the branch address for the next instruction. Hence in the flow diagram, the decision box follows the output field box. The flow diagram indicates the microcode line numbers corresponding to each box.

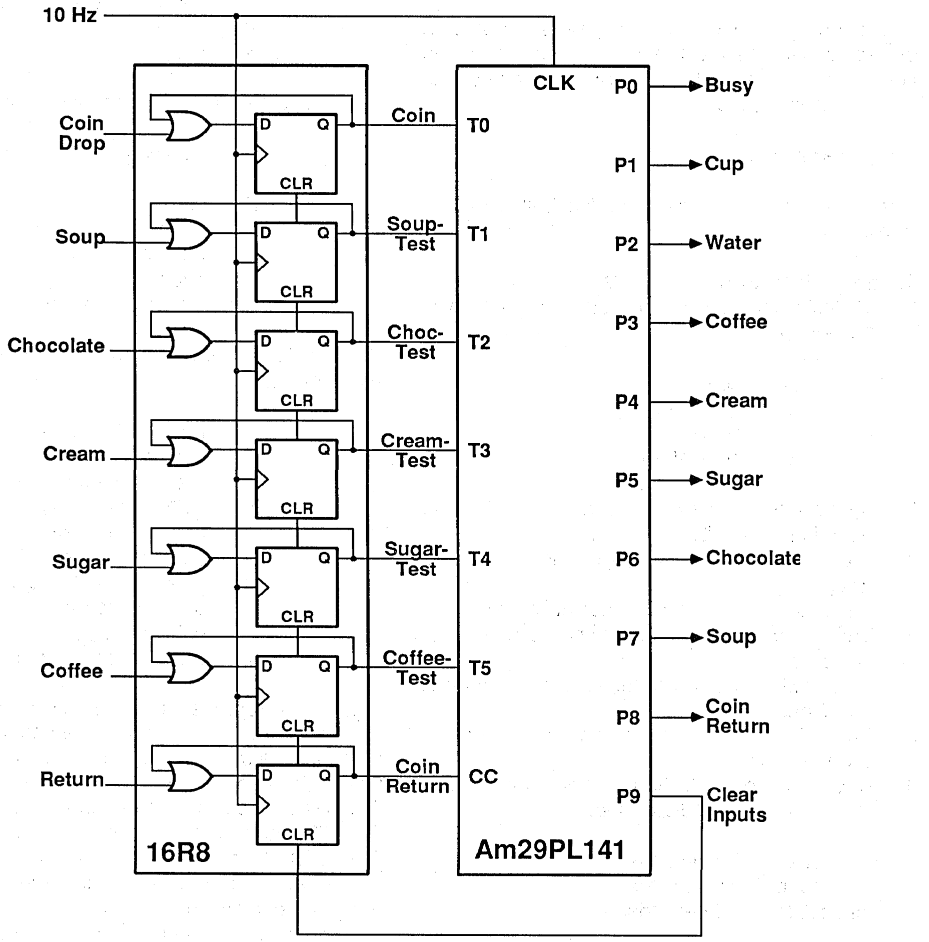
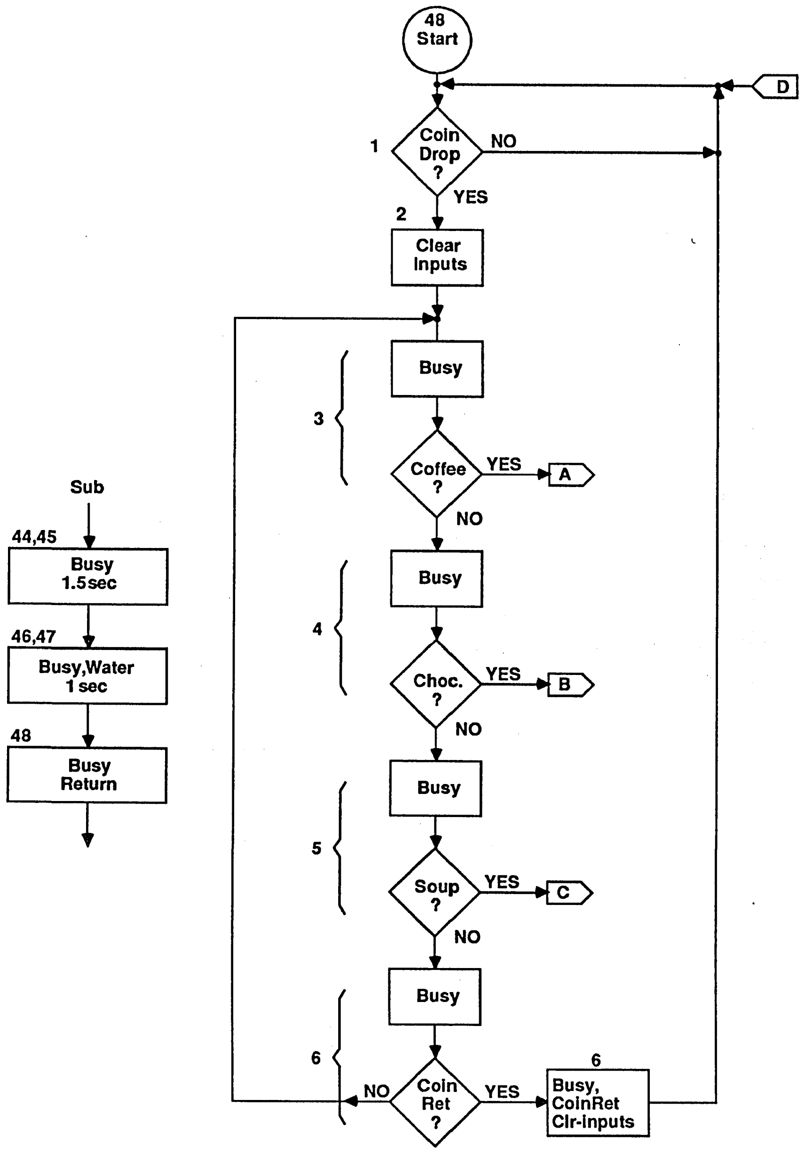
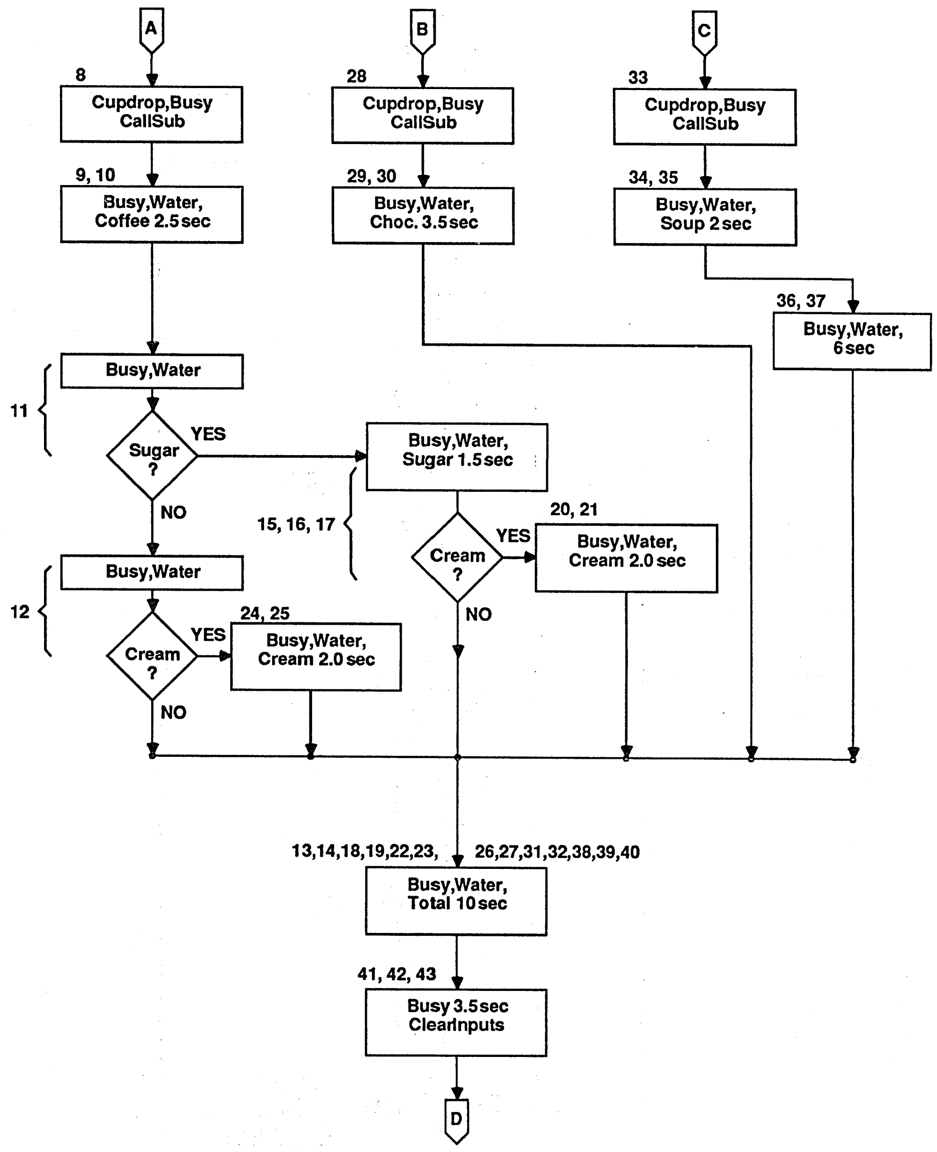


Figure 4-1. Coffee Machine Hardware



06591A 4-2

Figure 4-2. Coffee Machine Program Flow Diagram (Sheet 1 of 2)



06591A 4-2b

Figure 4-2. Coffee Machine Program Flow Diagram (Sheet 2 of 2)

Special care needs to be taken to ensure that water is on continuously for 10 seconds. Six possible paths lead to the microcode line labeled "last". At the end of each of these paths, the CREG is loaded with a value equal to 10 seconds minus the time in seconds for which water has already been on. Note that the value loaded into the CREG is one less than the expected value. This is because the value 0 in the CREG needs to be accounted for as the Am29PL141 checks the CREG and then decrements.

For example, coffee needs to be turned on for 2.5 seconds if selected. At 10 Hz, this translates into 25 clock periods. Coffee is on for one clock period during the instruction when the counter (CREG) is loaded with a countdown value (line 9 of the microcode). The counter therefore needs to be loaded with a countdown value of 23 which corresponds to coffee being on for 24 clock periods before the counter counts down to zero. The total time for which coffee is on is therefore 24 + 1 = 25 clock periods or 2.5 seconds.

On reset, the 'EQ' flag is cleared. For a 'pass' to occur, the flag must, therefore, be tested for a '0'. Hence the 'not fail' in each of the unconditional microcode lines instead of the more obvious 'pass'. Also on reset, the Am29PL141 executes the instruction on line 63. In this example, this line is an unconditional branch to line 1 of microcode. This is a wasted microcode line. If efficient coding is required to preserve microcode lines, the coin test on line 1 of the microcode could be placed on line 63 thus saving one line of microcode.

To assemble this file, type:

```
A> ASM14x -i COFFEE.EXP -o COFFEE.JED -b COFFEE.BIT
```

When the file COFFEE.EXP is assembled, two output files are created, COFFEE.JED and COFFEE.BIT. The JEDEC fuse map output is sent to the file COFFEE.JED (Figure 4-4). The PROM bit pattern is sent to the file COFFEE.BIT. See Figure 4-5 for a listing of this file.

```
DEVICE (PL141)

DEFAULT = 1 ;

DEFINE "test inputs are given name assignments"

       coin = t0
       soup_test = t1
       choc_test = t2
       cream_test = t3
       sugar_test = t4
       coffee_test = t5
       coin_ret = cc
       fail = eq

       "output/control bits are given name assignments"

       off = 0#h
       busy = 01#h
       cup = 02#h
       water = 04#h
       coffee = 08#h
       cream = 10#h
       sugar = 20#h
       choclat = 40#h
       soup = 80#h
       cn_ret = 100#h
       clr_inp = 200#h;

BEGIN
"wait for a coin to drop and check selection after coin detect"
"1" zero:off,                    if (not coin) then goto pl(zero);
"2"      clr_inp,                continue;
"3" test:busy,                   if (coffee_test) then goto pl(cofe);
"4"      busy,                   if (choc_test) then goto pl(choc);
"5"      busy,                   if (soup_test) then goto pl(sup);
"6"      busy,                   if (not coin_ret) then goto pl(test);
"7"      busy + cn_ret + clr_inp,if (not fail) then goto pl(zero);
```

Figure 4-3. Coffee Machine Source Program Listing (Sheet 1 of 2)

```
"routine for coffee. Check for sugar &/or cream"
"8" cofe:busy + cup,              if (not fail) then call pl(sub);
"9"      busy + coffee + water,   if (not fail) then load pl(23);
"10"sty2:busy + coffee + water,   while (creg <> 0) loop to pl(sty2);
"11"     busy + water,            if (sugar_test) then goto pl(sugr);
"12"     busy + water,            if (cream_test) then goto pl(crem);
"13"     busy + water,            if (not fail) then load pl(60);
"14"     busy + water,            if (not fail) then goto pl(last);

"routine for sugar"
"15"sugr:busy + sugar + water,    if (not fail) then load pl(12);
"16"sty4:busy + sugar + water,    while(creg <> 0) loop to pl(sty4);
"17"     busy + sugar + water,    if (cream_test) then goto pl(crm2);
"18"     busy + water,            if (not fail) then load pl(46);
"19"     busy + water,            if (not fail) then goto pl(last);

"routine for cream if sugar is selected"
"20"crm2:busy + cream + water,    if (not fail) then load pl(18);
"21"sty5:busy + cream + water,    while (creg <> 0) loop to pl(sty5);
"22"     busy + water,            if (not fail) then load pl(26);
"23"     busy + water,            if (not fail) then goto pl(last);

"routine for cream if sugar is not selected"
"24"crem:busy + cream + water,    if (not fail) then load pl(18);
"25"sty6:busy + cream + water,    while (creg <> 0) loop to pl(sty6);
"26"     busy + water,            if (not fail) then load pl(40);
"27"     busy + water,            if (not fail) then goto pl(last);

"routine for dispensing chocolate"
"28"choc:busy + cup,              if (not fail) then call pl(sub);
"29"     busy + choclat + water,  if (not fail) then load pl(33);
"30"sty7:busy + choclat + water,  while (creg <> 0) loop to pl(sty7);
"31"     busy + water,            if (not fail) then load pl(52);
"32"     busy + water,            if (not fail) then goto pl(last);

"routine for dispensing soup"
"33"sup: busy + cup,              if (not fail) then call pl(sub);
"34"     busy + soup + water,     if (not fail) then load pl(18);
"35"sty8:busy + soup + water,     while (creg <> 0) loop to pl(sty8);
"36"     busy + water,            if (not fail) then load pl(58);
"37"sty9:busy + water,            while (creg <> 0) loop to pl(sty9);
"38"     busy + water,            if (not fail) then load pl(07);
"39"     busy + water,            if (not fail) then goto pl(last);

"routine for finishing 10 sec water and wait for cup removal"
"40"last:busy + water,            while (creg <> 0) loop to pl(last);
"41"     busy,                    if (not fail) then load pl(32);
"42"sty3:busy,                    while (creg <> 0) loop to pl(sty3);
"43"     busy + clr_inp,          if (not fail) then goto pl(zero);

"routine for waiting for cup to be in place and 1 sec. water"
"44"sub: busy,                    if (not fail) then load pl(13);
"45"stay:busy,                    while (creg <> 0) loop to pl(stay);
"46"     busy + water,            if (not fail) then load pl(7);
"47"sty1:busy + water,            while (creg <> 0) loop to pl(sty1);
"48"     busy + water,            if (not fail) then ret;

        .org 63#d

"49"     clr_inp,                 if(not fail) then goto pl(zero);
end.
```

Figure 4-3. Coffee Machine Source Program Listing (Sheet 2 of 2)

```
F0*
L0000  0 00110 0 111 111111 1111111111111111 *
L0032  0 10010 0 000 000000 1111110111111111 *
L0064  0 00110 1 010 111000 1111111111111110 *
L0096  0 00110 1 101 100100 1111111111111110 *
L0128  0 00110 1 110 011111 1111111111111110 *
L0160  0 00110 0 001 111101 1111111111111110 *
L0192  0 00110 0 000 111111 1111110011111110 *
L0224  0 00011 0 000 010100 1111111111111100 *
L0256  0 11011 0 000 101000 1111111111110010 *
L0288  0 10111 0 000 110110 1111111111110010 *
L0320  0 00110 1 011 110001 1111111111111010 *
L0352  0 00110 1 100 101000 1111111111111010 *
L0384  0 11011 0 000 000011 1111111111111010 *
L0416  0 00110 0 000 011000 1111111111111010 *
L0448  0 11011 0 000 110011 1111111111011010 *
L0480  0 10111 0 000 110000 1111111111011010 *
L0512  0 00110 1 100 101100 1111111111011010 *
L0544  0 11011 0 000 010001 1111111111111010 *
L0576  0 00110 0 000 011000 1111111111111010 *
L0608  0 11011 0 000 101101 1111111111101010 *
L0640  0 10111 0 000 101011 1111111111101010 *
L0672  0 11011 0 000 100101 1111111111111010 *
L0704  0 00110 0 000 011000 1111111111111010 *
L0736  0 11011 0 000 101101 1111111111101010 *
L0768  0 10111 0 000 100111 1111111111101010 *
L0800  0 11011 0 000 010111 1111111111111010 *
L0832  0 00110 0 000 011000 1111111111111010 *
L0864  0 00011 0 000 010100 1111111111111100 *
L0896  0 11011 0 000 011110 1111111110111010 *
L0928  0 10111 0 000 100010 1111111110111010 *
L0960  0 11011 0 000 001011 1111111111111010 *
L0992  0 00110 0 000 011000 1111111111111010 *
L1024  0 00011 0 000 010100 1111111111111100 *
L1056  0 11011 0 000 101101 1111111101111010 *
L1088  0 10111 0 000 011101 1111111101111010 *
L1120  0 11011 0 000 000101 1111111111111010 *
L1152  0 10111 0 000 011011 1111111111111010 *
L1184  0 11011 0 000 111000 1111111111111010 *
L1216  0 00110 0 000 011000 1111111111111010 *
L1248  0 10111 0 000 011000 1111111111111010 *
L1280  0 11011 0 000 011111 1111111111111110 *
L1312  0 10111 0 000 010110 1111111111111110 *
L1344  0 00110 0 000 111111 1111110111111110 *
L1376  0 11011 0 000 110010 1111111111111110 *
L1408  0 10111 0 000 010011 1111111111111110 *
L1440  0 11011 0 000 111000 1111111111111010 *
L1472  0 10111 0 000 010001 1111111111111010 *
L1504  0 11101 0 000 000000 1111111111111010 *
L2016  0 00110 0 000 111111 1111110111111111 *
C6706*
9F58
```

Figure 4-4. JEDEC Fuse MAP for Coffee Machine Program

```
hex <dec>     OE  OPCODE POL  TEST   DATA          OUTPUT
000 <  0>   [ 1 | 11001 | 1 | 000 | 000000 | 0000000000000000 ]
001 <  1>   [ 1 | 01101 | 1 | 111 | 111111 | 0000001000000000 ]
002 <  2>   [ 1 | 11001 | 0 | 101 | 000111 | 0000000000000001 ]
003 <  3>   [ 1 | 11001 | 0 | 010 | 011011 | 0000000000000001 ]
004 <  4>   [ 1 | 11001 | 0 | 001 | 100000 | 0000000000000001 ]
005 <  5>   [ 1 | 11001 | 1 | 110 | 000010 | 0000000000000001 ]
006 <  6>   [ 1 | 11001 | 1 | 111 | 000000 | 0000001100000001 ]
007 <  7>   [ 1 | 11100 | 1 | 111 | 101011 | 0000000000000011 ]
008 <  8>   [ 1 | 00100 | 1 | 111 | 010111 | 0000000000001101 ]
009 <  9>   [ 1 | 01000 | 1 | 111 | 001001 | 0000000000001101 ]
00A < 10>   [ 1 | 11001 | 0 | 100 | 001110 | 0000000000000101 ]
00B < 11>   [ 1 | 11001 | 0 | 011 | 010111 | 0000000000000101 ]
00C < 12>   [ 1 | 00100 | 1 | 111 | 111100 | 0000000000000101 ]
00D < 13>   [ 1 | 11001 | 1 | 111 | 100111 | 0000000000000101 ]
00E < 14>   [ 1 | 00100 | 1 | 111 | 001100 | 0000000000100101 ]
00F < 15>   [ 1 | 01000 | 1 | 111 | 001111 | 0000000000100101 ]
010 < 16>   [ 1 | 11001 | 0 | 011 | 010011 | 0000000000100101 ]
011 < 17>   [ 1 | 00100 | 1 | 111 | 101110 | 0000000000000101 ]
012 < 18>   [ 1 | 11001 | 1 | 111 | 100111 | 0000000000000101 ]
013 < 19>   [ 1 | 00100 | 1 | 111 | 010010 | 0000000000010101 ]
014 < 20>   [ 1 | 01000 | 1 | 111 | 010100 | 0000000000010101 ]
015 < 21>   [ 1 | 00100 | 1 | 111 | 011010 | 0000000000000101 ]
016 < 22>   [ 1 | 11001 | 1 | 111 | 100111 | 0000000000000101 ]
017 < 23>   [ 1 | 00100 | 1 | 111 | 010010 | 0000000000010101 ]
018 < 24>   [ 1 | 01000 | 1 | 111 | 011000 | 0000000000010101 ]
019 < 25>   [ 1 | 00100 | 1 | 111 | 101000 | 0000000000000101 ]
01A < 26>   [ 1 | 11001 | 1 | 111 | 100111 | 0000000000000101 ]
01B < 27>   [ 1 | 11100 | 1 | 111 | 101011 | 0000000000000011 ]
01C < 28>   [ 1 | 00100 | 1 | 111 | 100001 | 0000000001000101 ]
01D < 29>   [ 1 | 01000 | 1 | 111 | 011101 | 0000000001000101 ]
01E < 30>   [ 1 | 00100 | 1 | 111 | 110100 | 0000000000000101 ]
01F < 31>   [ 1 | 11001 | 1 | 111 | 100111 | 0000000000000101 ]
020 < 32>   [ 1 | 11100 | 1 | 111 | 101011 | 0000000000000011 ]
021 < 33>   [ 1 | 00100 | 1 | 111 | 010010 | 0000000010000101 ]
022 < 34>   [ 1 | 01000 | 1 | 111 | 100010 | 0000000010000101 ]
023 < 35>   [ 1 | 00100 | 1 | 111 | 111010 | 0000000000000101 ]
024 < 36>   [ 1 | 01000 | 1 | 111 | 100100 | 0000000000000101 ]
025 < 37>   [ 1 | 00100 | 1 | 111 | 000111 | 0000000000000101 ]
026 < 38>   [ 1 | 11001 | 1 | 111 | 100111 | 0000000000000101 ]
027 < 39>   [ 1 | 01000 | 1 | 111 | 100111 | 0000000000000101 ]
028 < 40>   [ 1 | 00100 | 1 | 111 | 100000 | 0000000000000001 ]
029 < 41>   [ 1 | 01000 | 1 | 111 | 101001 | 0000000000000001 ]
02A < 42>   [ 1 | 11001 | 1 | 111 | 000000 | 0000001000000001 ]
02B < 43>   [ 1 | 00100 | 1 | 111 | 001101 | 0000000000000001 ]
02C < 44>   [ 1 | 01000 | 1 | 111 | 101100 | 0000000000000001 ]
02D < 45>   [ 1 | 00100 | 1 | 111 | 000111 | 0000000000000101 ]
02E < 46>   [ 1 | 01000 | 1 | 111 | 101110 | 0000000000000101 ]
02F < 47>   [ 1 | 00010 | 1 | 111 | 111111 | 0000000000000101 ]
03F < 63>   [ 1 | 11001 | 1 | 111 | 000000 | 0000001000000000 ]

Where:
     Oe        = Synchronous output enable for P[15:8]
     OPCODE    = Five-bit field for selecting one of the 29
                    microinstructions
     POL       = Test condition polarity select field
                    0 = Test for true (HIGH) condition
                    1 = Test for false (LOW) condition
     TEST      = Binary value of input line to be tested
          Value     Input condition  Value     Input Condition
          000       T0               100       T4
          001       T1               101       T5
          010       T2               110       CC
          011       T3               111       EQ
     DATA      = 6-bit conditional branch microaddress, test input mask,
                    or counter value field designated as PL in
                    microinstruction mnemonics (P[21:16])
     Output    = 16-bit user output control signals (P[15:0])
```

Figure 4-5. PROM File for Coffee Machine Application

CHAPTER 5

DEC PDP-11 UNIBUS CONTROLLER

5.1 THE DESIGN PROBLEM

This paper discusses the use of the Am29PL141 Fuse Programmable Controller (FPC) as a DEC PDP-11 Unibus interface controller.

Designing an interface for the Unibus is typical of the problems which can be readily solved using the Am29PL141 FPC. The complexity of Unibus handshaking is such that microprogramming is a reasonable design technique, but use of a separate sequencer, control memory, and pipeline register is not economical. Since the FPC contains a sequencer, memory, and pipeline, it fits this class of problem rather well. The PDP-11 was chosen for this example because it has a well documented protocol which is familiar to many engineers. An overview of the Unibus is included.

The problem this application note solves is to:

Design an interface between the Unibus and a generic I/O device to allow the following operations:

- Interface to handle all Unibus protocol for
  - DATI/DATO with device as slave
  - Device BR (interrupt)
  - Device NPR (direct memory access)
  - DATI/DATO with device as master
- Interface to handle synchronous parallel transfers with device

5.2 DEC UNIBUS OVERVIEW

The DEC PDP-11 Unibus is an asynchronous bus which supports programmed I/O, prioritized interrupts, and Direct Memory Access (DMA) in a memory mapped I/O environment. All bus transfers are between a bus master and bus slave, and are controlled by the master. A bus arbitrator grants bus mastership to requesting devices.

The six basic types of transfers allowed are:

DATO	–	word data transfer from master to slave
DATOB	–	byte data transfer from master to slave
DATI	–	word data transfer from slave to master
DATIP	–	word data transfer slave to master, inhibit restore cycle
NPR	–	Non Processor Request. DMA device wants to become bus master.
BRi	–	Bus Request. Interrupt request at level i (4,5,6,or 7).

The following control signals are used during transfers:

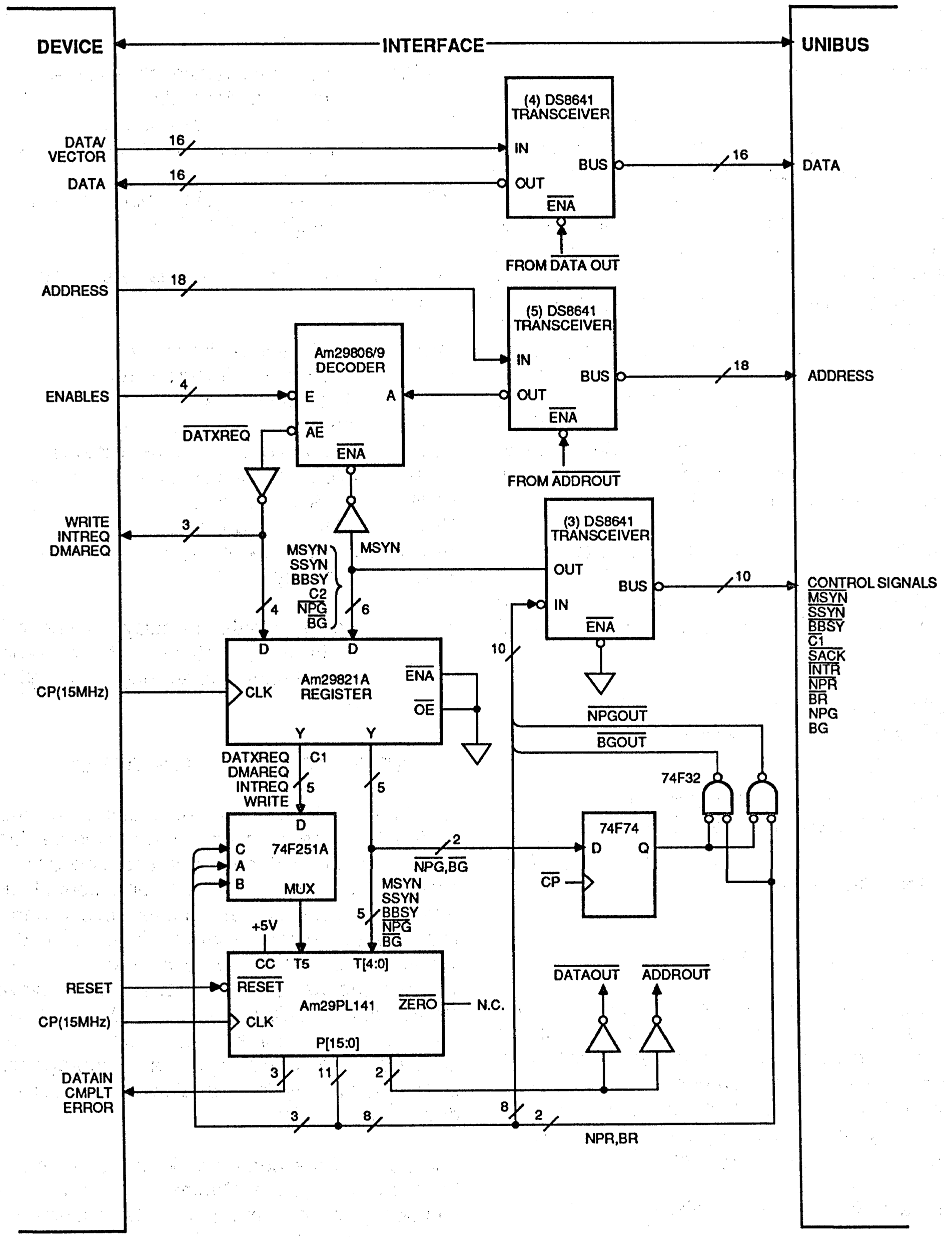
MSYN	master sync—timing control
SSYN	slave sync—timing control
C0, C1	data transfer type
BRi	interrupt bus request level i
BGi	interrupt bus grant level i (note 1)
INTR	interrupt vector strobe
NPR	DMA bus request
NPG	DMA bus grant (note 1)
SACK	select acknowledge
BBSY	bus busy

Note 1: These signals are daisy chained to form a physical priority level at each separate logical priority level (npg, bg4, bg5, bg6, bg7).

5.3 INTERFACE HARDWARE DESIGN

As shown in Figure 5-1, the architecture chosen for this interface consists of three main sections—Unibus signal buffering, address decoding, and control logic. Data, address, and control signal buffers provide proper Unibus levels and are implemented using DS8641 Quad Unified Bus Transceivers. The address decoder detects whether the device is addressed as a slave or master during Unibus DATI and DATO transfers, and is best implemented using Am29806 decoders. The control logic is a microprogrammed state machine which handles both Unibus and device handshaking.

The heart of the control logic is the Am29PL141 Fuse Programmable Controller. Its user-defined microprogram implements a state machine which handles both device and Unibus handshaking. Test inputs are synchronized with the FPC clock using an AM29821A 10-bit register. Five of these inputs go directly to the FPC, while the other five go through a multiplexer which expands the FPC conditional test capability from seven to fourteen signals. Two D flip-flops and OR gates are used to



06591A 5-1

Figure 5-1. Unibus Interface Block Diagram

implement the Unibus request/grant handshaking. Because the clock period must be at least 64.5 ns, a clock frequency of 15 MHz is appropriate (66.6 ns). A detailed control logic timing analysis is shown in Figure 5-2.

5.4 MICROWORD FORMAT

The microword organization for this application of the FPC is shown in Figure 5-3. The 32-bit microword is subdivided into fields of various sizes and functions. The 16 most significant bits are used during next address generation within the FPC, while the lower 16 bits are tailored to the application.

OE is a synchronous output enable for output bits 15 through 8. The 5-bit OPCODE field contains the FPC next address instruction.

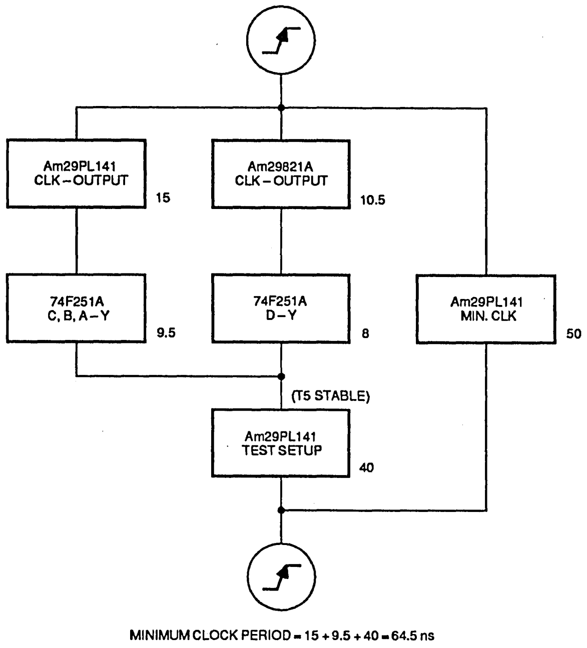
POL controls polarity of the test condition selected by the 3-bit TEST field.

DATA is a 6-bit address, test mask, or counter value; depending on the OPCODE used.

ERROR is an interface timeout indication to the peripheral device.

AUX TEST is a 3-bit field which controls the external multiplexer for additional test inputs. The TEST field must have a value of 5 to use the test selected by AUX TEST.

The 12 COMMAND outputs are single bit control signals. ADDROUT and DATAOUT enable Unibus address and data buffers. DATAIN clocks Unibus data into peripheral device registers. COMPLT indicates to the device that an interrupt or DMA operation has been completed. The remaining outputs are Unibus control signals described in Section 5.2.



06591A 5-2

Figure 5-2. Control Logic Timing

Microword Format:

```
: 31 :  30 - 26  :  25 : 24,23,22 : 21 - 16 :    15  : 14,13,12 :  11 - 0   :
:....:...........:.....:..........:.........:........:..........:............:
: oe :   opcode  : pol :   test   :   data  : error  : aux tst  :  command   :
:....:...........:.....:..........:.........:........:..........:............:

oe: output enable
(31)

opcode:       29PL141 command
(30-26)       00 - RETPL     08 - LPPL      10 - CMP       18 - FORK
              01 - RETPLN    09 - DEC       11 - CMP       19 - GOTOPL
              02 - RET       0A - LPPLN     12 - CMP       1A - WAIT
              03 - RETN      0B - GOTOPLZ   13 - CMP       1B - DECGO/C
              04 - LDPL      0C - DECAL     14 - PSHPL     1C - CALPL
              05 - LDPLN     0D - CONT      15 - PSH       1D - CALPLN
              06 - LDTM      0E - DECTM     16 - PSHTM     1E - CALTM
              07 - LDTMN     0F - GOTOTM    17 - PSHN      1F - CALTMN

pol:          test polarity ( 1 = negate )
(25)

test:         conditional test input select
(24,23,22)         0 - msyn      4 - npg
                   1 - ssyn      5 - aux tests
                   2 - bbsy      6 - pass
                   3 - bg        7 - equal flag

data:         branch address, test input mask, or counter load value
(21-16)

error:        timeout error indication to device
(15)

aux test:     additional test inputs when test = 5
(14,13,12)         0 - datxreq  4 - cl
                   1 - dmareq   5 - spare
                   2 - intreq   6 - spare
                   3 - write    7 - spare

command:
(11-0)

:  11  :  10  :   9  :  8  :  7  :   6  :  5 :  4  :   3  :   2  :   1  :  0   :
:......:......:......:.....:.....:......:....:.....:......:......:......:......:
: addr : data : data : com : cl  : intr : br : npr : sack : bbsy : ssyn : msyn :
: out  : out  : in   : plt :     :      :    :     :      :      :      :      :
:......:......:......:.....:.....:......:....:.....:......:......:......:......:
```

06591A 5-3

Figure 5-3. Microword Organization

5.5 UNIBUS CONTROLLER MICROCODE

Two things always happen during execution of a microinstruction—the address of the next microinstruction is determined using the OPCODE, POLARITY, TEST, and DATA fields; while concurrently, the Unibus and device interfaces are controlled by signals from the COMMAND field.

The microcode which controls the FPC was written using the Am29PL141 assembler available from AMD. The mnemonics used in the source code are shown in Figure 5-4. Note that these definitions are consistent with the microword definition of Figure 5-3. Figure 5-4 also contains the source code for the FPC. Figure 5-5 shows the FPC PROM contents. Note that one line of source generates one PROM word. The general source format is:

```
<label>: <outputs>, <FPC
         instruction>; "comment"
```

Outputs may be either mnemonic or constants, and may be logically "ANDed" or "ORed" together. The FPC assembler instructions are included in Chapter 2. The following paragraphs describe the code written for this FPC application. It is helpful to refer to the microcode source program listing (Figure 5-4) and the timing diagrams (Figures 5-6, 5-7, 5-8, and 5-9).

After reset to address 63, the program branches to address 0 (label TOP) and loops until one of the external conditions DATXREQ, DMAREQ, or INTREQ is asserted. For example, at TOP, if auxiliary test condition DATXREQ is asserted, the subroutine DATX is called. Otherwise, the next sequential instruction is executed.

DATXREQ true indicates that a Unibus master has initiated a DATO or DATI transfer with the interface and causes a branch to the subroutine at label DATX, with the return address being saved in the FPC SREG. Unibus signal C1 is tested to determine direction, and then a DATO or DATI slave sequence is completed beginning at label DATO or DATI. At DATI, Unibus signal SSYN is asserted and data gated onto the Unibus using DATAOUT, until test MSYN is negated. The next instruction has no control signals asserted (OFF), and returns from the subroutine by branching to the address saved in SREG. DATO processing is similar.

DMAREQ indicates that the device is requesting a Direct Memory Access cycle, which causes a branch to label NPRX. The program waits at NPRX until NPG is de-asserted. NPR is then asserted and the program loops at NPR1 until NPG is reasserted. SACK is asserted, and the program loops at NPR2 until the three signals NPG, BBSY, and SSYN are unasserted. Note how the compare instruction masks the test inputs with the constant NPG_BBSY_SSYN and compares the result to 0. This allows concurrent testing of three inputs in only two microcycles. BBSY is asserted, making the interface bus master, and WRITE is tested to determine DMA direction. If a DATI cycle is to occur, we fall through to NPRDATI.

Front-end 150 ns de-skewing is done at NPRDATI and WAIT1, concurrent with loading the FPC CREG with 31 hex for a 15 microsecond slave timeout. WAIT2 is the top of the timeout loop. If the slave Unibus device asserts SSYN within 15 microseconds, the program branches to pass1 for tail-end 75 ns de-skew. Otherwise it falls through to the error exit at ERROR1. DATO processing is similar to DATI, and begins at NPRDATO.

INTREQ is asserted when the device wants to interrupt the Unibus CPU, causing execution to continue at INTR0. Interrupt request/grant processing occurs at INTR0 and INTR1. SACK is then asserted and the program loops at INTR2 until BG, BBSY, and SSYN are unasserted. The device supplied interrupt vector is gated onto the Unibus data lines at INTR3, and the interrupt handshake is finished at WAIT0.

5.6 CONCLUSION

One of the advantages of microprogrammed design is that it is relatively easy to change. In this application, Unibus DATOB and DATIP transfers were not differentiated from DATO and DATI transfers. This could be easily accommodated by modifying the DATX microcode to test Unibus signal C1 by adding a few words of additional code. Another change to be considered is to change the device interface to a less rudimentary protocol. Additional control signals could be provided by adding a decoder at the FPC output, and encoding eight signals using only 3 microword bits. Spare multiplexer inputs could be used for additional device status lines. Additional control signals can also be provided by adding another FPC.

```
"     Unibus Controller microcode using Am29PL141 assembler          "

"     Version 1.2                   R. Purvis, 19 December 85        "

device (PL141)
default = 1 ;

define
"       **********   DEFINITION OF TEST INPUTS   ***********  "

        tmsyn = t0              " test Unibus signal MSYN     "
        tssyn = t1              "                    SSYN     "
        tbbsy = t2              "                    BBSY     "
        tbg   = t3              "                    BG       "
        tnpg  = t4              "                    NPG      "
        aux   = t5              " auxiliary test conditions   "
        pass  = cc              " unconditional pass          "
        bg_bbsy_ssyn  = 0e#h    " test mask                   "
        npg_bbsy_ssyn = 16#h    " test mask                   "

"       **********  DEFINITION OF OUTPUTS  ****************  "

"       AUXILIARY TEST CONDITIONS                             "
        datxreq = 0000#h      " Unibus DATI or DATO request   "
        dmareq  = 1000#h      " device DMA request            "
        intreq  = 2000#h      " device Interrupt request      "
        write   = 3000#h      " device write request          "
        tcl     = 4000#h      " Unibus signal cl              "
                              " aux tests 5 - 7 are unused    "

"       CONTROL SIGNALS                                       "
        off     = 0000#h       " no signals active            "
        error   = 8000#h       " error flag to device         "
        addr    = 0800#h       " gate address onto Unibus     "
        dataout = 0400#h       " gate data onto Unibus        "
        datain  = 0200#h       " strobe data in from Unibus   "
        complt  = 0100#h       " complete flag to device      "

        cl      = 0080#h       " assert Unibus signal Cl      "
        intr    = 0040#h       "                      INTR    "
        br      = 0020#h       "                      BR      "
        npr     = 0010#h       "                      NPR     "
        sack    = 0008#h       "                      SACK    "
        bbsy    = 0004#h       "                      BBSY    "
        ssyn    = 0002#h       "                      SSYN    "
        msyn    = 0001#h ;     "                      MSYN    "

        test_condition = cc;   " default test condition       "

begin      "   ***************   Source Code   ***************       "
           "   Unibus Controller V1.2                                "

"          ************************************************************     "
"          *   MAIN LOOP  -  Loop at TOP until external condition           "
"          *   DATXREQ, DMAREQ, or INTREQ is true.                          "
"          ************************************************************     "

top:       datxreq, if (aux) call pl(datx);
           dmareq, if (aux) call pl(nprx);
           intreq, if (not aux) goto pl(top);


"          ************************************************************     "
"          *    INTERRUPT SERVICE ROUTINE  -  Device interrupt service      "
"          *    request.  Perform Unibus interrupt handshake.               "
"          ************************************************************     "
```

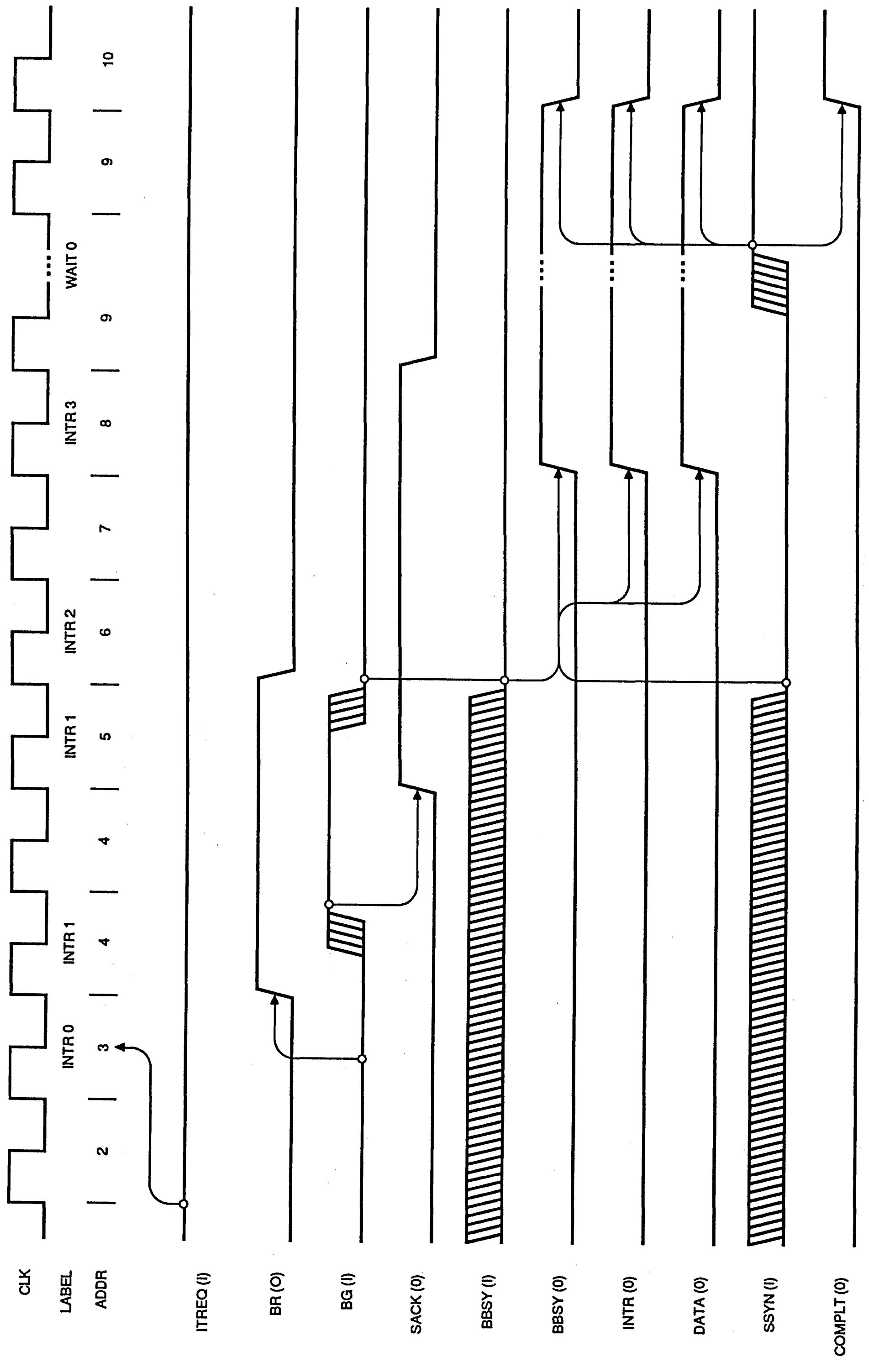
Figure 5-4. Unibus Controller Source Program Listing (Sheet 1 of 2)

```
intr0:      off, if (tbg) goto pl (intr0);        " request/grant handshake "
intr1:      br, if (not tbg) goto pl(intr1);
            br + sack, continue;
intr2:      br + sack, cmp tm(bg_bbsy_ssyn) to pl(0);
            br + sack, if (not eq) goto pl(intr2);
intr3:      sack + bbsy + intr + dataout, continue;     " interrupt vector "
wait0:      bbsy + intr + dataout, if (not tssyn) goto pl(wait0);
            complt, goto pl(top);


"           ***********************************************************      "
"           *     PROGRAMMED I/O ROUTINE  -  Unibus master accessing        "
"           *     device.  Perform Unibus DATO/DATI handshake.              "
"           ***********************************************************      "

datx:       tcl, if (aux) goto pl(dato);

dati:       ssyn + dataout, if (tmsyn) goto pl(dati);    " unibus slave DATI "
            off, ret;

dato:       ssyn + datain, if (tmsyn) goto pl(dato);   " unibus slave DATO "
            off, ret;

"           ***********************************************************      "
"           *     DMA SERVICE ROUTINE  -  Device DMA service request.       "
"           *     Perform Unibus DMA handshake.                             "
"           ***********************************************************      "

nprx:       off, if (tnpg) goto pl(nprx);       "  request/grant handshake  "
npr1:       npr, if (not tnpg) goto pl(npr1);
            npr + sack, continue;
npr2:       npr + sack, cmp tm(npg_bbsy_ssyn) to pl(0);
            npr + sack, if (not eq) goto pl(npr2);
            bbsy + write, if (aux) goto pl(nprdato);      " bus master now "


"           DMA READ ROUTINE  (unibus master DATI)                          "

nprdati:    bbsy + addr, load pl(31#h);
wait1:      bbsy + addr, if (tssyn) goto pl(wait1);
wait2:      bbsy + addr + msyn, if (tssyn) goto pl(pass1);   " 15 us        "
            bbsy + addr + msyn, if (tssyn) goto pl(pass1);   " timeout      "
            bbsy + addr + msyn, if (tssyn) goto pl(pass1);
            bbsy + addr + msyn, while (creg<>0) loop to pl(wait2);
error1:     bbsy + addr + error, ret;                    "  timeout error  "
pass1:      bbsy + addr + complt + datain, ret;          "  normal exit    "


"           DMA WRITE ROUTINE  (unibus master DATO)                         "

nprdato:    bbsy + addr + cl + dataout, load pl(31#h);
wait3:      bbsy + addr + cl + dataout, if (tssyn) goto pl (wait3);
wait4:      bbsy + addr + cl + dataout + msyn, if (tssyn) goto pl(pass2);
            bbsy + addr + cl + dataout + msyn, if (tssyn) goto pl(pass2);
            bbsy + addr + cl + dataout + msyn, if (tssyn) goto pl(pass2);
            bbsy + addr + cl + dataout + msyn, while(creg<>0)loop to pl(wait4);
error2:     bbsy + addr + cl + error, ret;                 "  timeout error  "
pass2:      bbsy + addr + cl + complt, ret;                "  normal exit    "
            .org 63#d
            off, goto pl(0);                         "  hardware reset here. "

end.
```

Figure 5-4. Unibus Controller Source Program Listing (Sheet 2 of 2)

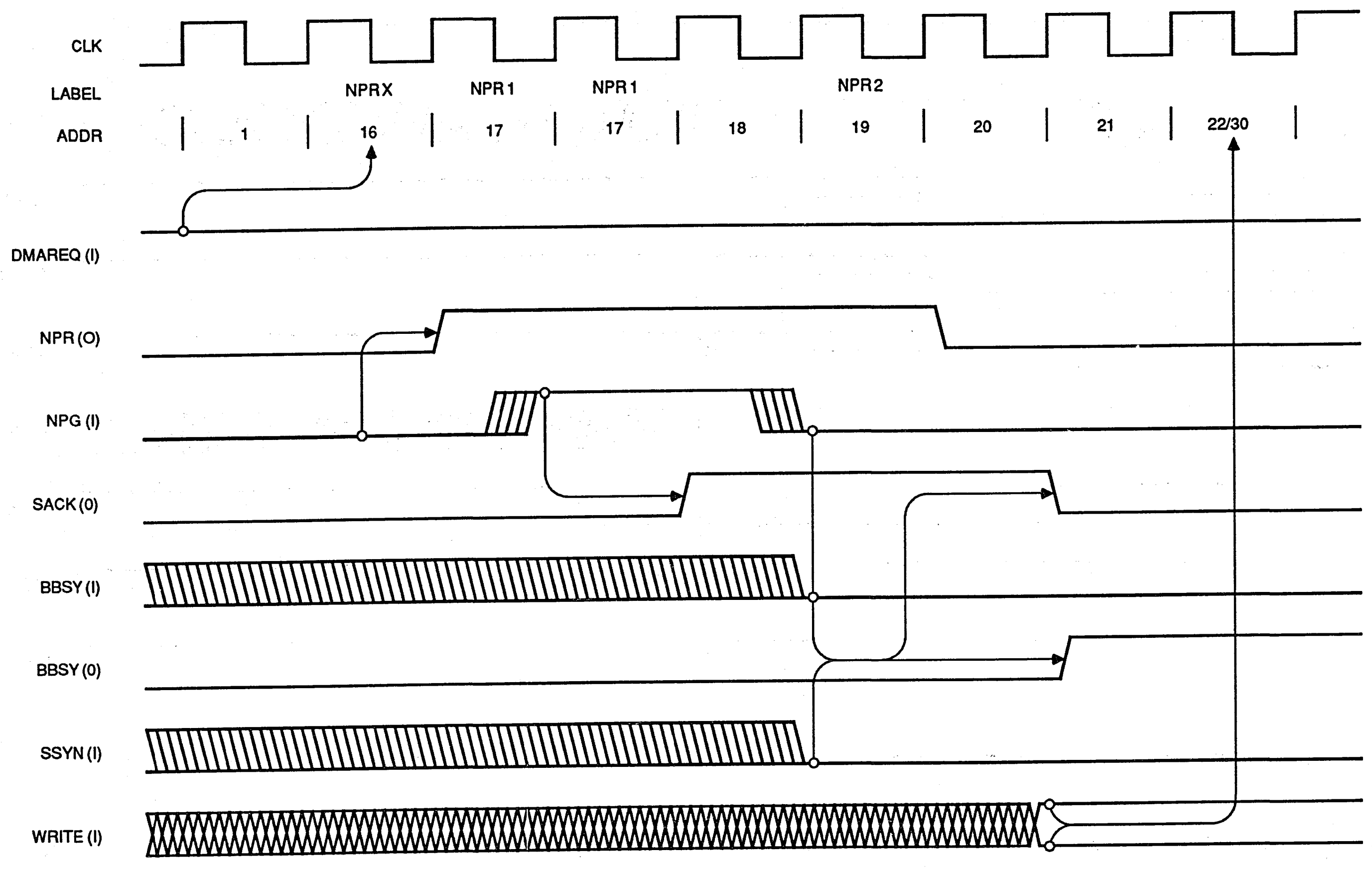
```
PROM Contents are :

hex <dec>      OE  OPCODE POL  TEST   DATA          OUTPUT
000 <   0>   [ 1 | 11100 | 0 | 101 | 001011 | 0000000000000000 ]
001 <   1>   [ 1 | 11100 | 0 | 101 | 010000 | 0001000000000000 ]
002 <   2>   [ 1 | 11001 | 1 | 101 | 000000 | 0010000000000000 ]
003 <   3>   [ 1 | 11001 | 0 | 011 | 000011 | 0000000000000000 ]
004 <   4>   [ 1 | 11001 | 1 | 011 | 000100 | 0000000000100000 ]
005 <   5>   [ 1 | 01101 | 1 | 111 | 111111 | 0000000000101000 ]
                 OPCODE    CONSTANT   DATA
006 <   6>   [ 1 |   100 |   000000 | 001110 | 0000000000101000 ]
007 <   7>   [ 1 | 11001 | 1 | 111 | 000110 | 0000000000101000 ]
008 <   8>   [ 1 | 01101 | 1 | 111 | 111111 | 0000010001001100 ]
009 <   9>   [ 1 | 11001 | 1 | 001 | 001001 | 0000010001000100 ]
00A <  10>   [ 1 | 11001 | 0 | 110 | 000000 | 0000000100000000 ]
00B <  11>   [ 1 | 11001 | 0 | 101 | 001110 | 0100000000000000 ]
00C <  12>   [ 1 | 11001 | 0 | 000 | 001100 | 0000010000000010 ]
00D <  13>   [ 1 | 00010 | 0 | 110 | 111111 | 0000000000000000 ]
00E <  14>   [ 1 | 11001 | 0 | 000 | 001110 | 0000001000000010 ]
00F <  15>   [ 1 | 00010 | 0 | 110 | 111111 | 0000000000000000 ]
010 <  16>   [ 1 | 11001 | 0 | 100 | 010000 | 0000000000000000 ]
011 <  17>   [ 1 | 11001 | 1 | 100 | 010001 | 0000000000010000 ]
012 <  18>   [ 1 | 01101 | 1 | 111 | 111111 | 0000000000011000 ]
                 OPCODE    CONSTANT   DATA
013 <  19>   [ 1 |   100 |   000000 | 010110 | 0000000000011000 ]
014 <  20>   [ 1 | 11001 | 1 | 111 | 010011 | 0000000000011000 ]
015 <  21>   [ 1 | 11001 | 0 | 101 | 011110 | 0011000000000100 ]
016 <  22>   [ 1 | 00100 | 0 | 110 | 110001 | 0000100000000100 ]
017 <  23>   [ 1 | 11001 | 0 | 001 | 010111 | 0000100000000100 ]
018 <  24>   [ 1 | 11001 | 0 | 001 | 011101 | 0000100000000101 ]
019 <  25>   [ 1 | 11001 | 0 | 001 | 011101 | 0000100000000101 ]
01A <  26>   [ 1 | 11001 | 0 | 001 | 011101 | 0000100000000101 ]
01B <  27>   [ 1 | 01000 | 0 | 110 | 011000 | 0000100000000101 ]
01C <  28>   [ 1 | 00010 | 0 | 110 | 111111 | 1000100000000100 ]
01D <  29>   [ 1 | 00010 | 0 | 110 | 111111 | 0000101100000100 ]
01E <  30>   [ 1 | 00100 | 0 | 110 | 110001 | 0000110010000100 ]
01F <  31>   [ 1 | 11001 | 0 | 001 | 011111 | 0000110010000100 ]
020 <  32>   [ 1 | 11001 | 0 | 001 | 100101 | 0000110010000101 ]
021 <  33>   [ 1 | 11001 | 0 | 001 | 100101 | 0000110010000101 ]
022 <  34>   [ 1 | 11001 | 0 | 001 | 100101 | 0000110010000101 ]
023 <  35>   [ 1 | 01000 | 0 | 110 | 100000 | 0000110010000101 ]
024 <  36>   [ 1 | 00010 | 0 | 110 | 111111 | 1000100010000100 ]
025 <  37>   [ 1 | 00010 | 0 | 110 | 111111 | 0000100110000100 ]
03F <  63>   [ 1 | 11001 | 0 | 110 | 000000 | 0000000000000000 ]
```

Figure 5-5. FPC PROM Contents



06591A 5-6

Figure 5-6. BR Timing Diagram



06591A 5-7

Figure 5-7. NPR Timing Diagram

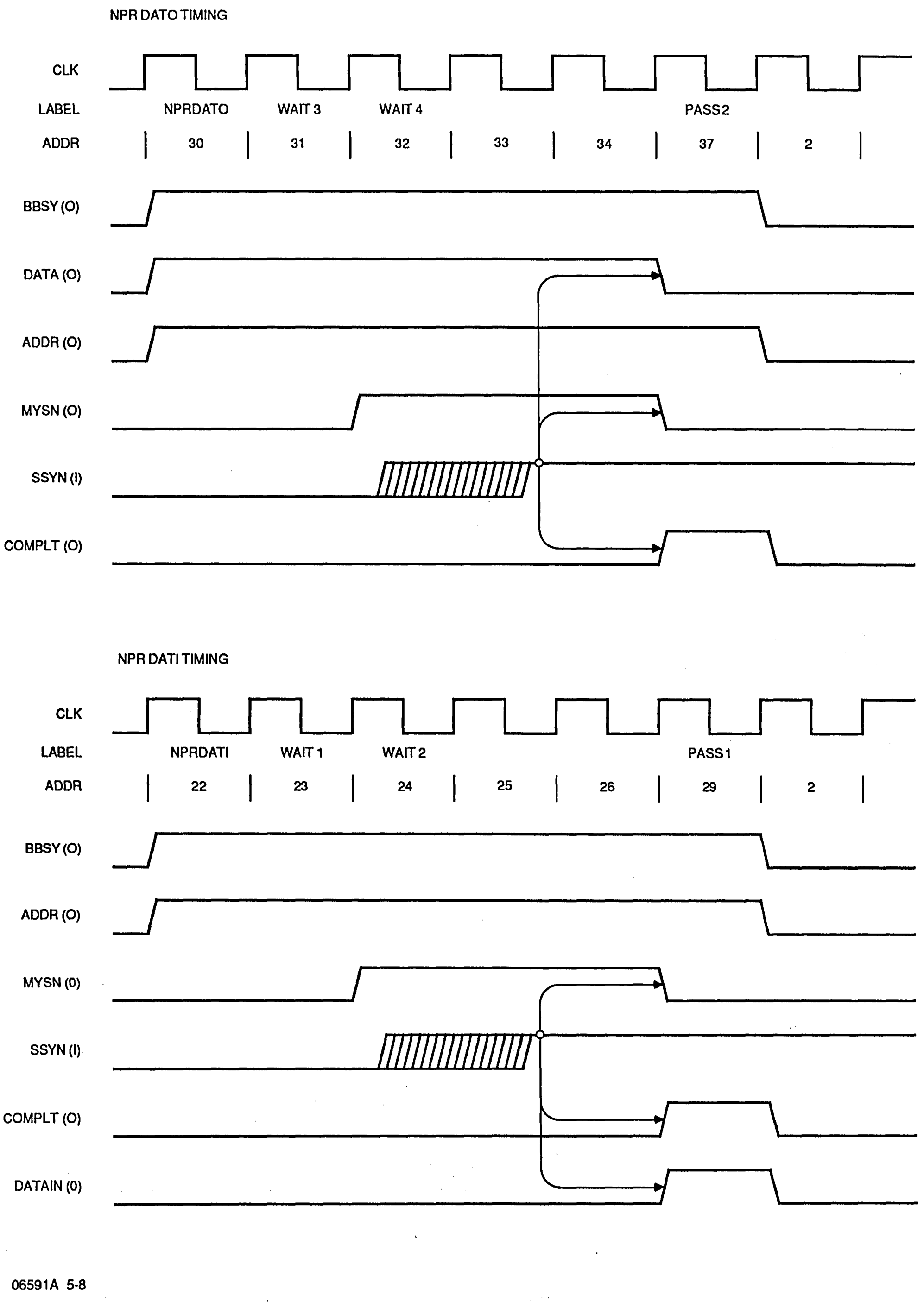
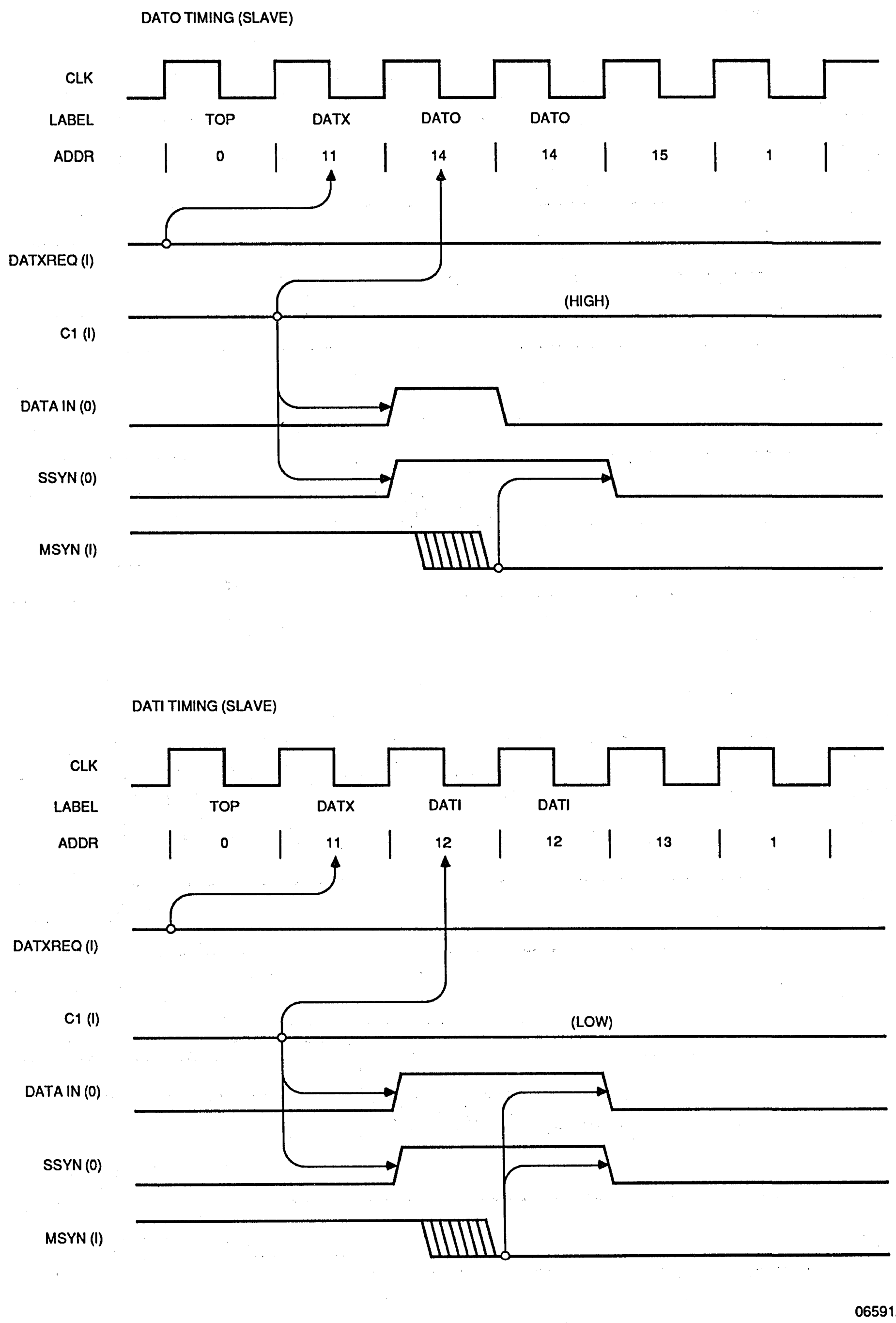


Figure 5-8. NPR DATI and DATO Timing Diagram



06591A 5-9

Figure 5-9. DATI and DATO (Slave) Timing Diagram

CHAPTER 6

Am29PL141 BASED DEC Q-BUS CONTROLLER

6.1 THE DESIGN PROBLEM

Designing an interface for the DEC Q-Bus has been approached using many techniques. One technique, microprogramming, has in the past been economically unattractive because it required use of a separate sequencer, control store, and pipeline registers. Now that Advanced Micro Devices has introduced the single chip Am29PL141 Fuse Programmable Controller, engineers can economically apply powerful microprogramming techniques to the design of medium complexity state machines like that required to control the Q-Bus.

The problem is to design an interface between the Q-Bus and a generic device to allow the following operations:

- DATI/DATO with device as slave
- Device interrupt request
- Device direct memory access request
- DATI/DATO with device as master

The DEC Q-Bus is an asynchronous bus which supports Programmed I/O, prioritized Interrupts, and Direct Memory Access (DMA) operations. All bus transfers are between a bus master and bus slave, and are controlled by the master. An arbitrator grants bus mastership to requesting devices.

The nine basic types of transfers allowed are:

DATI	–	Word data transfer from slave to master
DATO	–	Word data transfer from master to slave
DATOB	–	Byte data transfer from master to slave
DATIO	–	Read-modify-write word transfer
DATIOB	–	Read-modify-write byte transfer
DATBI	–	Block data transfer from slave to master
DATBO	–	Block data transfer from master to slave
DMR	–	Direct Memory Access request to become bus master.
IRQi	–	Interrupt request at level i (4,5,6,or 7).

The following control signals are used during transfers:

SNYC	sync	– master timing control
DOUT	data out	– indicates master write
DIN	data in	– indicates master read
RPLY	reply	– slave acknowledge
WTBT	write/byte	– byte write cycle
BS7	I/O page select	
IRQi	interrupt request level i	
IAK	interrupt grant	
DMR	DMA request	
DMG	DMA grant	
SACK	select acknowledge	

6.2 Q-BUS CONTROLLER HARDWARE DESIGN

A block diagram of this interface is shown in Figure 6-1. It consists of three sections—Q-Bus buffering, address decoding, and control logic. The address decoder detects addressing of the device as a slave during DATI and DATO transfers.

The control logic is based on the Am29PL141 Fuse Programmable Controller (FPC). Its microprogram implements a state machine to control both device and Q-Bus handshaking. Test inputs are synchronized with the FPC clock using an AM29821A 10-bit register and a D flip-flop. Note the use of a multiplexer to expand the FPC test capability. The additional D flip-flop and AND gates are used to implement the interrupt and DMA request/grant handshaking.

6.3 MICROWORD FORMAT

The microword organization for this application of the FPC is shown in Figure 6-2. The 32-bit microword is subdivided into fields of various sizes and functions. The 16 most significant bits are used during next address generation within the FPC, while the lower 16 bits are application interface signals.

6.4 MICROCODE

The microcode of Figure 6-3 was written using the Am29PL141 assembler available from AMD. Mnemonic definitions are shown, followed by code to control the interface. Figure 6-4 shows the FPC PROM contents. A brief description of the code follows.

After reset to address 63, the program branches to label TOP and loops until one of the external conditions DATXREQ, DMAREQ, or INTREQ is asserted.

DATXREQ true indicates a Q-Bus DATO or DATI operation addressing the device and causes a subroutine call to DATX. Q-Bus signal WTBT is tested, and DATO or DATI handshaking is completed beginning at label DATO or DATI.

INTREQ is asserted when the device wants to interrupt the CPU, causing execution to continue at INTR0. Interrupt request/grant processing occurs and then the vector is read by the CPU.

6.5 CONCLUSION

The problem statement for this interface does not require block, byte, or read-modify-write master handshaking. These features can be implemented by adding extra device request lines and microcoding the additional handshake algorithms. Another possible change is to implement the Q-Bus four-level interrupt configuration. These changes are left as an exercise for the interested reader!

References:

Microsystems Handbook, Digital Equipment Corporation, 1985.

Am29PL141 FPC Data Sheet, Advanced Micro Devices, 1987.

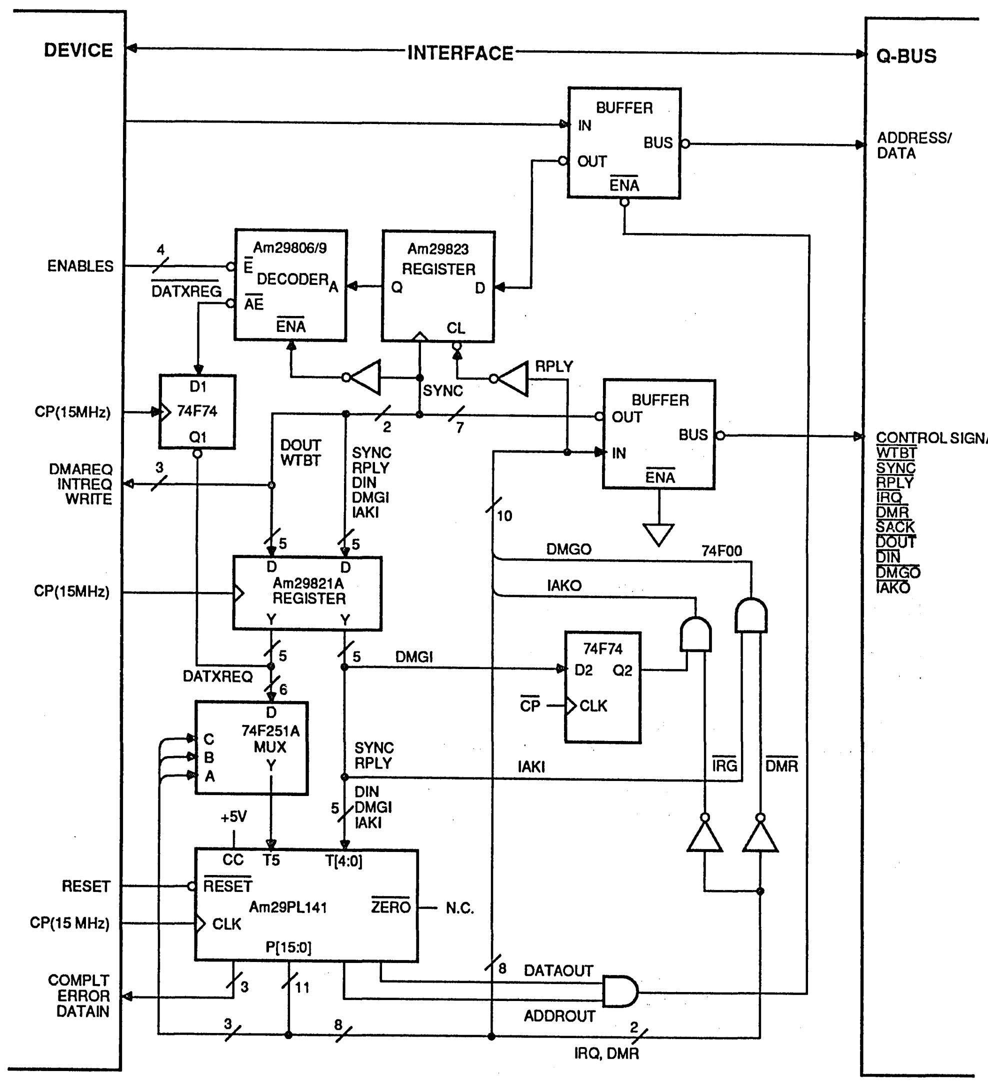


Figure 6-1. Q-Bus Controller Block Diagram

```
: 31 :  30 - 26  :  25 : 24,23,22 : 21 - 16 :   15   : 14,13,12 :   11 - 0   :
:....:...........:....:..........:.........:........:..........:............:
: oe :   opcode  : pol :   test   :   data  : error  : aux tst  :  command   :
:....:...........:....:..........:.........:........:..........:............:

oe:             output enable
(31)

opcode:         29PL141 command
(30-26)         00 - RETPL      08 - LPPL      10 - CMP       18 - FORK
                01 - RETPLN     09 - DEC       11 - CMP       19 - GOTOPL
                02 - RET        0A - LPPLN     12 - CMP       1A - WAIT
                03 - RETN       0B - GOTOPLZ   13 - CMP       1B - DECGO/C
                04 - LDPL       0C - DECAL     14 - PSHPL     1C - CALPL
                05 - LDPLN      0D - CONT      15 - PSH       1D - CALPLN
                06 - LDTM       0E - CTTM      16 - PSHTM     1E - CALTM
                07 - LDTMN      0F - GOTOTM    17 - PSHN      1F - CALTMN

pol:            test polarity  ( 1 = negate )
(25)

test:           conditional test input select
(24,23,22)           0 - sync        4 - iak
                     1 - rply        5 - aux tests
                     2 - din         6 - pass
                     3 - dmg         7 - equal flag

data:           branch address, test input mask, or counter load value
(21-16)

error:          timeout error indication to device
(15)

aux test:       additional test inputs when test = 5
(14,13,12)           0 - datxreq     4 - dout
                     1 - dmareq      5 - wtbt
                     2 - intreq      6 - spare
                     3 - write       7 - spare

command:
(11-0)

:  11 :  10  :  9  :  8  :  7  :  6 :  5  :  4  :  3  :  2 :  1  :  0  :
:.....:......:.....:.....:.....:....:.....:.....:.....:....:.....:.....:
: com : data : data : addr : rply : irq : dmr : sack : dout : din : sync : wtbt :
: plt :  in  : out* : out* :      :     :     :      :      :     :      :      :
:.....:......:.....:.....:.....:....:.....:.....:.....:....:.....:.....:

* - indicates active low microcode bits
```

Figure 6-2. Q-Bus Controller Microword Format

```
"     Q-Bus Controller microcode using Am29PL141 assembler        "

"     Version 1.1                    R. Purvis, 3 January 86       "

device (pl141)
default = 1 ;

define
"      **********   DEFINITION OF TEST INPUTS   ********   "

       tsync = t0               " test Q-Bus signal SYNC "
       trply = t1               "                   RPLY "
       tdin  = t2               "                   DIN  "
       tdmgi = t3               "                   DMGI "
       tiaki = t4               "                   IAKI "
       aux   = t5               " auxiliary test conditions "
       pass  = cc               " unconditional pass "
       sync_rply  = 03#h        " test mask "

"      **********  DEFINITION OF OUTPUTS  **************     "

"      AUXILIARY TEST CONDITIONS                              "
       datxreq = 0300#h      " Q-Bus DATI or DATO request    "
       dmareq  = 1300#h      " device DMA request            "
       intreq  = 2300#h      " device Interrupt request      "
       write   = 3300#h      " device write request          "
       tdout   = 4300#h      " Q-Bus signal DOUT             "
       twtbt   = 5300#h      " Q-Bus signal WTBT             "
                             " aux tests 6 and 7 are spares  "

"      CONTROL SIGNALS                                        "
       off     = 0300#h       " no signals active            "
       error   = 8300#h       " error flag to device         "
       complt  = 0B00#h       " complete flag to device      "
       datain  = 0700#h       " strobe data in from Q-Bus    "
       dataout = FDFF#h       " gate data onto Q-Bus         "
       addrout = FEFF#h       " gate address onto Q-Bus      "

       rply    = 0380#h       " assert Q-Bus signal RPLY     "
       irq     = 0340#h       "                     IRQ      "
       dmr     = 0320#h       "                     DMR      "
       sack    = 0310#h       "                     SACK     "
       dout    = 0308#h       "                     DOUT     "
       din     = 0304#h       "                     DIN      "
       sync    = 0302#h       "                     SYNC     "
       wtbt    = 0301#h ;     "                     WTBT     "

test_condition = cc;  " default test condition    "
```

Figure 6-3. Q-Bus Controller Source Program Listing (Sheet 1 of 3)

```
" *****   assumptions   ******          "

"   -  no I/O page DMA                  "
"   -  single xfer DMA (not block mode)  "
"   -  single level interrupts          "
"   -  no byte operations               "
"   -  no parity                        "

begin      "   ***************  Source Code   ***************         "
           "   Q-Bus Controller V1.0                                  "


"          **********************************************************   "
"          *    MAIN LOOP  -  Loop at TOP until external condition      "
"          *    DATXREQ, DMAREQ, or INTREQ is true.                     "
"          **********************************************************   "

top:       datxreq, if (aux) call pl(datx);
           dmareq, if (aux) call pl(dmax);
           intreq, if (not aux) goto pl(top);


"          ***********************************************************  "
"          *     INTERRUPT SERVICE ROUTINE  -  Device interrupt service "
"          *     request.  Perform Q-Bus interrupt handshake.           "
"          ***********************************************************  "

intr0:     off, if (tdin) goto pl(intr0);       "  request/grant handshake "
intr1:     irq, if (not tdin) goto pl(intr1);
intr2:     irq, if (not tiaki) goto pl(intr2);
intr3:     rply * dataout, if (tdin) goto pl(intr3);      "  output vector "
intr4:     rply * dataout, if (tiaki) goto pl(intr4);
           complt, goto pl(top);


"          ***********************************************************  "
"          *     PROGRAMMED I/O ROUTINE  -  Q-Bus master accessing      "
"          *     device.  Perform Q-Bus DATO/DATI handshake.            "
"          ***********************************************************  "

datx:      twtbt, if (aux) goto pl(dato);

dati:      off, if (not tdin) goto pl(dati);             " slave DATI "
wait6:     rply * dataout, if (tdin) goto pl(wait6);
           off, ret;

dato:      tdout, if (not aux) goto pl(dato);            " slave DATO "
wait5:     rply + datain + tdout, if (aux) goto pl(wait5);
           off, ret;
```

Figure 6-3. Q-Bus Controller Source Program Listing (Sheet 2 of 3)

```
"          *********************************************************    "
"          *     DMA SERVICE ROUTINE  -  Device DMA service request.     "
"          *     Perform Q-Bus DMA handshake.                            "
"          *********************************************************    "

dmax:      off, if (tdmgi) goto pl(dmax);       "  request/grant handshake "
dma1:      dmr, if (not tdmgi) goto pl(dma1);
dma2:      dmr, cmp tm(sync_rply) to pl(0);
           dmr, if (not eq) goto pl(dma2);
           sack + write, if (aux) goto pl(dmadato);    "  bus master now "

"          DMA READ ROUTINE  (Q-Bus master DATI)                          "

dmadati:   sack * addrout, continue;                    "  addr setup   "
           sack * addrout, continue;
          (sack + sync) * addrout, continue;            "  addr hold    "
          (sack + sync) * addrout, load pl(2B#h);       " 10 us timeout "
wait1:     sack + sync + din, if (trply) goto pl(pass1);
           sack + sync + din, if (trply) goto pl(pass1);
           sack + sync + din, while (creg<>0) loop to pl(wait1);
error1:    sack + sync + error, ret;                    " timeout exit  "
pass1:     sack + sync + din, continue;                 " data deskew   "
           sack + sync + din, continue;
wait2:     sack + sync, if (trply) goto pl(wait2);      " clock data in "
           complt, ret;

"          DMA WRITE ROUTINE  (Q-Bus master DATO)                         "

dmadato:  (sack + wtbt) * addrout, continue;             "  addr setup  "
          (sack + wtbt) * addrout, continue;
          (sack + wtbt + sync) * addrout, continue;      "  addr hold   "
          (sack + wtbt + sync) * addrout, load pl(2b#h);
wait3:    (sack + sync + dout) * dataout, if (trply) goto pl(pass2);
          (sack + sync + dout) * dataout, if (trply) goto pl(pass2);
          (sack + sync + dout) * dataout, while(creg<>0)loop to pl(wait3);
error2:    sack + sync + error, ret;                    " timeout exit  "
pass2:    (sack + sync + dout) * dataout, continue;     " data deskew   "
          (sack + sync + dout) * dataout, continue;
          (sack + sync) * dataout, continue;            " data hold     "
          (sack + sync) * dataout, continue;
wait4:     sack + sync, if(trply) goto pl(wait4);
           complt, ret;
           .org 63#d
           off, goto pl(0);                    "  hardware reset here. "

end.
```

Figure 6-3. Q-Bus Controller Source Program Listing (Sheet 3 of 3)

```
 PROM Contents are :
hex <dec>      OE  OPCODE POL  TEST   DATA            OUTPUT
000 <   0>   [ 1 | 11100 | 0 | 101 | 001001 | 0000001100000000 ]
001 <   1>   [ 1 | 11100 | 0 | 101 | 010000 | 0001001100000000 ]
002 <   2>   [ 1 | 11001 | 1 | 101 | 000000 | 0010001100000000 ]
003 <   3>   [ 1 | 11001 | 0 | 010 | 000011 | 0000001100000000 ]
004 <   4>   [ 1 | 11001 | 1 | 010 | 000100 | 0000001101000000 ]
005 <   5>   [ 1 | 11001 | 1 | 100 | 000101 | 0000001101000000 ]
006 <   6>   [ 1 | 11001 | 0 | 010 | 000110 | 0000000110000000 ]
007 <   7>   [ 1 | 11001 | 0 | 100 | 000111 | 0000000110000000 ]
008 <   8>   [ 1 | 11001 | 0 | 110 | 000000 | 0000101100000000 ]
009 <   9>   [ 1 | 11001 | 0 | 101 | 001101 | 0101001100000000 ]
00A <  10>   [ 1 | 11001 | 1 | 010 | 001010 | 0000001100000000 ]
00B <  11>   [ 1 | 11001 | 0 | 010 | 001011 | 0000000110000000 ]
00C <  12>   [ 1 | 00010 | 0 | 110 | 111111 | 0000001100000000 ]
00D <  13>   [ 1 | 11001 | 1 | 101 | 001101 | 0100001100000000 ]
00E <  14>   [ 1 | 11001 | 0 | 101 | 001110 | 0100011110000000 ]
00F <  15>   [ 1 | 00010 | 0 | 110 | 111111 | 0000001100000000 ]
010 <  16>   [ 1 | 11001 | 0 | 011 | 010000 | 0000001100000000 ]
011 <  17>   [ 1 | 11001 | 1 | 011 | 010001 | 0000001100100000 ]
                 OPCODE    CONSTANT   DATA
012 <  18>   [ 1 |   100 |  000000 | 000011 | 0000001100100000 ]
013 <  19>   [ 1 | 11001 | 1 | 111 | 010010 | 0000001100100000 ]
014 <  20>   [ 1 | 11001 | 0 | 101 | 100001 | 0011001100010000 ]
015 <  21>   [ 1 | 01101 | 1 | 111 | 111111 | 0000001000010000 ]
016 <  22>   [ 1 | 01101 | 1 | 111 | 111111 | 0000001000010000 ]
017 <  23>   [ 1 | 01101 | 1 | 111 | 111111 | 0000001000010010 ]
018 <  24>   [ 1 | 00100 | 0 | 110 | 101011 | 0000001000010010 ]
019 <  25>   [ 1 | 11001 | 0 | 001 | 011101 | 0000001100010110 ]
01A <  26>   [ 1 | 11001 | 0 | 001 | 011101 | 0000001100010110 ]
01B <  27>   [ 1 | 01000 | 0 | 110 | 011001 | 0000001100010110 ]
01C <  28>   [ 1 | 00010 | 0 | 110 | 111111 | 1000001100010010 ]
01D <  29>   [ 1 | 01101 | 1 | 111 | 111111 | 0000001100010110 ]
01E <  30>   [ 1 | 01101 | 1 | 111 | 111111 | 0000001100010110 ]
01F <  31>   [ 1 | 11001 | 0 | 001 | 011111 | 0000001100010010 ]
020 <  32>   [ 1 | 00010 | 0 | 110 | 111111 | 0000101100000000 ]
021 <  33>   [ 1 | 01101 | 1 | 111 | 111111 | 0000001000010001 ]
022 <  34>   [ 1 | 01101 | 1 | 111 | 111111 | 0000001000010001 ]
023 <  35>   [ 1 | 01101 | 1 | 111 | 111111 | 0000001000010011 ]
024 <  36>   [ 1 | 00100 | 0 | 110 | 101011 | 0000001000010011 ]
025 <  37>   [ 1 | 11001 | 0 | 001 | 101001 | 0000000100011010 ]
026 <  38>   [ 1 | 11001 | 0 | 001 | 101001 | 0000000100011010 ]
027 <  39>   [ 1 | 01000 | 0 | 110 | 100101 | 0000000100011010 ]
028 <  40>   [ 1 | 00010 | 0 | 110 | 111111 | 1000001100010010 ]
029 <  41>   [ 1 | 01101 | 1 | 111 | 111111 | 0000000100011010 ]
02A <  42>   [ 1 | 01101 | 1 | 111 | 111111 | 0000000100011010 ]
02B <  43>   [ 1 | 01101 | 1 | 111 | 111111 | 0000000100010010 ]
02C <  44>   [ 1 | 01101 | 1 | 111 | 111111 | 0000000100010010 ]
02D <  45>   [ 1 | 11001 | 0 | 001 | 101101 | 0000001100010010 ]
02E <  46>   [ 1 | 00010 | 0 | 110 | 111111 | 0000101100000000 ]
03F <  63>   [ 1 | 11001 | 0 | 110 | 000000 | 0000001100000000 ]
```

Figure 6-4. FPC PROM Program Listing

CHAPTER 7

STARLAN CONTROLLER USING Am7990 AND Am29PL141

7.1 THE DESIGN PROBLEM

This application note describes the use of an Am29PL141 to make it feasible to use an Am7990 and Am7960 in a Starlan type of environment. In this application, an Am29PL141 controls a dual-ported memory to isolate DMA transfers from the CPU. The Am7960 could just as easily be connected in a half duplex mode providing a very inexpensive Ethernet-like Local Area Network (LAN).

The 7990 is designed as a 10 MHz Ethernet/Cheapernet communications controller. Since this part also operates at 1 MHz, it is applicable for Starlan (IEEE 802.3 1 base 5).

The Am7990 has the following characteristics:

1. It is dedicated for 16-bit interface

2. Data transfer can only be in eight word, non-preemptible bursts. This requirement limits LANCE's flexibility in adapting to a network such as Starlan because of the great difference in speed between Starlan (1 MHz) and Ethernet (10 MHz). Therefore an intelligent bus arbitrator or a dual array buffer memory is desirable.

3. However, the transmit clock provides the reference for both the DMA state machine and the network transfer rate. Because of this, the DMA transfer rate is slow at the 1 MHz network transfer rate.

In a system environment, DMA Bus time is at a premium so, during packet data transfers for transmit or receive, the DMA transfers eight (8) words (16 bytes) at a time, with each transfer taking 6 TCLKs. This takes 4.8 microseconds at 10 MHz. However, slowing down the network speed to the 1 MHz Starlan speed in order to use conventional telephone cable has disastrous consequences on system throughput. At 1 MHz network speed, the System Bus time for a DMA transfer of eight words is 48 microseconds, far too long for the bus to be unavailable to the CPU.

This problem can be solved by using an Am29PL141 controller to manage the DMA transfer, freeing the CPU for other tasks.

7.2 FUNCTIONAL DESCRIPTION

The heart of the design is the two port memory arbitrator. This is accomplished in a 29PL141 Fuse Programmable Sequencer. Refer to the simplified block diagram, Figure 7-1, the control circuitry diagram, Figure 7-2, and the address and data circuitry, Figure 7-3. Figure 7-4 shows the miscellaneous circuitry including the READY, RESET, address decoder, and clock circuit.

As seen in Figure 7-3, the memory (two 8K x 8 Static RAMs) are buffered from the CPU and also from the Am7990. The memory chips are linked to a memory controller (U3 and U4) that makes them available to the main CPU to move data as well as to the 7990 to move packets and ring control data. In this manner, the CPU is isolated from the 48 microsecond burst time of the 7990 and, in fact, is isolated from all DMA activity. (This architecture could even be used at a 10 MHz Ethernet rate to provide the same DMA bus isolation for the CPU.)

In the following discussion, refer to Figure 7-2 for the control circuitry blocks: U1, 2, 3, 4, 13, 16, and 17.

U1 (25LS2521) provides I/O space address detection on 128 address granularity. See Figure 7-2.

U2 (25LS2521) provides memory bus address comparison and is jumper-selectable on 16K byte boundaries. The output from U2 goes to a decoder (see Figure 7-4) and then to U3. The decoder also provides a HOLDA active signal to the CPU bus at pin BD0. The CPU can test this pin to see if the Am7990 is using the DMA.

U3 is a 16L8 PAL device, which provides all the Bus interface control lines. See Section 7.4 for the actual equations of the PAL devices.

U4 (Am16L8) is the 7990's control PAL device. It handles Data Bus control and direction as well as the ready lines of the 7990. U4 also provides the Write Enable for the 99C88s to determine high and low byte writes.

U16 (Am29PL141) is the main memory arbritrator/scheduler. It provides control signals to control the Am7990. It also provides inputs to the

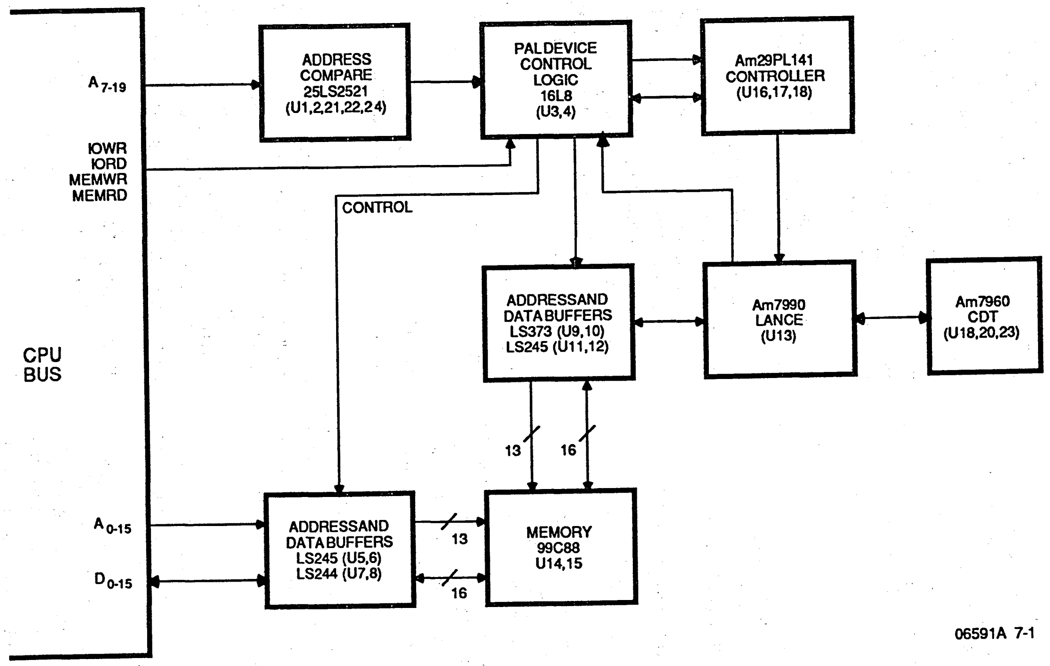

A7-19
IOWR
IORD
MEMWR
MEMRD
CPU BUS
A0-15
D0-15
ADDRESS COMPARE 25LS2521 (U1,2,21,22,24)
PAL DEVICE CONTROL LOGIC 16L8 (U3,4)
Am29PL141 CONTROLLER (U16,17,18)
CONTROL
ADDRESSAND DATABUFFERS LS373 (U9,10) LS245 (U11,12)
Am7990 LANCE (U13)
Am7960 CDT (U18,20,23)
13
16
ADDRESSAND DATABUFFERS LS245 (U5,6) LS244 (U7,8)
MEMORY 99C88 U14,15
13
16
06591A 7-1

Figure 7-1. Starlan DMA Controller Block Diagram

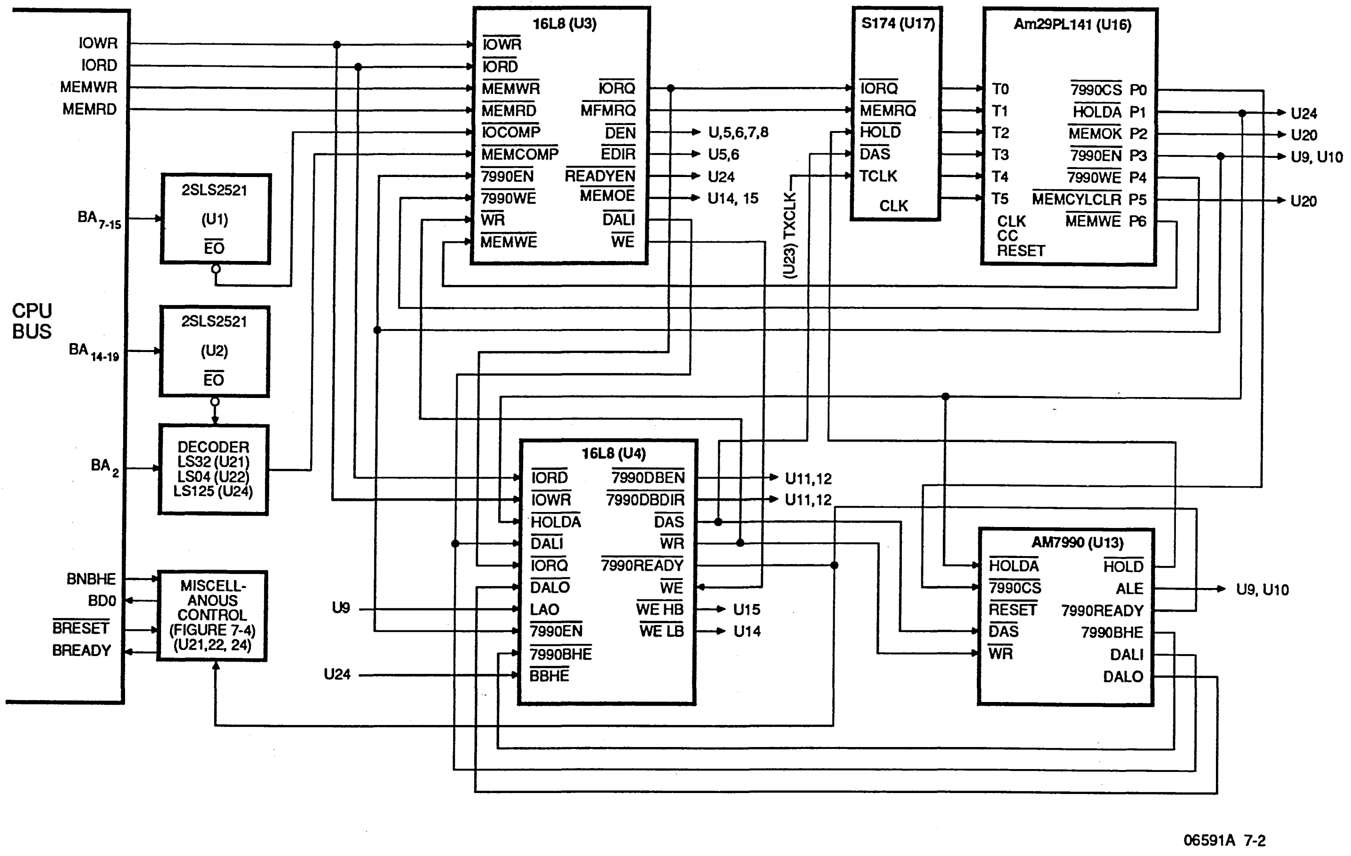


Figure 7-2. Starlan Controller Circuitry

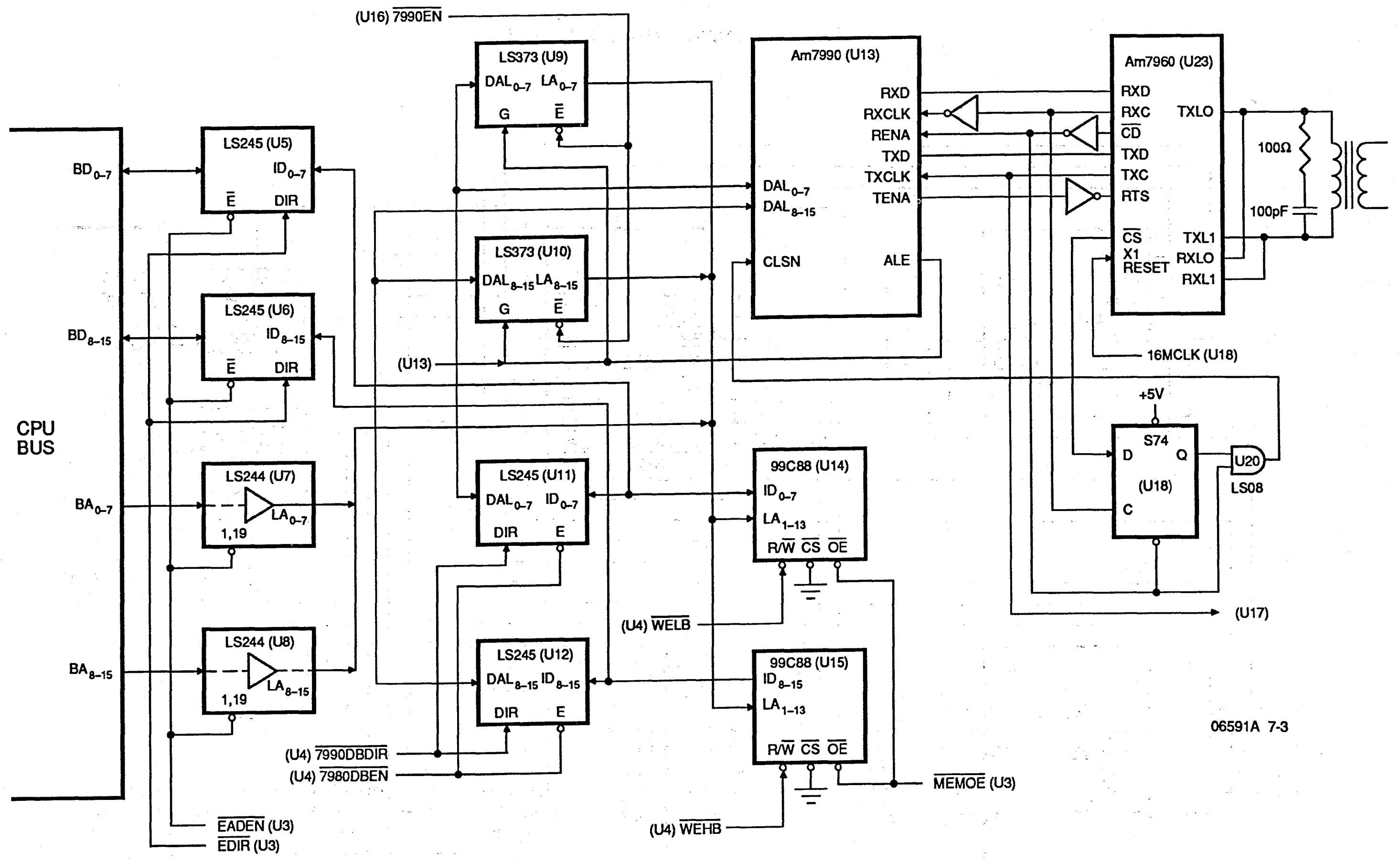


Figure 7-3. Starlan Address and Data Circuitry

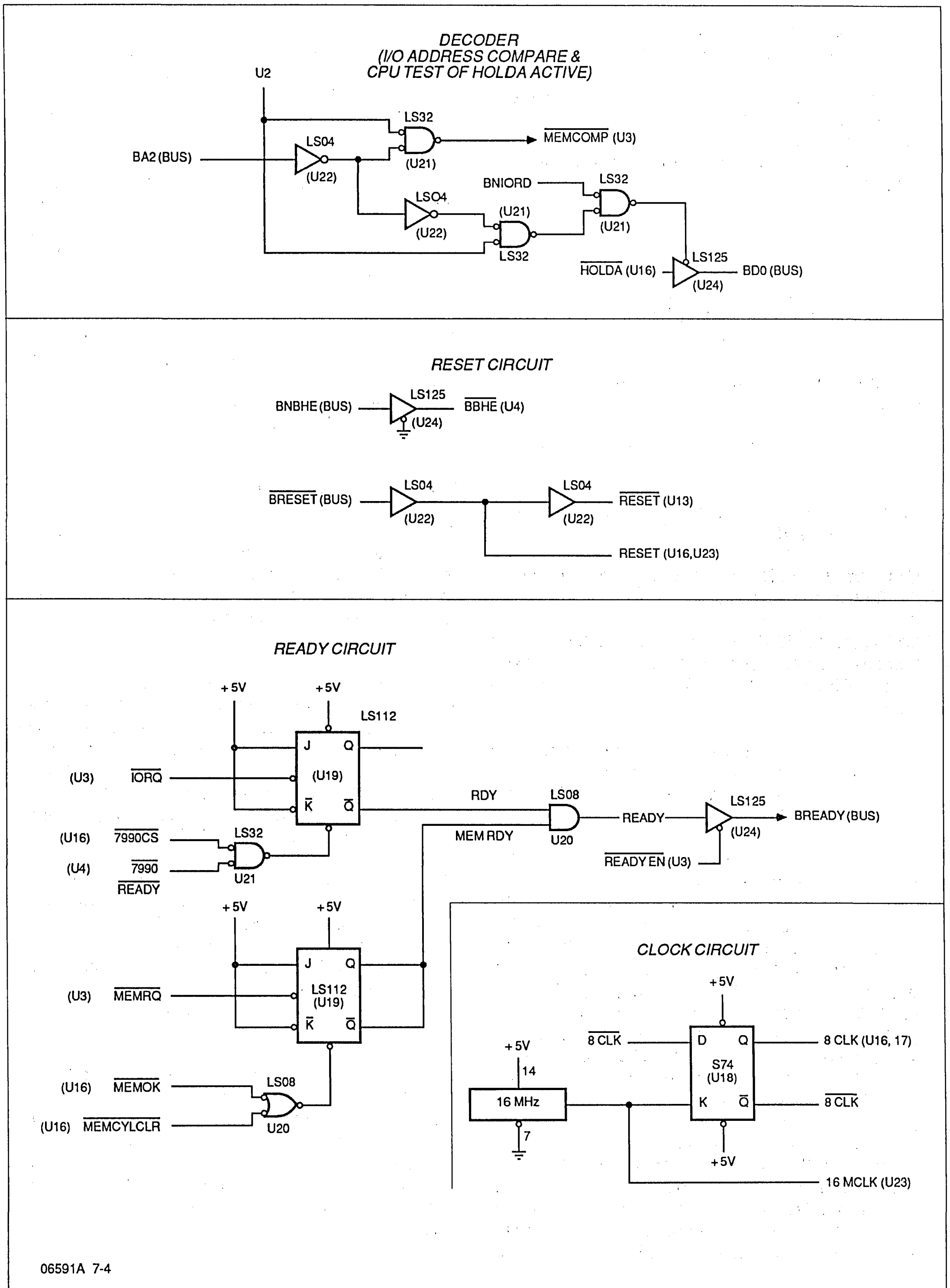


Figure 7-4. Miscellaneous Control Circuits

PAL Devices to control the memory access. See Figure 7-2 for the routing of its signals.

The Am29PL141 can accept seven (7) different test inputs and control 16 different events. This application uses six (6) input lines and eight (8) output lines to accomplish the handshaking and control.

U17 (S174) is used to provide metastability hardening of the Am29PL141.

In the following discussion, refer to Figure 7-3 for the address and data circuity blocks: U5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 18, and 23.

U5 and U6 (LS245) provide the Data Bus buffering.

U7and U8 (LS244) provide the address bus buffering.

U9 and U10 (LS373s) serve as address latches to demultiplex the 7990's DAL bus.

U11 and U12 (LS245s) are data buffers to isolate the 7990 for the dual porting.

U13 is the Am7990. It uses U23 (Am7960) as the Manchester encoder/decoder and media interface to the TXD and RXD lines. This circuitry is shown in Figure 7-3.

U14 and U15 (99C88) are the memories themselves. These may also be expanded very easily if required. The address and data lines are shown in Figure 7-3.

U19 (LS112), U20 (LS08), U21 (LS32), and U24 (LS125) provide the Ready line conditions appropriate to the Bus timing of valid data to the main CPU. This circuitry is shown in Figure 7-4. The clock circuit is also shown in Figure 7-4. The 16 MHz clock is a crystal oscilator. Its fundamental use is to drive the Am7960 (U23) directly. The oscillator frequency is divided by two to drive the prelatch (U17) and the Am29PL141 (U16). Figure 7-4 also shows the RESET circuitry which sends a CPU bus reset signal to the Am29PL141, Am7990, and the Am7960.

This design may also be used to not only provide isolation to the DMA but also to provide a bus translation service for an 8 bit CPU. The 16 bit I/O transfer needed by the Am7990 write and read can be accomplished if, on the data bus side, the D8-15 LS245s are replaced with LS373. In memory operation, the LS373s are made transparent but in I/O, the high byte is written first and then as the low byte is written, both are enabled into the 7990. On a read, the full 16 bit transfer takes place and the low byte is read immediately. The next operation reads location I/O + 2 for the D8-15 value.

In this application, memory is treated as memory and the 7990 is treated as I/O space. The 2 port memory is used by the CPU to set up ring descriptors as well as the rings themselves. Packet buffers can be assembled and disassembled in this area under the operating system at low level drivers. 16K space is enough for 8 512-byte transmit rings and 8 512-byte receive rings. At 1 MHz data rate, that is probably more than enough. However, a 10 MHz design may require 64K DRAM to provide sufficient high speed memory bandwidth.

7.3 MICROPROGRAM

The Am29PL141 controller's major function is to process a HOLD request by the Am7990. When the Am7990 is not active, it processes normal CPU memory read/write and normal I/O read/write (Figure 7-5 shows the microprogram flow diagram).

When the Am29PL141 receives a HOLD request, it sends a HOLDA signal to the Am7990 to activate the Am7990. The HOLDA signal also goes to the BD0 pin of the CPU so that the CPU can check to see if the Am7990 is using the DMA. Only in the HOLDA path (main path) is another task allowed besides the normal path. In the HOLDA path, the CPU is allowed access until T5 of the Am7990 state machine. At that point, the memory is diverted and remains until the completion of the 7990 DMA. The Am7990 dropping Hold Request (HOLD) is what finally clears the HOLDA cycle and returns control to the Am29PL141. Branch #1 is just a normal CPU I/O read/write and branch #2 is a normal CPU memory read/write when the HOLDA is not active. Figure 7-6 is the actual microcode of the 29PL141.

Note: The 7990 cannot be slave-accessed with HOLDA valid. Therefore, any I/O request is blocked in the controller during a DMA transfer. In order to prevent a possible 48 microsecond Ready/Wait signal, HOLDA can be sampled by the CPU at the data I/O pin BD0 and when logically false, the I/O request can then be made at I/O address of 7990 + 4.

7.4 PAL DEVICE EQUATIONS

```
PAL Device #1 (U3): CPU Bus Control
                    (AmPAL16L8)

PIN

  /IORD     = 1         /MEMWE    = 11

  /IOWR     = 2         /WE       = 12
  /MEMRD    = 3         /DALI     = 13
  /MEMWR    = 4         /MEMOE    = 14
  /MEMCOMP  = 5         /READYEN  = 15
  /IOCOMP   = 6         /EDIR     = 16
  /7990EN   = 7         /EADEN    = 17
  /7990WE   = 8         /IORQ     = 18
  /WR       = 9         /MEMRQ    = 19
;

BEGIN.

  MEMRQ = MEMRD * MEMCOMP + MEMWR *
    MEMCOMP ;

  IORQ    = IORD * IOCOMP + IOWR *
     IOCOMP ;

  EADEN = MEMRQ * /7990EN + IORQ *
     /7990EN ;

  EDIR  = MEMRD + IORD ;

  READYEN = MEMRQ + IORQ ;

  WE = 7990WE * WR + MEMWE * MEMWR +
   /7990EN * MEMRQ * MEMWR ;

  MEMOE = /7990EN * MEMRD + 7990EN *
   DALI ;

END.


PAL Device #2 (U4): 7996 control
                       equations

PIN
  /IORD     = 1         /BBHE        = 11

  /IOWR     = 2         /WELB        = 12
  /HOLDA    = 3         /WEHB        = 13
  /DALI     = 4         /WE          = 14
  /IORQ     = 5         /7990READY   = 15
  /DALO     = 6         /WR          = 16
  /LA0      = 7         /DAS         = 17
  /7990EN   = 8         /7990DBDIR   = 18
  /7990BHE  = 9         /7990JDBEB   = 19
;

BEGIN.

  IF ( /HOLDA ) THE ENABLE (DAS , WR,
  7990READY ) ;

  DAS = OWWR + IORD ;

  WR = IOWR ;

  7990READY = HOLDA;

  7990DBEN = /HOLDA * IORQ + HOLDA *
  7990EN * ( DALI + DALO ) ;

  7990DBIR = /HOLDA * IORQ + HOLDA *
  DALI ;

  WELB = /LA0 * WE ;

  WEHB = WE *7990EN * 7990BHE + WE *
  /7990EN * BBHE ;

END.
```

7.5 SUMMARY

In summary, the design solves the system requirements of double buffering and DMA isolation using a minimum of parts yet retaining memory at bus bandwidth without a large number of wait states added. The 7990 is allowed full access as needed without ever seeing a slow down and the basic design has a large amount of frequency latitude for the LAN Speed.

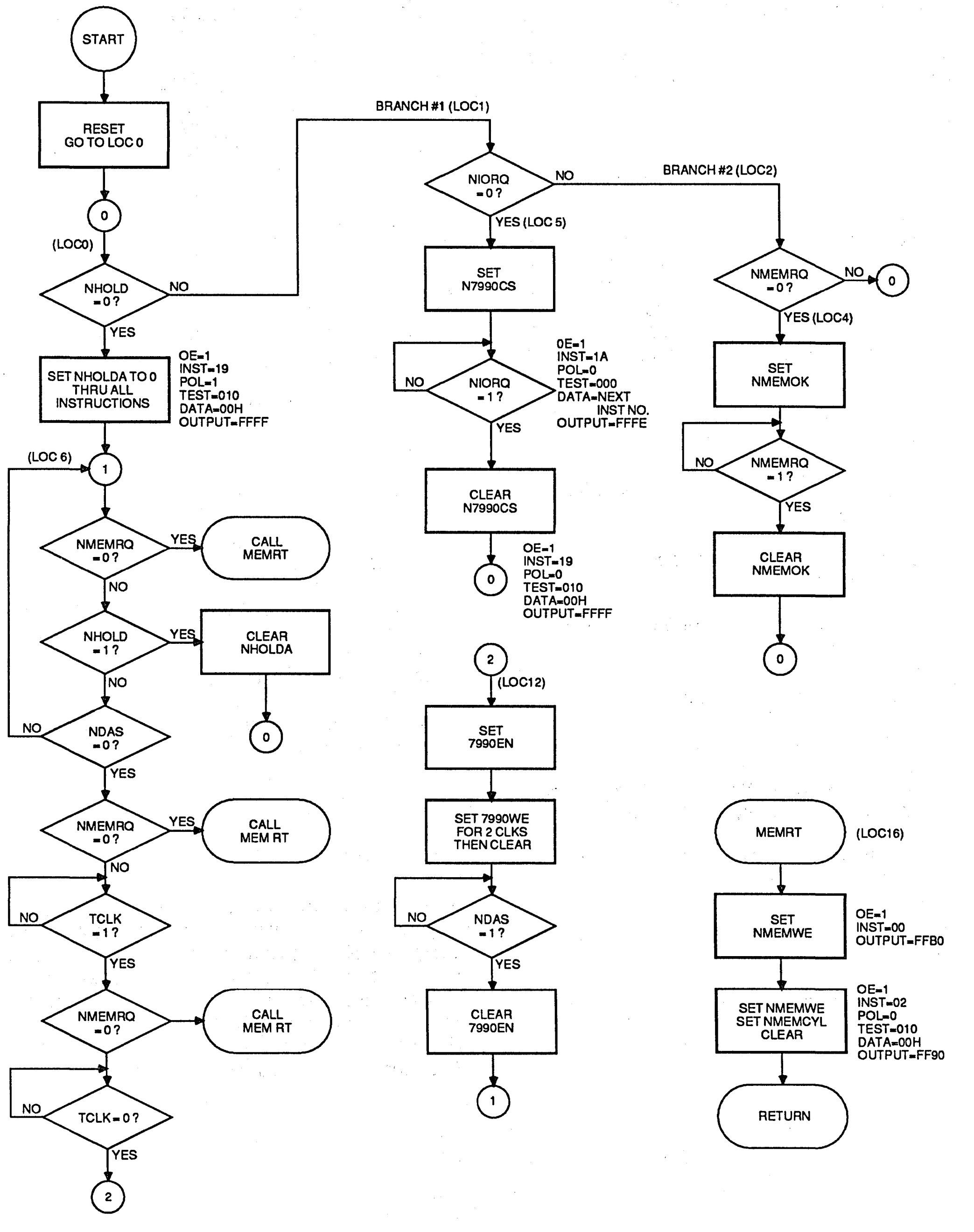


Figure 7-5. Starlan Controller Program Flow Diagram

```
DEVICE ( PL141 )

DEFAULT = 1 ;

 DEFINE
          NIORQ = T0
          NMEMRQ = T1
          NHOLD = T2
          NDAS = T3
          TCLK = T4
          VCC = CC
          N7990CS = FFFE#H
          NHOLDA = FFFD#H
          NMEMOK = FFFB#H
          N7990EN = FFF7#H
          N7990WE = FFEF#H
          NMEMCYLCLR = FFDF#H
          NMEMWE = FFBF#H
          NEXEC = FF7F#H;

DEFAULT_OUTPUT = FFFF#H;

BEGIN

EXEC :    NEXEC , IF ( NOT NHOLD ) THEN GOTO PL ( HOLDA ) ;
          NEXEC , IF ( NOT NIORQ )THEN GOTO PL ( IORQ ) ;
          NEXEC , IF ( NOT NMEMRQ ) THEN GOTO PL ( MEMRQ ) ;
          NEXEC , IF ( VCC ) THEN GOTO PL ( EXEC ) ;

MEMRQ :   NMEMOK , IF ( NMEMRQ ) THEN GOTO PL (EXEC) ELSE WAIT;

IORQ :    N7990CS , IF ( NIORQ ) THEN GOTO PL (EXEC) ELSE WAIT;

HOLDA :   NHOLDA , IF ( NOT NMEMRQ ) THEN CALL PL ( MEM ) ;
          NHOLDA , IF ( NHOLD ) THEN GOTO PL ( EXEC ) ;
          NHOLDA , IF ( NDAS ) THEN GOTO PL ( HOLDA ) ;
          NHOLDA , IF ( NOT MEMRQ ) THEN CALL PL ( MEM ) ;
          NHOLDA , IF ( NOT TCLK ) THEN GOTO PL ( HOLDA1 ) ELSE WAIT;
HOLDA1 :  NHOLDA , IF ( NOT NMEMRQ ) THEN CALL PL ( MEM ) ;
          NHOLDA , IF ( TCLK ) THEN GOTO PL (HOLDA2) ELSE WAIT;
HOLDA2 :  FFF5#H , CONTINUE ;
          FFE5#H , CONTINUE ;
          FFE5#H , CONTINUE ;
          FFF5#H , IF ( NDAS ) THEN GOTO PL (HOLDA) ELSE WAIT;

MEM :     FFBB#H , CONTINUE ;
          FF9B#H , CONTINUE ;
          NHOLDA , IF (VCC ) THEN RET ;
          .ORG 63#D
          EXEC   , IF ( VCC ) THEN GOTO PL ( EXEC ) ;

END.
```

Figure 7-6. Starlan Controller Source Program Listing

CHAPTER 8

IBM PC-SSR INTERFACE USING an Am29PL141 CONTROLLER

8.1 THE DESIGN PROBLEM

This application note describes the use of an Am29PL141 controller and an IBM PC or other computer to run diagnostics tests on a device containing a Serial Shadow Register (SSR). The SSR is a special serial in, serial out register built into devices to facilitate diagnostic testing.

To test a complex state machine or a microcoded CPU engine in a manufacturing environment is a complex task. The conventional method has been to use a "Bed of Nails" consisting of probes making contact to the printed circuit board (PCB) in specially assigned places. A master program in the tester provides a stimulus and then checks the response. These Bed of Nails test fixtures are complex and costly and worst of all, are mechanically interlinked in such a manner that a simple movement of an IC on the PCB may cause a whole fixture to be scrapped or at least reworked. Each fixture may cost up to $10,000 and requires an expensive tester to control it.

8.2 SSR FUNCTIONAL DESCRIPTION

AMD in conjunction with MMI pioneered a concept called Serial Shadow Register (SSR). Typically in state machines or microcoded CPUs, data is latched into a register on one clock to drive the logic and on the next clock, the result is latched into a destination register. The SSR is an additional diagnostic register linked to the main device register. It can load new information into the device register and capture the response of the device. Various test inputs are entered into the SSR serially from a computer with the assistance of a controller (FPC). The device executes the input and returns the result into the SSR. The controller serially extracts the result from the SSR and transfers it to the computer. The computer then checks the response with the known correct response. Using serial input and output to the SSR keeps the pin count down.

SSRs can be used in all phases of the product testing because they are a part of the device and therefore available at all times. They can be used in engineering to debug the design, in manufacturing to test each device for compliance, and, in field service, to diagnose faulty operation either at the customer site or at the repair depot.

The controller's task is to convert the parallel IBM PC bus, or equivalent, to a serial data stream to be shifted into the SSRs. The SSR is driven from a relatively inexpensive Personal Computer (PC) that has a file of many stimulus patterns and the corresponding response patterns. In operation, the PC writes the first byte of the stimulus pattern to the SSR controller, in parallel (See Figure 8-1). The controller then shifts the pattern out to the SSR (stimulus chain, N1 bits long, in the device to be tested) and informs the PC through the "DONE" flag that it can accept more parallel data. This interchange goes on until the stimulus chain in the device being tested is full (N1 bits shifted).

Then the PC changes the state from "SHIFT OUT" to "EXECUTE" and the controller generates the necessary clocks to compute the response. The FPC then loads the first byte from the SSR response chain into the PC read register and informs the PC. The PC now examines, on a bit for bit basis, the response pattern just read with the known good response pattern in its file. Any errors can be flagged and output to the printer or displayed on the CRT screen, thereby helping pinpoint the exact area of fault. This byte compare goes on until the entire response chain of N2 bits has been examined. This whole sequence can be done as many times as necessary to fully check out the PCB at the bit level.

8.3 ARCHITECTURE

The heart of the operation is the AMD Am29PL141, Fuse Programmable Controller. It takes care of controlling the D clock, P clock, and Mode of the serial chain. It shifts the 8 bits out and then specifies "DONE". It monitors the "SHIFT OUT" and "Go" control bits for status change. Figure 8-2 gives pin level detail of the blocks or units shown in the the block diagram. Figure 8-3 shows the user interface circuitry.

U1 serves as an address decode PAL Device whose equations are given later. U2 is just a data bus buffer to keep the loading to 1 LS TTL load.

U3 and U4 form the handshake flip flops for the Am29PL141 controller to the PC interface. U3 controls the state of the controller. The "MODE SHIFT OUT" is for the loading of stimulus into the SSR. When that is through, the PC clears this flip flop and sets the "GO" bit which is a trigger for the Am29PL141 (U11) to set the SSR Mode line to 1, issue 2 P CLKS and then 1 D clock.

U5 is the input shift register used for the parallel to serial conversion. U6 is the number of bits to shift this time. It must always be written prior to the writing of U5. Writing to U5 sets the "NEW DATA" flag flip flop U3 to tell the Am29PL141 to shift out. U8 is just an interface to the 6 pin connector to isolate the board from the device under test.

U7 is the serial to parallel converter. When read, it puts data on the internal data bus (BDO-7). When U8 is read, the "READ DATA" flip flop is set to inform the Am29PL141 to shift in 8 new bits for interrogation.

U10 is a register to pre-synchronize the asynchronous signals to guarantee set up times for the Am29PL141. This prevents any metastability problem for the Am29PL141.

U11 is the Am29PL141 programmable controller (See Figure 8-2). It provides the timing and control signals to operate the device. It controls the parallel flow of data between the IBM PC and the serial/parallel convertors, U5 and U7. It controls the serial flow of data between U5 and U7 and the device under test. One of the outputs EXEC is a status signal. It is set when the microprogram is in the EXEC loop. This output can be monitored by a diagnostic probe if desired.

The typical user interface circuitry is shown in Figure 8-3. It represents the minimum that needs to be done on the board under test.

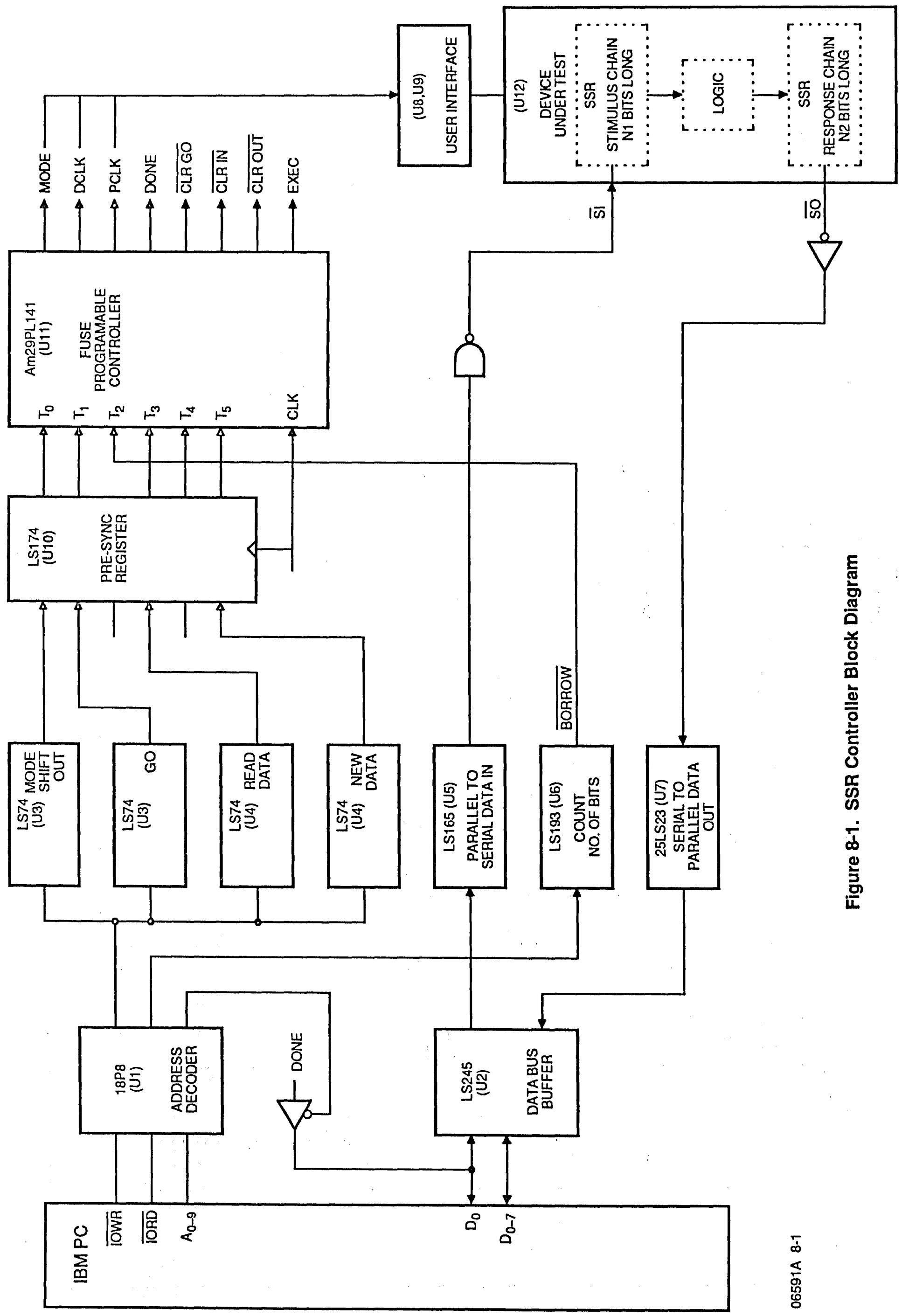
The flow diagram of the microcode is shown in Figure 8-4. The assembler source code for the program is given in Figure 8-5. The PAL Device equations are given in Section 8.4.

8.4 PAL DEVICE EQUATIONS FOR 18P8

```
(in PLPL)

Device (Am18P8)

PIN
 /IOWR = 1 /IORD = 2  [A9:A3] = [3:9]
        GND = 10 . A2 = 11

 A1 = 12  A0 = 13  /COUNTLD = 19  /DBEN = 18
        /DATAIN = 17

 DATAOUT = 16  /STATUS WR = 15  STATUS
        RD = 14 ;

BEGIN

 STATUS RD = IORD * A9 * A8 * A7 * A6 * A5 *
        A4 * /A3 * /A2 * /A1 * /A0 ;

 STATUS WR = IOWR * A9 * A8 * A7 * A6 * A5 *
        A4 * /A3 * /A2 * /A1 * /A0 ;

 DATA OUT = IORD * A9 * A8 * A7 * A6 * A5 *
        A4 * /A3 * /A2 * /A1 * A0 ;

 DATA IN = IOWR * A9 * A8 * A7 * A6 * A5 * A4
        * /A3 * /A2 * /A1 * A0 ;

 DBEN = ( IORD + IOWR ) * A9 * A8 * A7 * A6
        * A5 * A4 * /A3 * /A2 ;

 COUNT LD = IOWR * A9 * A8 * A7 * A6 * A5 *
        A4 * /A3 * /A2 *A1 * A0 ;

END.
```

8.5 SUMMARY

This design represents a minimum number of ICs to do an interlinked control job for the SSR chain. The objective is to show that a low cost testing alternative is available to diagnose state machines and microprocessors.



Am29PL141 (U11) FUSE PROGRAMABLE CONTROLLER
T0 T1 T2 T3 T4 T5 CLK
MODE
DCLK
PCLK
DONE
CLR GO
CLR IN
CLR OUT
EXEC
(U8,U9) USER INTERFACE
(U12) DEVICE UNDER TEST
SSR STIMULUS CHAIN N1 BITS LONG
LOGIC
SSR RESPONSE CHAIN N2 BITS LONG
SI
SO
LS174 (U10) PRE-SYNC REGISTER
LS74 (U3) MODE SHIFT OUT
LS74 (U3) GO
LS74 (U4) READ DATA
LS74 (U4) NEW DATA
LS165 (U5) PARALLEL TO SERIAL DATA IN
LS193 (U6) COUNT NO. OF BITS
BORROW
25LS23 (U7) SERIAL TO PARALLEL DATA OUT
18P8 (U1) ADDRESS DECODER
DONE
LS245 (U2) DATA BUS BUFFER
IBM PC
IOWR
IORD
A0–9
D0
D0–7
06591A 8-1

Figure 8-1. SSR Controller Block Diagram

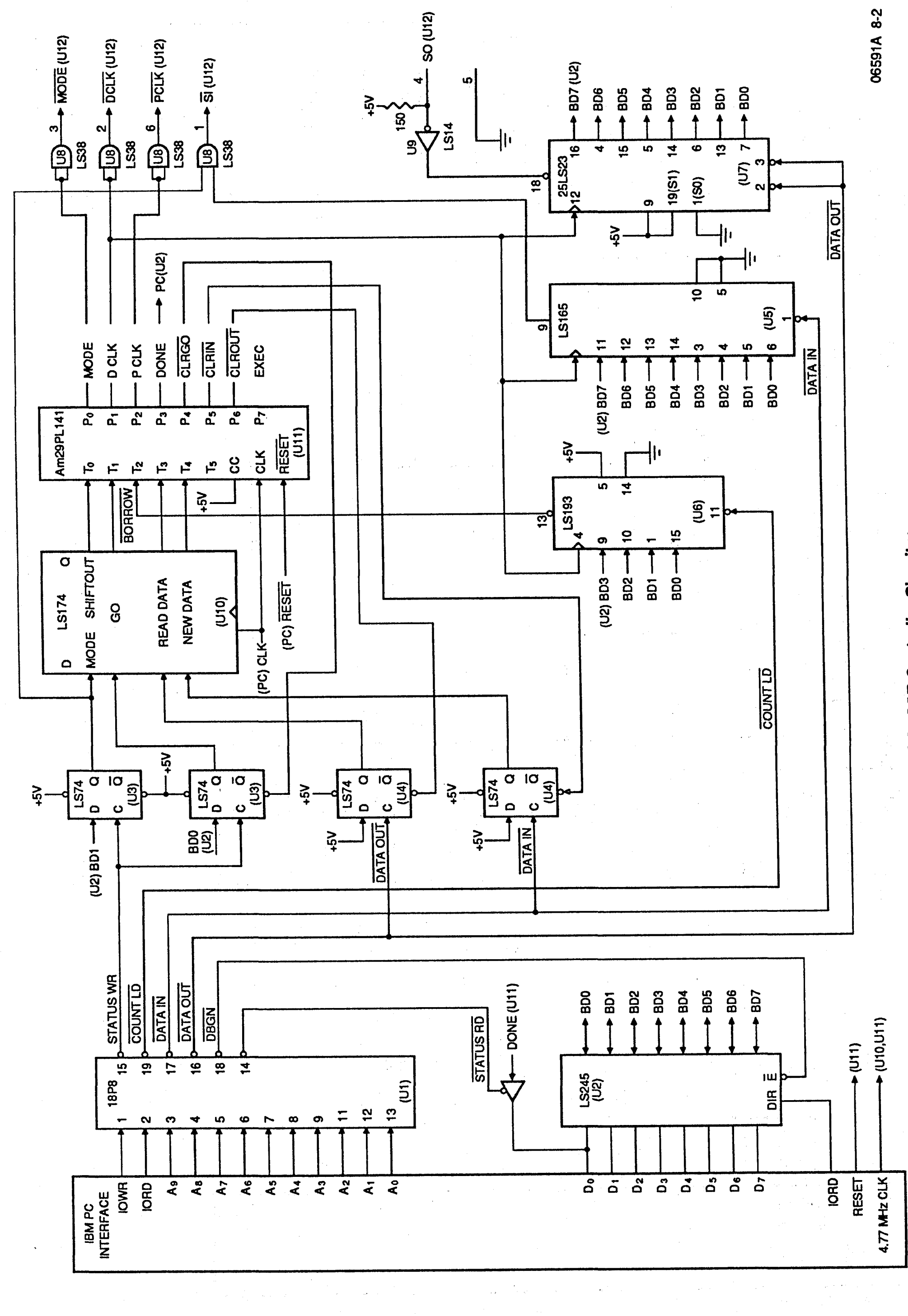
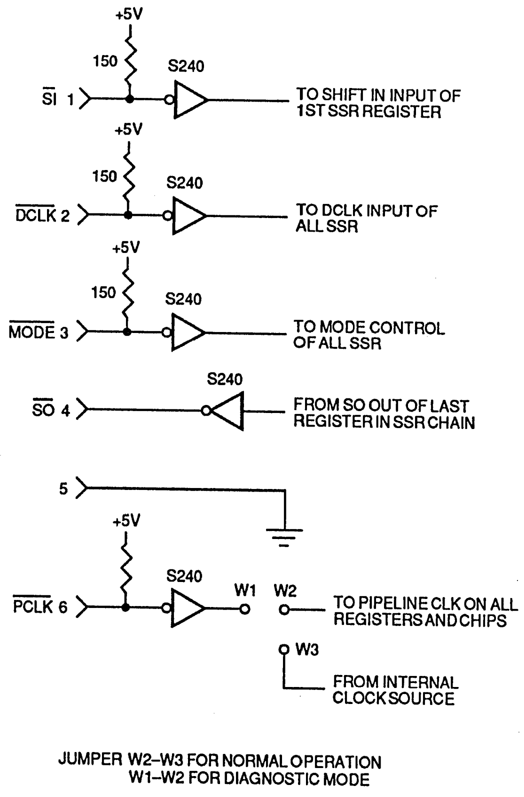
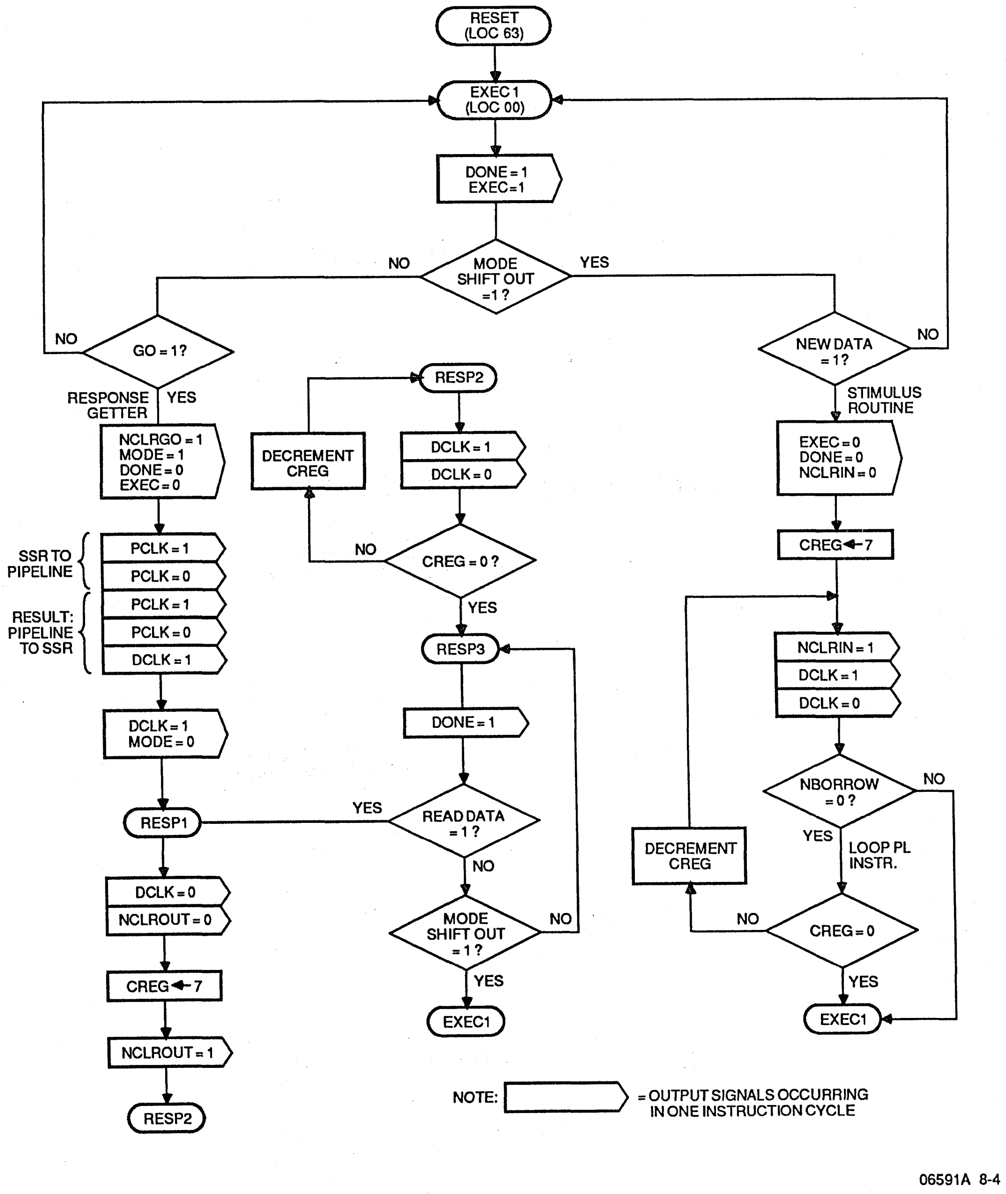


Figure 8-2. SSR Controller Circuitry


+5V
150
S240
SI 1
TO SHIFT IN INPUT OF 1ST SSR REGISTER
+5V
150
S240
DCLK 2
TO DCLK INPUT OF ALL SSR
+5V
150
S240
MODE 3
TO MODE CONTROL OF ALL SSR
S240
SO 4
FROM SO OUT OF LAST REGISTER IN SSR CHAIN
5
+5V
S240
PCLK 6
W1
W2
TO PIPELINE CLK ON ALL REGISTERS AND CHIPS
W3
FROM INTERNAL CLOCK SOURCE
JUMPER W2–W3 FOR NORMAL OPERATION
W1–W2 FOR DIAGNOSTIC MODE

06591A 8-3

Figure 8-3. User Equipment Interface Circuitry



06591A 8-4

Figure 8-4. SSR Controller Program Flow Diagram

```
DEVICE ( PL141 )

DEFAULT = 1 ;

 DEFINE
         MODE_SHIFTOUT = T0
         GO = T1
         NBORROW = T2
         READDATA = T3
         NEWDATA = T4
         VCC = CC
         MODE = 0071#H
         DCLK = 0072#H
         PCLK = 0074#H
         DONE = 0078#H
         NCLRGO = 0060#H
         NCLRIN = 0050#H
         NCLROUT = 0030#H
         EXEC = 00F0#H ;

DEFAULT_OUTPUT = 0070#H ;

BEGIN

EXEC1 :   EXEC + DONE , IF ( MODE_SHIFTOUT ) THEN GOTO PL ( EXEC2 ) ;
         EXEC + DONE , IF ( GO ) THEN GOTO PL ( RESP ) ;
         EXEC + DONE , IF ( VCC )  THEN GOTO PL ( EXEC1 ) ;
EXEC2 :   EXEC + DONE ,IF ( NOT NEWDATA ) THEN GOTO PL ( EXEC1 ) ;


STIM :    NCLRIN , IF ( VCC ) THEN LOAD PL ( 07#H ) ;
STIM1 :   DCLK , CONTINUE ;
             , CONTINUE ;
             , IF ( NOT NBORROW ) THEN GOTO PL ( STIM2 ) ;
             , WHILE ( CREG < > 0 ) LOOP TO PL ( STIM1 ) ;
STIM2 :   EXEC + DONE , IF ( VCC ) THEN GOTO PL ( EXEC1 ) ;


RESP :    NCLRGO + MODE , CONTINUE ;
         MODE + PCLK , CONTINUE ;
         MODE , CONTINUE ;
         MODE + PCLK , CONTINUE ;
         MODE + DCLK , CONTINUE ;
RESP1 :   NCLROUT , IF ( VCC ) THEN LOAD PL ( 07#H ) ;
RESP2 :   DCLK , CONTINUE ;
               , CONTINUE ;
                , WHILE ( CREG < > 0 ) LOOP TO PL ( RESP2 ) ;
RESP3 :        , IF ( READDATA )  THEN GOTO PL ( RESP1 ) ;
              , IF ( NOT MODE_SHIFTOUT ) THEN GOTO PL ( RESP3 ) ;
         DONE + EXEC , IF ( VCC ) THEN GOTO PL ( EXEC1 ) ;
         .ORG 63#D
         DONE + EXEC , IF ( VCC ) THEN GOTO PL ( EXEC1 ) ;
END.
```

Figure 8-5. SSR Controller Source Program Listing

CHAPTER 9

QUARTER-INCH TAPE CARTRIDGE and SMALL COMPUTER SYSTEM INTERFACE CONTROLLER USING Am29PL141

9.1 OVERVIEW

This application note describes the use of the Am29PL141 Fuse Programmable Controller (FPC), to control both the Quarter Inch Tape Cartridges via the QIC-02 industry standard and the Small Computer Systems Interface (SCSI), also an industry standard as defined by ANSI X3T9.2 subcommittee. This controller functions as the "Host" to the QIC-02 interface and as an "Initiator" to a SCSI system. This design provides the capability to transfer data in both directions, between the SCSI bus and QIC-02.

A practical use is to back up data on a hard disk (SCSI) via Tape (QIC-02). The FPC functions as a high performance (50 ns instruction cycle time) I/O Controller which is slave to the system CPU (host). It supports the maximum data rates of both interfaces (1.5 Mbyte/Sec. asynchronous mode for SCSI). This design uses the 80188 microprocessor, but any host microprocessor could be interfaced to the FPC in a similar fashion. The QIC-02 standard interface is fully supported and the single initiator multiple target mode is supported for SCSI. Although this application does not include using all advanced features of SCSI, the section on "Advanced Features of SCSI" does provide insight into upgrading this design.

In the following discussions, it is assumed that the reader is somewhat familiar with the 80188, FPC, QIC-02, and SCSI. An overview of the QIC- 02 and SCSI is given below. A discussion of the QIC-02 and SCSI, including timing diagrams, has been included in Appendix B.

9.1.1 QIC-02 Overview

QIC-02 is an industry standard which defines the interface between a host system and Quarter Inch Cartridge Tape Drives. Read/write commands, status and, of course, data are transmitted over this interface, as depicted in Figure 9-1. The bus and control signals between QIC-02 and host are all standard TTL levels. Timing diagrams for this interface are given in Appendix B. This interface handshake timing is duplicated for the host side by the FPC and two AmPAL22V10s.

The interface lines are used as follows:

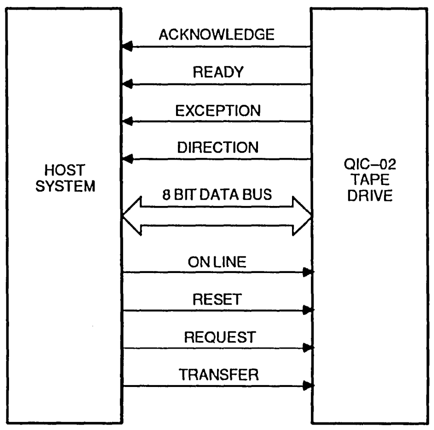
ACKNOWLEDGE (ACK) is used with Transfer to transfer data across the interface.

READY (RDY) indicates that the tape drive can accept a command. It is used to handshake the command across the interface. In the write mode, READY indicates that the drive's internal buffer is empty and ready to receive new data. In the read mode, READY indicates the drive buffer can now be accessed by the host.

EXCEPTION (EXP) alerts the host that the execution of a command has been terminated. This may be a normal completion or an interrupt due to a fault (hard errors, write protected, etc.). The response by the host must be READ STATUS.

DIRECTION (DIRC) indicates direction of data flow. This signal is used to enable/disable the data bus transceivers in the HOST.

ON-LINE signal is deasserted at the beginning of a read (from tape) or write (to tape) operation.



06591A 9-1

Figure 9-1. QIC-02 Interface

RESET initializes the tape drive. The drive repositions the heads to track zero.

REQUEST indicates that a command is on the data bus.

TRANSFER is used with ACKNOWLEDGE to handshake data over the bus; see timing diagram.

9.1.2 SCSI Overview

Small Computer Systems Interface (SCSI) is a disk controller standard developed by the ANSI X3T9.2 subcommittee. SCSI defines an 8-bit parallel bi-directional data bus with parity, plus nine control lines. The SCSI protocol allows single or multiple host computers (initiators) to share multiple peripherals (targets, i.e. hard disk, floppy disks, etc.). Up to eight daisy chained devices can reside on the SCSI bus, with data transfer rates of 4 Mbytes/sec. synchronous and 1.5 Mbyte/sec. asynchronous. The timing diagrams are given in Appendix B.

The following is a summary of the interface signals:

I/O is driven by a target to control the direction of data movement. True indicates input to the initiator.

MSG is driven by a target to indicate "Message Phase". When MSG is asserted, REQ (Request) is also asserted by the target for transfer of data byte indicating the end of the operational phase ("Message").

REQ is asserted by target to indicate that a data byte is to be transferred on the data bus. Data byte is transferred via handshake with ACK (Acknowledge).

ATN (Attention) is driven by an initiator to indicate to target an "attention" condition.

An initiator uses SEL along with asserting the appropriate data (address) bits (0-7) to select a target. Select line is deasserted after the target asserts BSY to acknowledge selection.

RST (Reset) is a pulse asserted by the initiator to stop the target's present operation and return same to idle condition.

Data bus and control signals require open collector drivers capable of sinking 48 mA each to support SCSI mode of multiple initiators with multiple targets. SCSI provides for either single ended (6 meter max. cable length) transmission or differential (up to 25 meters).

9.2 FUNCTIONAL DESCRIPTION

Figure 9-2 shows the block diagram of the Am29PL141 (FPC) QIC-02 and SCSI Controller. This controller functions as a "Host" to the QIC-02

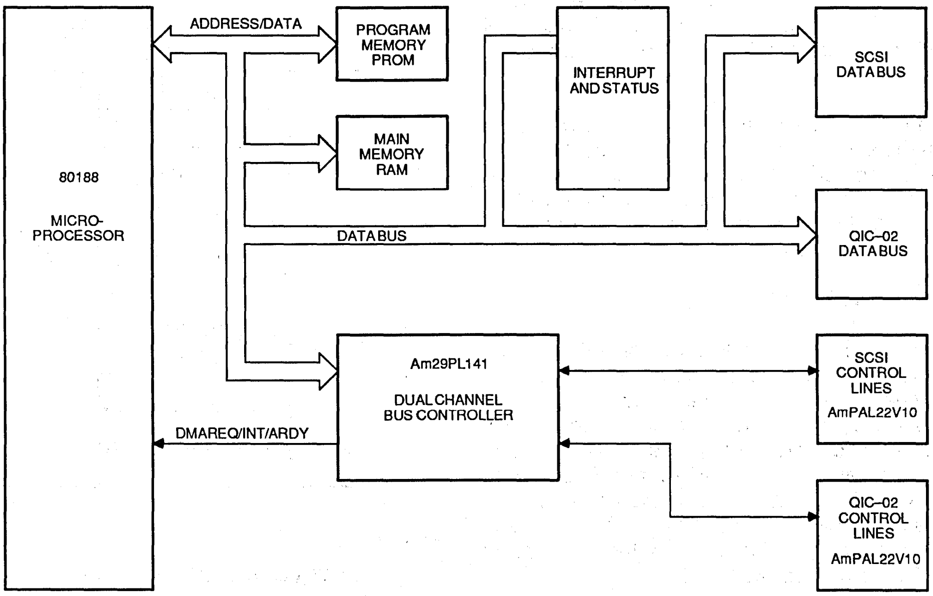


Figure 9-2. AmPL141 QIC-02 and SCSI Controller Block Diagram 06591A 9-2

interface and as an "Initiator" to a SCSI system. This design is composed of three main functional blocks: Microprocessing Unit, Dual Channel Bus Controller, I/O Bus Interface.

The Microprocessing Unit is a straightforward design centered around the 80188 microprocessor which provides system level control to the FPC through commands issued over its 8-bit data bus and with feedback from the FPC via DMA requests, interrupt, wait state insertion asynchronous ready (ARDY), Interrupt Register, and a Status Register.

The heart of the Dual Channel Bus Controller is the Fuse Programmable Controller (Am29PL141) which generates and monitors interface control signals for both I/O bus interfaces (QIC-02 and SCSI). The FPC is slave to the 80188, and controls the transfers of commands, status, and data to/from both I/O interfaces via single byte DMA transfers to/from Main Memory. Interleaved single byte transfer to/from both I/O devices is provided. This approach supports maximum rates for both I/O channels.

The I/O Bus Interface provides single-ended drive for both I/O channels (48 mA per line). Open collector drivers are required for all SCSI generated control signals; however, standard (Am29800 family) buffers and transceivers are satisfactory for the QIC-02 and SCSI data bus.

Each of these functional blocks are now described in detail.

9.2.1 80188 Microprocessing Unit

The Microprocessing Unit in this design performs all of the high level system control and application functions required when interfacing to tape and disk. These functions include system and application programs, direct memory access (DMA) controllers, timers, interrupt controllers and chip select decoders. The 80188 High Integration Microprocessor was chosen for this design because all of the above functions except the programs and associated memory are contained in a single chip. The 80188 provides two DMA channels, three programmable timers, a programmable interrupt controller and a programmable chip select decoder. In this design, both DMA channels, one timer, one external interrupt and four peripheral chip selects (PCS1-4) are dedicated to the SCSI and QIC-02 interfaces.

In order to configure the 80188 for this application, certain operations must be performed prior to executing any instructions which will access the SCSI or QIC-02 interfaces. After reset, only the Upper Memory Chip Select (UMCS) is active in order to allow the 80188 to begin execution at location FFFOH. At this time, UMCS is programmed for a block size of 1K bytes. To allow full use of the Am27512, 64KX8 EPROM, the UMCS register should be programmed with the value F03DH. This sets UCMS for a 64K byte block size, inserts one automatic wait state and ignores external RDY in the range FOOOOH to FFFFFH. Likewise, the Lower Memory Chip Select (LMCS) must be programmed via the LMCS register.

Programming this register with the value O1FCH selects an 8K byte block size, zero automatic wait states and ignores external RDY in order to take full advantage of the Am99C88-70, 70ns 8KX8 CMOS Static RAM. Finally, the Peripheral Chip Selects (PCSM) must be configured. Four of these PCSM are used to select the SCSI and QIC-02 interfaces. The PCSM are configured via MPCS and PACS control registers. The MPCS register is programmed with the value, 84B8H, which places the PCSM in I/O address space, enables all seven PCS lines, inserts no automatic wait state, and uses external RDY. This value also configures the Mid-Range Memory Selects (MCSM) for 8 Kbyte block size The PACS register is programmed with the value 0078H. This places the PCS block at I/O address OOOOH, inserts no automatic wait states, and uses external RDY.

With the hardware now configured, the 80188 is prepared to run applications utilizing the SCSI and QIC-02 interfaces. An example of a simple application is shown in Figure 9-3. This application selects DISKO on the SCSI and reads 2000 bytes into a data buffer. It then rewinds the tape on the QIC-02 and writes the data buffer onto the tape. As can be seen in Figure 9-3, there are several support routines which perform the actual communication with the SCSI/QIC-02 interface.

SOFTWARE SUPPORT ROUTINES:

FPC Control. This procedure outputs a function and a code to the FPC command register. It also reinitializes the watchdog timer via another procedure (WD.Init) not described here. The watchdog timer is used to reset the Am29PL141 in the event that a device on either the SCSI or QIC-02 fails to complete the proper handshake and locks up the bus of 80188.

SCSI–Init. This procedure uses the FPC Control routine to assert and deassert the SCSI RST signal in order to initialize the SCSI interface.

QIC2–Init. This procedure asserts and deasserts the QIC-02 RESET signal to initialize the interface.

```
PROGRAM MAIN;

/* THIS PROGRAM IS AN EXAMPLE OF THE ROUTINES NECESSARY TO
UTILIZE THE SCSI/QIC-Ø2 INTERFACE.  EACH ROUTINE IS DESCRIBED IN
THE ACCOMPANYING TEXT.  THE MAIN PROGRAM PERFORMS THE SIMPLE
OPERATIONS OF READING A MULTI-SECTOR BUFFER, REWINDING THE TAPE
AND WRITING THAT BUFFER TO THE TAPE. */

    CONST
        DISKØ = 1; /* DISK ADDRESS ON THE SCSI BUS */

        DATN =
        DRST =
        INT1 =
        DTREQ =
        TPONL =
        TRINT =
        TPRST =
        DACK =

        SET =
        RESET = Ø; /* CONTROL CODE FOR RESET OPERATION */

        FPC_COMMAND = Ø; /* FPC COMMAND REGISTER ADDRESS */
        SCSI = 128; /* SCSI DATA PORT ADDRESS */
        TAPE = 256; /* QIC-Ø2 DATA PORT ADDRESS */
        ISR = 384;  /* INTERRUPT STATUS REGISTER ADDRESS */
        STAT = 512; /* STATUS BUFFER ADDRESS */

        READ_COMMAND = BYTE [ 8, /* READ COMMAND CODE */
                              Ø, /* LUN Ø, HEAD Ø , TRACK Ø,
                                    SECTOR Ø */
                              Ø,
                              Ø,
                              4, /* FOUR BLOCKS OF 512 TO BE READ */
                              Ø] /* ENABLE RETRIES AND ERROR CORRECTION */

        CHANØ = Ø; /* DMA CHANNEL INDICATORS */
        CHAN1 = 1;

        EOI = 34 + 6528Ø; /* EOI REGISTER OFFSET PLUS CONTROL
                             BLOCK BASE ADDRESS */

        INT1_IS = 13; /* INTERRUPT 1 IDENTIFIER TO RESET
                         IN-SERVICE BIT IN EOI REGISTER */
        DMAØ_IS = 1Ø; /* DMA CHANNEL Ø IDENTIFIER TO RESET
                         IN-SERVICE BIT IN EOI REGISTER */
        DMA1_IS = 11; /* DITTO FOR DMA CHANNEL 1 */

    VAR
        SCSI_FLAG, TAPE_FLAG, COUNT, I : INTEGER;
        DATA_BUFFER [2ØØØ] : BYTE;
        STATUS_BUFFER [2] : BYTE;

    PROCEDURE FPC_CONTROL (FUNC, CODE);
        CONST CMDMASK = BYTE 8;
        VAR CMDACK : BYTE;

        BEGIN
            WD_INIT; /* INITIALIZE WATCHDOG TIMER */
            CMDACK := 8;
            DO WHILE CMDACK <> Ø
                CMDACK := CMDMASK AND INPUT(ISR);
            OUTPUT(FUNC*8+CODE, FPC_COMMAND);
        END;
```

Figure 9-3. SCSI/QIC-02 Driver Example (Sheet 1 of 3)

```
PROCEDURE SCSI_INIT;
    BEGIN
         FPC_CONTROL (SET, DRST); /* ASSERT SCSI RST */
         DELAY (100); /* WAIT 100 USECS */
         FPC_CONTROL (RESET, DRST); /* DEASSERT SCSI RST */
    END;

PROCEDURE QIC2_INIT;
    BEGIN
         FPC_CONTROL (SET, TPRST); /* ASSERT QIC-02 RESET */
         DELAY (100); /* WAIT 100 USECS */
         FPC_CONTROL (RESET, TPRST); /* DEASSERT RESET */
    END;

PROCEDURE D_SELECT (IDENT);
    BEGIN
         WD_INIT; /* INITIALIZE WATCHDOG TIMER */
         OUTPUT (IDENT, SCSI); /* OUTPUT THE IDENTIFIER TO THE
                                  SCSI PORT */
    END;

PROCEDURE T_CMD (COMMAND);
    BEGIN
         WD_INIT;
         OUTPUT (COMMAND, TAPE);
    END;

PROCEDURE D_XFER (FUNC, BUFFER, COUNT);
    BEGIN
         IF FUNC = READ THEN
              DMA_SETUP (SCSI, BUFFER, COUNT, CHAN0);
         ELSE
              DMA_SETUP (BUFFER, SCSI, COUNT, CHAN0);
         WD_INIT;
         DMA_START (CHAN0);
    END;

PROCEDURE T_READ (BUFFER, COUNT);
    BEGIN
         DMA_SETUP (TAPE, BUFFER, COUNT, CHAN1);
         WD_INIT;
         DMA_START (CHAN1);
    END;

PROCEDURE T_WRITE (BUFFER, COUNT);
    BEGIN
         DMA_SETUP (BUFFER, TAPE, COUNT, CHAN1);
         WD_INIT;
         DMA_START (CHAN1);
    END;

PROCEDURE FPC_ISR;
    VAR INTSTAT : BYTE;
    BEGIN
         INTSTAT := INPUT (ISR); /* GET THE INTERRUPT STATUS */
         IF INTSTAT AND TRDY_MASK THEN
              BEGIN
                   FPC_CONTROL (RESET, TRINT);
                   TAPE_FLAG := 0;
              END;
         IF INTSTAT AND SCSI_ERROR_MASK THEN
              SCSI_INIT;
         IF INTSTAT AND TAPE_ERROR_MASK THEN
              QIC2_INIT;
         FPC_CONTROL (RESET, INT1);
         OUTPUT (INT1_IS, EOI);
    END;
```

Figure 9-3. SCSI/QIC-02 Driver Example (Sheet 2 of 3)

```
     PROCEDURE DMA0_ISR;
         BEGIN
              SCSI_FLAG := -;
              OUTPUT (DMA0_IS, EOI);
         END;

     PROCEDURE DMA1_ISR;
         BEGIN
              TAPE_FLAG := 0;
              OUTPUT (DMA1_IS, EOI);
         END;

     BEGIN  /* MAIN PROGRAM BODY */
         SCSI_INIT;
         TAPE_INIT;
         D_DELECT (DISK0);
         SCSI_FLAG := 1; /* SHOW SCSI OPERATION IN PROGRESS */
         D_XFER (WRITE, READ_COMMAND, 6); /* SEND READ COMMAND TO DISK */
         DO WHILE SCSI_FLAG = 1
              I:= I+1; /* WASTE TIME WAITING FOR COMPLETION */
         SCSI_FLAG := 1; /* SHOW A NEW SCSI OPERATION IN PROGRESS */
         D_XFER (READ, DATA_BUFFER, 2000); /* READ 2000 BYTES */

         /* START AN OPERATION ON THE QIC-02 SIDE OF THE INTERFACE
            TO RUN IN PARALLEL WITH THE SCSI OPERATION */

         TAPE_FLAG := 1; /* SHOW QIC-02 OPERATION IN PROGRESS */
         T_CMD (REWIND); /* REWIND THE TAPE */
         FPC_CONTROL (SET, TRINT); /* ENABLE INTERRUPT ON TAPE RDY */
         DO WHILE TAPE_FLAG = OR SCSI_FLAG = 1
              I := I+1; /* WAIT FOR THE OPERATIONS TO COMPLETE */

         /* BOTH OPERATIONS ARE NOW COMPLETE */

         SCSI_FLAG := 1;
         D_XFER (READ, STATUS_BUFFER, 2); /* GET DISK STATUS */
         DO WHILE SCSI_FLAG = 1
              I := I+1;
         IF STATUS_BUFFER [1] = GOOD_STATUS THEN
              BEGIN
                   TAPE_FLAG := 1;
                   T_CMD (WRITE); /* PUT TAPE IN WRITE MODE */
                   T_WRITE (DATA_BUFFER, 2000); /* SEND OUT THE DATA */
                   DO WHILE TAPE_FLAG = 1
                        I := I+1;
              END;
     END;
END.
```

Figure 9-3. SCSI/QIC-02 Driver Example (Sheet 3 of 3)

D–Select. This procedure outputs an eight bit select code to the SCSI interface. This process is intercepted by the Am29PL141 which performs the SELECT handshake.

T–CMD. This procedure outputs an eight bit command to the QIC-02 interface. This process is intercepted by the Am29PL141 which performs the COMMAND handshake.

D–XFER. This procedure performs all data, command and status transfers to and from the SCSI interface.

T–Read. This procedure reads data from the QIC-02 and places it in a memory data buffer.

T–Write. This procedure writes data from a memory buffer to the QIC-02.

FPC–SR. This procedure is the interrupt service routine for the Am29PL141. Upon entry it obtains the interrupt status from the FPC Interrupt Status Register (ISR). This status is examined to detect the occurrence of any errors. If any are detected, the offending interface is reinitialized. This is a very rudimentary form of error handling and is used only for purposes of this example. More elaborate error handling is possible in actual applications. Prior to exiting this procedure, the interrupt source is reset and the in-service bit in the interrupt controller is cleared.

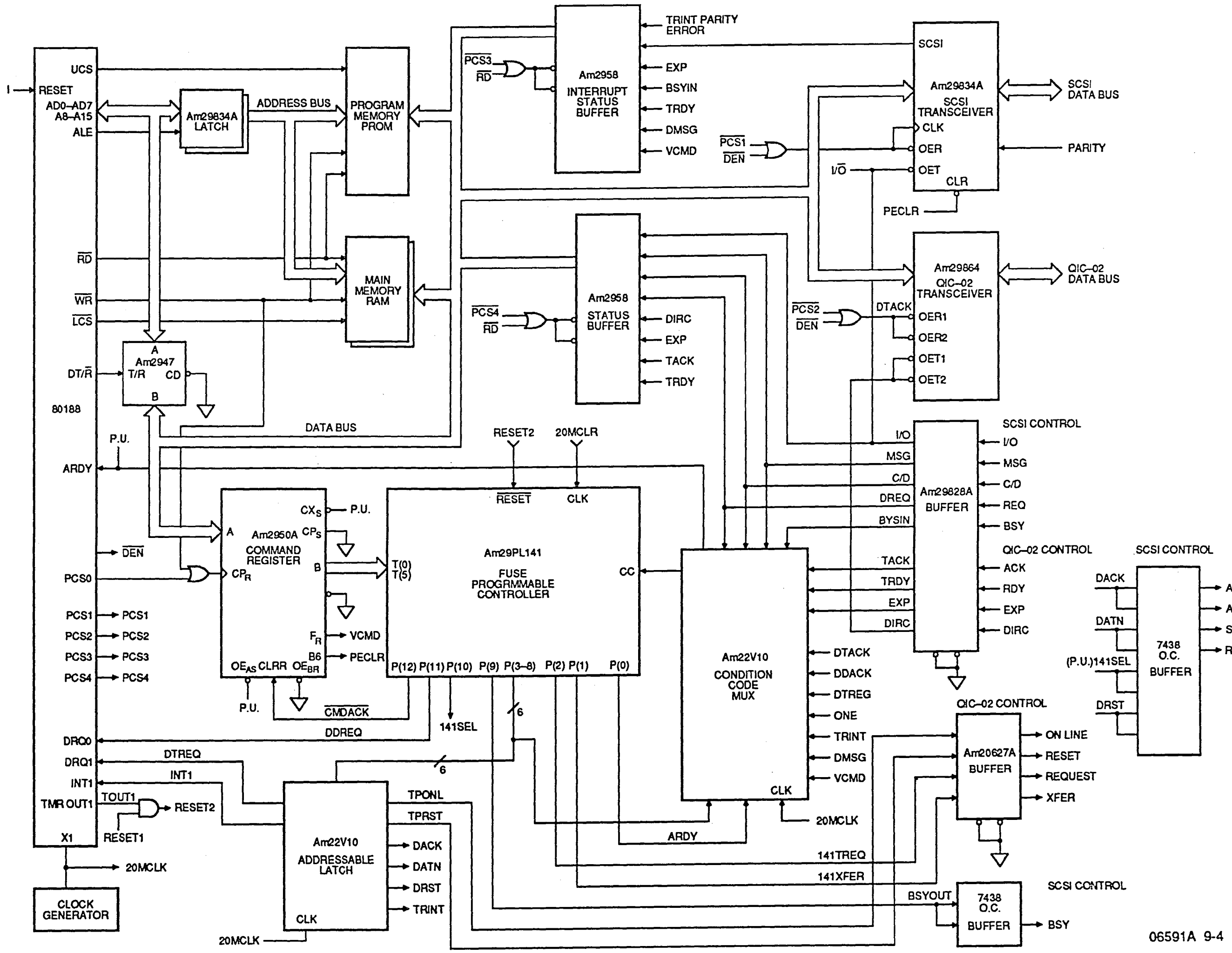


Figure 9-4. Am29PL141 QIC-02 and SCSI Controller Circuitry

DMAO–ISR, DMA1–ISR. These procedures signal the completion of data transfers to other modules by clearing the appropriate in-process flag (SCSI - FLAG, TAPE - FLAG).

9.2.2 Dual Channel Bus Controller Architecture

Refer to the complete schematic (Figure 9-4) for the Am29PL141 QIC-02 and SCSI Controller and two AmPAL22V10s. These three programmable devices provide the intelligence to control SCSI and QIC-02 interfaces, and required additional MSI control logic off-loading these tasks from the 80188 (any host CPU). In this application, the FPC can be thought of as a high speed microprocessor-like controller with twenty-nine fixed instructions, and sixteen programmable output control lines (thirteen of which are used in this application). Each instruction is executed during a single clock cycle of 50 ns. Although it can operate as a stand-

```
DEVICE   condition_code_mux   (AmPAL22V1Ø) ;

"This device selects one of many input conditions to be tested
 by the Am29PL141 and registers it in order to meet the CC setup
 time requirement.  It also collects two pieces of miscellaneous
 logic necessary to produce the ARDY and DMSG signals."

PIN
        clk = 1             vcmd = 2            trint = 3
        dtreq = 4           ddack = 5           dtack = 6
        exp = 7             trdy = 8            tack = 9
        bsyin = 1Ø          dreq = 11           c_d_bar = 13
        msg = 14            dmsg = 15           ardy = 16
        cc = 17             spare = 18          ardy_in = 19
        cc_mux_sel_3 = 2Ø                       cc_mux_sel_2 = 21
        cc_mux_sel_1 = 22                       cc_mux_sel_Ø = 23 ;

BEGIN
        ardy = ardy_in + /ddack * /ardy_in ;

        dmsg = c_d_bar * msg ;

        CASE (cc_mux_sel_3,cc_mux_sel_2,cc_mux_sel_1,cc_mux_sel_Ø)
        BEGIN
                Ø)         cc := vcmd ;
                1)         cc := ddack ;
                2)         cc := dreq ;
                3)         cc := tack ;
                4)         cc := dtack * dtreq ;
                5)         cc := dtack * /dtreq ;
                6)         cc := msg * c_d_bar + trint * trdy ;
                7)         cc := exp ;
                8)         cc := bsyin ;
                9)         cc := 1 ;
               1Ø)         cc := dtack ;
               11)         cc := dreq * ddack ;
               12)         cc := trdy ;
        END;
END.

Test_vectors

IN
        clk cc_mux_sel_3 cc_mux_sel_2 cc_mux_sel_1 cc_mux_sel_Ø
        vcmd ddack dreq tack dtack dtreq
        msg c_d_bar trint trdy exp ardy_in bsyin ;
I_O ;
OUT
        cc dmsg ardy ;
```

Figure 9-5. Condition Code MUX PAL Device Description (Sheet 1 of 2)

```
BEGIN

  cccc
  cccc
  ____
  mmmm
  uuuu         c     a
  xxxx         _     r
  ____  d   dd d t   db
  ssss vddt tt _ rt  ys    d a
c eeee cara armb ire _y    m r
l 1111 mcec cesa ndx ii  c s d
k 321Ø dkqk kqgr typ nn  c g y
______________________  _____"

Ø XXXX XXXX XXXX XXX 1X  X X H; "ardy"
Ø XXXX X1XX XXXX XXX ØX  X X L;
Ø XXXX XØXX XXXX XXX ØX  X X H;

Ø XXXX XXXX XX11 XXX XX  X H X; "dmsg"
Ø XXXX XXXX XX1Ø XXX XX  X L X;
Ø XXXX XXXX XXØ1 XXX XX  X L X;
Ø XXXX XXXX XXØØ XXX XX  X L X;

C ØØØØ ØXXX XXXX XXX XX  L X X; "cc = vcmd"
C ØØØØ 1XXX XXXX XXX XX  H X X;

C ØØØ1 XØXX XXXX XXX XX  L X X; "cc = ddack"
C ØØØ1 X1XX XXXX XXX XX  H X X;

C ØØ1Ø XXØX XXXX XXX XX  L X X; "cc = dreq"
C ØØ1Ø XX1X XXXX XXX XX  H X X;

C ØØ11 XXXØ XXXX XXX XX  L X X; "cc = tack"
C ØØ11 XXX1 XXXX XXX XX  H X X;

C Ø1ØØ XXXX 11XX XXX XX  H X X; "cc = dtack * dtreq"
C Ø1ØØ XXXX Ø1XX XXX XX  L X X;
C Ø1ØØ XXXX 1ØXX XXX XX  L X X;
C Ø1ØØ XXXX ØØXX XXX XX  L X X;

C Ø1Ø1 XXXX 11XX XXX XX  L X X; "cc = dtack * /dtreq"
C Ø1Ø1 XXXX 1ØXX XXX XX  H X X;
C Ø1Ø1 XXXX Ø1XX XXX XX  L X X;
C Ø1Ø1 XXXX ØØXX XXX XX  L X X;

C Ø11Ø XXXX XX11 ØØX XX  H X X; "cc = msg * c_d_bar + trint * trdy"
C Ø11Ø XXXX XXØØ 11X XX  H X X;
C Ø11Ø XXXX XXØØ ØØX XX  L X X;
C Ø111 XXXX XXXX XXØ XX  L X X; "cc = exp"
C Ø111 XXXX XXXX XX1 XX  H X X;

C 1ØØØ XXXX XXXX XXX XØ  L X X; "cc = bsyin"
C 1ØØØ XXXX XXXX XXX X1  H X X;

C 1ØØ1 XXXX XXXX XXX XX  H X X; "cc = 1"

C 1Ø1Ø XXXX ØXXX XXX XX  L X X; "cc = dtack"
C 1Ø1Ø XXXX 1XXX XXX XX  H X X;

C 1Ø11 X11X XXXX XXX XX  H X X; "cc = dreq * ddack"
C 1Ø11 XØ1X XXXX XXX XX  L X X;
C 1Ø11 X1ØX XXXX XXX XX  L X X;

C 11ØØ XXXX XXXX XØX XX  L X X; "cc = trdy"
C 11ØØ XXXX XXXX X1X XX  H X X;

END.
```

Figure 9-5. Condition Code MUX PAL Device Description (Sheet 2 of 2)

alone controller, the FPC has been made a slave to the 80188 uP, through the FPC test inputs (T0-T5) and the Command Register (Am2950A).

The processor (80188) writes to the Command Register which contains valid system commands (6 bits) to the FPC. During the IDLE loop of the FPC software, the FPC selects VCMD (by setting output lines P3-P6) as its CC (condition code) input through the condition code mux. If CC (VCMD) is a "pass" condition (asserted) meaning the Command Register has been updated, then the FPC branches to the instruction whose address is given by input T0-T5 (from command register). After the command has been processed, the FPC deasserts the VCMD bit (in the Command Register) and returns to the IDLE loop to check for either another command from the processor or a function required by either SCSI or QIC-02.

Checking for a VCMD and then branching to the processor's command address enables the FPC to operate asynchronous to the processor, whose bus T states (100 ns) are at one-half the FPC's clock rate and skewed in time. The seventh bit in the command register is used for the parity error latch in the SCSI transceiver, Am29834A, (upper right corner of schematic, Figure 9-4).

The Condition Code Mux (CCM) selects the appropriate input to "CC" of the FPC as defined by the FPC's output lines P3-P6. This multiplexing is not always a straight selection but does include logical combinations of input signals in some cases (see Figure 9-5, Condition Code Mux PAL Definition File).

The CCM provides two other outputs. ARDY (asynchronous ready) to the processor is asserted when instructed by the FPC and is used to lengthen the processor's bus cycle time (amount of time data remains valid on the 80188 bus) when QIC-02 or SCSI data transfer timing requires it.

The remaining output from the CCM is DMSG (Disk Message) which is an input to the Interrupt Status Buffer. This is asserted when SCSI asserts both MSG and C/D. Under this condition, the FPC generates an interrupt (INT1), through the Addressable Latch (AmPAL22V10), to the processor indicating that the Disk (SCSI) is requesting "Command" Data. The processor then reads the Interrupt Status Buffer to determine this condition (DMSG asserted). The following inputs are available to the CCM: VCMD, DTACK, and DDACK signals (generated by the processor); MSG, C/D, DREQ, and BSYIN (generated by the SCSI control bus); TACK, TRDY, and EXP (generated by the QIC-02 control bus) and TRINT from the Addressable Latch.

Since the outputs from the FPC are subject to change on an instruction by instruction basis (each clock cycle), certain signals must be latched. The AmPAL22V10 serves as an addressable latch, addressed by the FPC output lines P3-P8 (LADDR). Note that output lines P4-P6 are overlaid with the 3-bit field for the CCM. This technique frees up three spare output lines at the expense of instruction lines in the FPC. Lines P4-P6 select which of the eight latches is selected. P8 enables all latches. P7 determines set or clear of the latch, and P3 (ARESET) provides an asynchronous reset to all latches. The eight outputs from LADDR are: INT1 and DTREG to the processor; TPONL and TPRST to the QIC-02 control bus; DACK, DATN and DRST (control signals to SCSI); and TRINT (a feedback signal to the CCM). Figure 9-6 describes this PAL (LADDR).

9.2.3 Am29PL141 Microprogram

The Am29PL141 is a single-chip Fuse Programmable Controller. It is used in this application as a complex controller by programming the appropriate sequence of instructions. The available instruction set is quite rich. It includes jumps, loops, waits, and subroutine calls, which can be conditionally executed based on the test inputs (T0-T5) or CC input (all of these are used in this application). The FPC flowcharts provide the details of the FPC microprogramming used in this design.

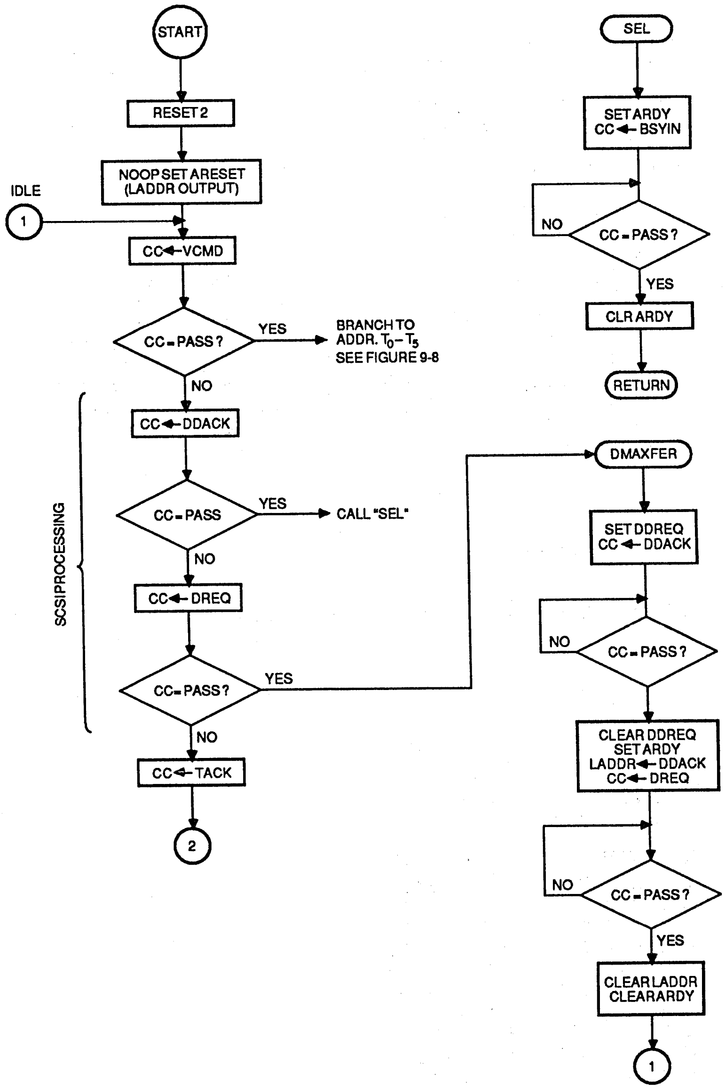
As shown in Figure 9-7, the IDLE LOOP flow diagram, the FPC continually cycles through this loop from initial power-on reset (RESET2), and jumps to one of nine routines depending on the task at hand. After completion of the task, control returns to the idle loop. RESET2 initializes the FPC to start at address sixty-three. RESET2 is generated on system power-up and when the processor's watchdog timer times out (TMROUT1). This timer is programmed to time out if the disk or tape accesses fail to complete the proper handshake in a reasonable time or the FPC locks up the bus of the 80188 because of some error condition.

The first instruction (at address 63) is a NOOP. It is used to assert ARESET (output line) to LADDR for deasserting of latches and to deassert all other output lines. The next instruction is the return/entry point into the idle loop. It selects the CCM to enable path for VCMD to CC input of FPC.

The next state is the first condition test. If CC is a PASS condition, there is a valid command (VCMD asserted). The FPC branches to the address given in Command Register (T0-T5). If VCMD is not asserted (CC = FALSE), it selects DDACK as an input for CC and continues to next incremental

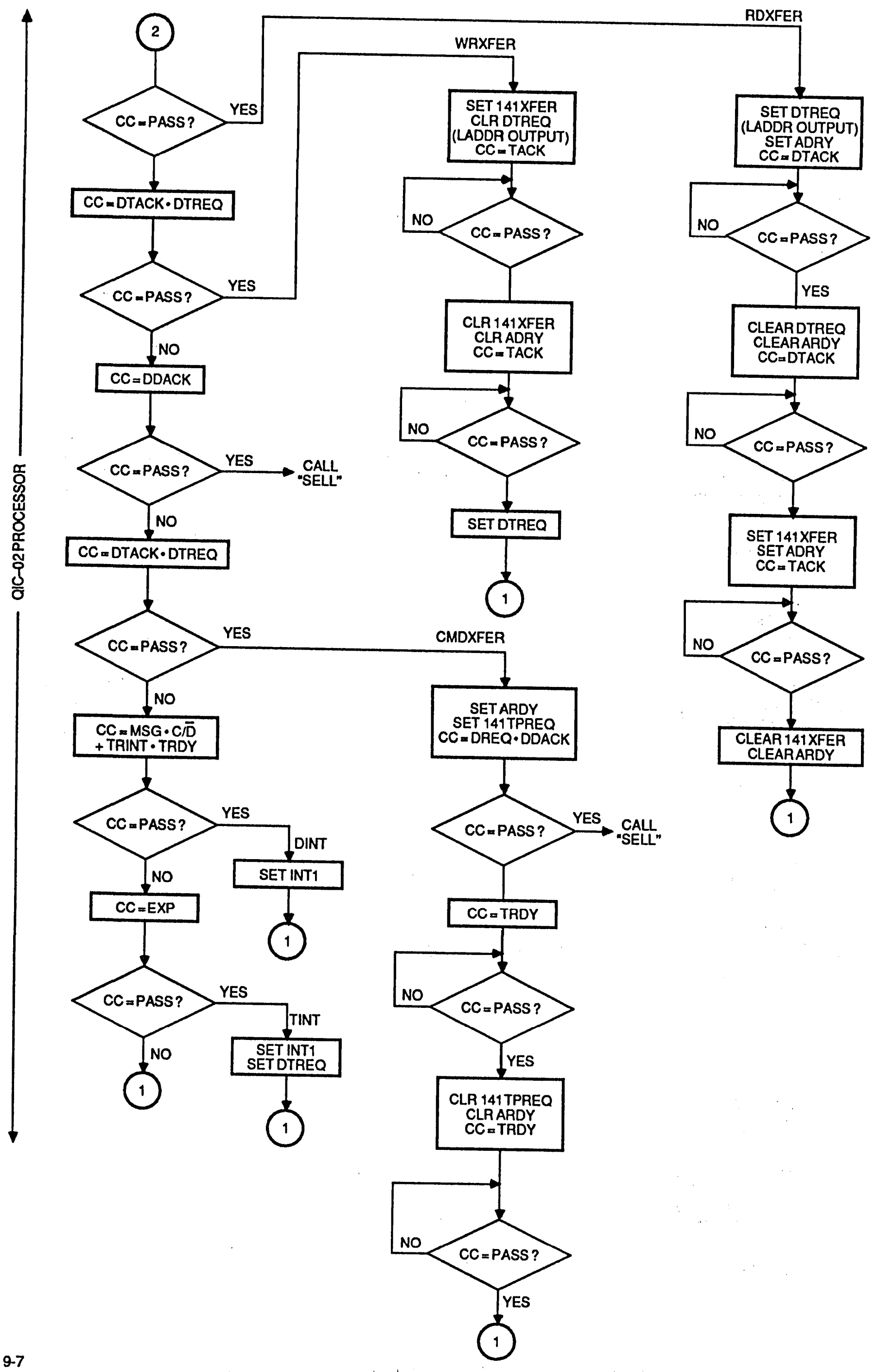
```
DEVICE  addressable_latch (AmPAL22V10);
"This device is the addressable latch used by the Am29PL141 to expand
its I/O capabilities."

PIN
          clk    = 1     enable      = 2           a0       = 3
          a1     = 4     a2          = 5           function = 6
          reset  = 7     spare[0:4]  = 8:11,13     /datn    = 14
          /drst  = 15    intl        = 16          dtreq    = 17
          /tponl = 18    /trint      = 19          /tprst   = 20
          /dack  = 21    spare_out[0:1] = 22:23 ;
DEFINE
          set = function
BEGIN
          IF (reset) THEN ARESET() ;
          case (A2,A1,A0)
          BEGIN
               0)   datn := datn * /enable + set * enable ;
               1)   drst := drst * /enable + set * enable ;
               2)   intl := intl * /enable + set * enable ;
               3)   dtreq := dtreq * /enable + set * enable ;
               4)   tponl := tponl * /enable + set * enable ;
               5)   trint := trint * /enable + set * enable ;
               6)   tprst := tprst * /enable + set * enable ;
               7)   dack := dack * /enable + set * enable ;
          END;
END.

Test_vectors
IN
          clk enable a2 a1 a0 function reset ;
I_O;
OUT
          /datn /drst intl dtreq /tponl /trint /tprst /dack;

BEGIN
                  f
                  u
            e     n         ///
            n     c r     // d ttt/
            a     t e     ddit prpd
          c b     i s     arnr oira
          l l aaa o e     tste nnsc
          k e 210 n t     ntlq lttk
                                   "
          ______________________________
          X X XXX X 1     HHLL HHHH;
          C 0 XXX X 0     HHLL HHHH;
          C 1 000 1 0     LHLL HHHH;
          C 1 000 0 0     HHLL HHHH;
          C 1 001 1 0     HLLL HHHH;
          C 1 001 0 0     HHLL HHHH;
          C 1 010 1 0     HHHL HHHH;
          C 1 010 0 0     HHLL HHHH;
          C 1 011 1 0     HHLH HHHH;
          C 1 011 0 0     HHLL HHHH;
          C 1 100 1 0     HHLL LHHH;
          C 1 100 0 0     HHLL HHHH;
          C 1 101 1 0     HHLL HLHH;
          C 1 101 0 0     HHLL HHHH;
          C 1 110 1 0     HHLL HHLH;
          C 1 110 0 0     HHLL HHHH;
          C 1 111 1 0     HHLL HHHL;
          C 1 111 0 0     HHLL HHHH;
```

Figure 9-6. Addressable Latch PAL Device



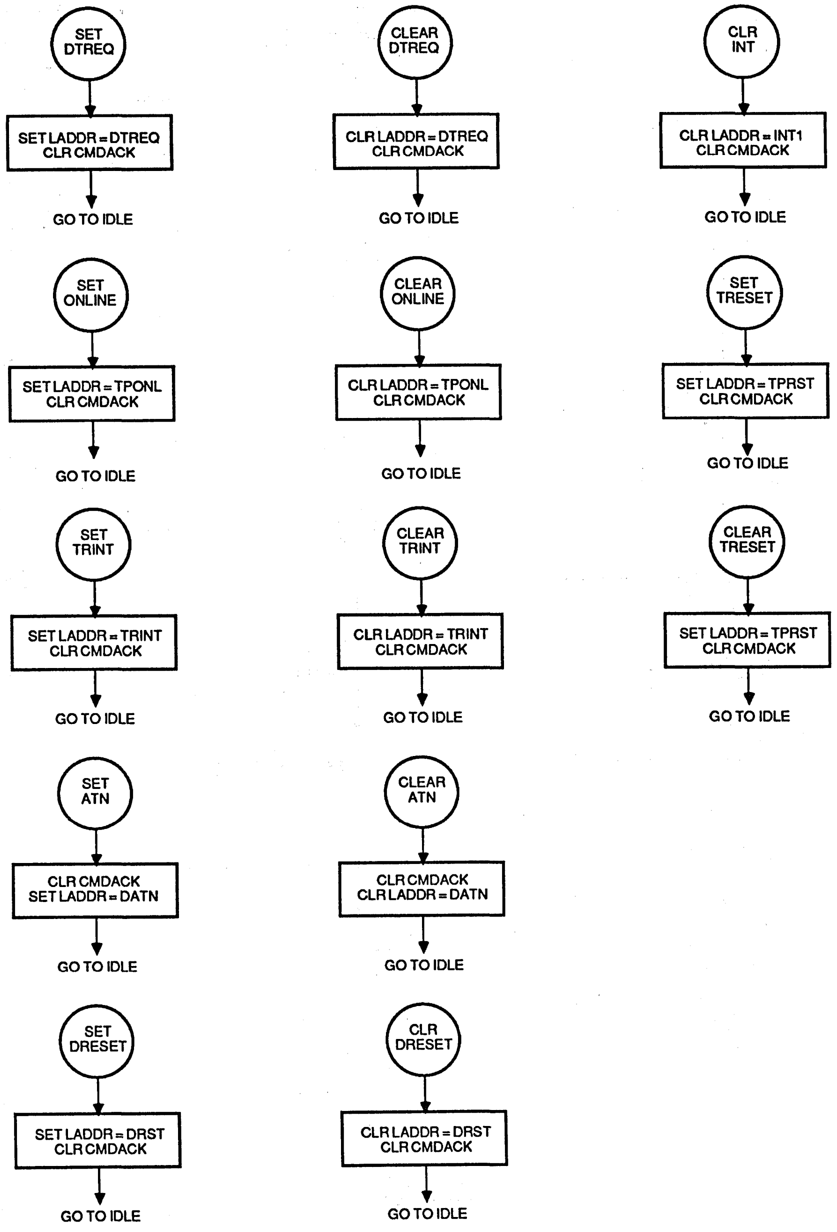
06591A 9-7

Figure 9-7. QIC-02 Controller Program Flow Diagram (Sheet 1 of 2)



06591A 9-7

Figure 9-7. QIC-02 Controller Program Flow Diagram (Sheet 2 of 2)



06591A 9-8

Figure 9-8. Am29PL141 Valid Command Routines

```
device (pl141)
"Am29PL141 QIC-02 and SCSI controller"
default = 1;
define
        def = 1000#h
        vcmd = 1000#h     "condition code mux select lines"
        ddack = 1010#h
        dreq = 1020#h
        tack = 1030#h
        dtareq = 1040#h
        dtanreq = 1050#h
        mctirdy = 1060#h
        exp = 1070#h
        bsyin = 1080#h
        one = 1090#h
        dtack = 10a0#h
        drack = 10b0#h
        trdy = 10c0#h

        datn = 1000#h      "addressable latch lines"
        drst = 1010#h
        intl = 1020#h
        dtreq = 1030#h
        tponl = 1040#h
        trint = 1050#h
        tprst = 1060#h
        dack = 1070#h

        cmdack = 0111#h  "other output lines"
        ddreq = 1800#h
        sel = 1400#h
        bsyout = 1200#h
        lsrccms = 1080#h
        len = 1100#h
        ccmardy = 1001#h
        xfer = 1002#h
        tpreq = 1004#h
        lareset = 1008#h;

test_condition = cc;

begin
idle:   vcmd,   continue;
        vcmd,   goto tm(3f#h);
        ddack, if (cc) then call pl(nsel);
        dreq,   goto pl(dmaxfer);
        tack, goto  pl(rdxfer);
        dtareq, goto pl(wrxfer);
        ddack,  if (cc) then call pl(nsel);
        dtanreq, goto pl(cmdxfer);
        mctirdy, goto pl(dint);
        exp,    goto pl(tint);
        one,    goto pl(idle);
nsel:   ccmardy+bsyin, if (cc) then goto pl(next) else wait;
next:   one, goto pl(idle);

dmaxfer:ddreq+ddreq, if (cc) then goto pl(next1) else wait;
next1:  ccmardy+dack+lsrccms+len, continue;
        dreq, if (cc) then goto pl(next2) else wait;
next2:  dack+len, goto pl(idle);

rdxfer: ccmardy+dtreq+lsrccms+len, continue;
        ccmardy+dtack, if (cc) then goto pl(next3) else wait;
next3:  dtreq + len, continue;
        dtack, if (not cc) then goto pl(next4) else wait;
next4:  xfer+ccmardy+tack, if (not cc) then goto pl(next5) else wait;
```

Figure 9-9. QIC-02 Controller Source Program Listing (Sheet 1 of 2)

```
next5:    one, goto pl(idle);

wrxfer:   xfer+dtreq+len,continue;
          tack+xfer, if (cc) then goto pl(next6) else wait;
next6:    tack, if (not cc) then goto pl(next7) else wait;
next7:    dtreq+len+lsrccms, continue;
          one, goto pl(idle);

cmdxfer:  ccmardy+treq+drack, if (cc) then call pl (nsel);
          trdy,   if (cc) then goto pl(next8) else wait;
next8:    trdy, if (not cc) then goto pl(idle) else wait;

dint:     intl+len+lsrccms, continue;
          one,  goto pl(idle);
tint:     intl+len+lsrccms, continue;
          dtreq+len+lsrccms, continue;
          one, goto pl(idle);
setatn:   datn+len+lsrccms, continue;
          one, goto pl(idle);

clratn:   datn+len, continue;
          one, goto pl(idle);
setdrst:   drst+len+lsrccms, continue;
          one, goto pl(idle);

clrdrst:   drst+len, continue;
          one, goto pl(idle);

clrint:   intl+len, continue;
          one, goto pl(idle);

sdtreq:   dtreq+len+lsrccms, continue;
          one, goto pl(idle);

cdtreq:   dtreq+len, continue;
          one, goto pl(idle),
stponl:   tponl+len+lsrccms, continue
          one, goto pl(idle);
ctponl:   tponl+len, continue;
          one, goto pl(idle);
strint:   trint+len+lsrccms, continue;
          one, goto pl(idle);
ctrint:   trint+len, continue;
          one, goto pl(idle);
stprst:   tprst+len+lsrccms, continue;
          one, goto pl(idle);
ctprst:   tprst+len, continue;
          one, goto pl(idle);
          .ORG 63#d
          lareset, continue;
end.
```

Figure 9-9. QIC-02 Controller Source Program Listing (Sheet 2 of 2)

address (PC+1). The IDLE loop continues in this fashion to select and test CCM input conditions and branch accordingly.

Figure 9-8 shows the Valid Command (VCMD) routines. Each command from the processor will branch to one of these thirteen valid routines. All of these routines are single instructions which set (assert) or clear (deassert) output control lines, which always include resetting the VCMD signal in the Command Register and returning to idle.

Figure 9-9 is the FPC Microprogram source code listing.

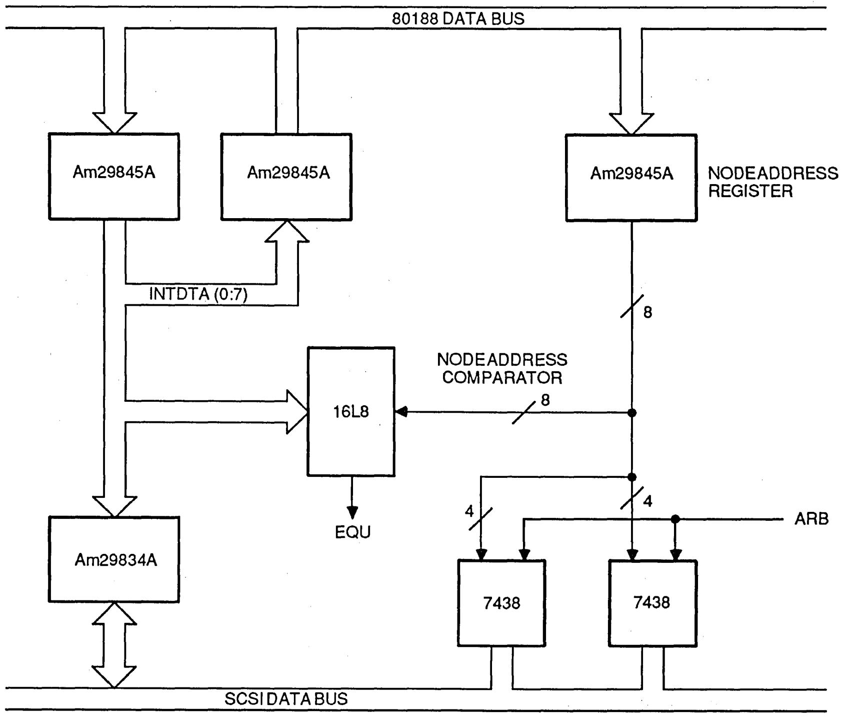
SCSI Interface: The second conditional test in the idle loop is based on DDACK (disk DMA acknowledge). This subroutine is called after the FPC has generated DDREQ (Disk DMA Request) and the processor responded appropriately. The DDACK signal also enables the SCSI bus transceivers for transfer of data. Figure 9-7 shows this call routine (SEL). The FPC asserts ARDY output, to insure processor bus is open long enough for transfer of SCSI data to main memory, and selects BSYIN as CC test input. The FPC waits for SCSI to assert BSYIN before proceeding. BSYIN indicates that the disk is using the SCSI bus. At this time, ARDY can be deasserted, since the data byte is in main memory, and FPC can return to idle at point of exit.

The IDLE Loop then conditionally tests the signal DREQ. If DREQ is asserted, then a jump to the DMAXFER routine takes place. DREQ stands for disk request for data. This signal is generated by SCSI during data transfer, write to or read from disk, as the handshake with acknowledge (ACK) from the FPC. Detecting DREQ being asserted causes the FPC to begin single byte DMA transfer to/from main memory.

First, the FPC asserts DDREQ (disk DMA request) on DMA Request Channel 0 (DRQ0) as an input to Processor (80188). The processor acknowledges this DMA request by asserting DDACK (disk DMA acknowledge) which is an input to the CCM. DDACK is the PCS1 (programmable chip select #1) from the processor. PCS1 is qualified (gated) with DEN, also from the processor, to enable the SCSI transceiver onto the internal 8-bit data bus. Direction of this transceiver is controlled by the I/O signal from the SCSI control bus.

After detecting DDACK asserted, FPC then deasserts DDREQ output, asserts output ARDY (to extend 80188 DMA bus cycle) and sets output to LADDR (addressable latch) which asserts DACK (disk acknowledge). DACK is asserted to SCSI (through LADDR) to continue the data byte transfer handshake (refer to SCSI timing diagram Figures in Appendix B). The CCM is selected for DREQ input. After DREQ is again asserted by SCSI, the transfer is complete. DACK and ARDY are deasserted by the FPC and flow returns to idle loop. This DMA transfer routine is used for both writes to and reads from SCSI since the only difference in timing signals is the I/O directional signal which is controlled by SCSI.

QIC-02 Interface. The next conditional jump instruction tests TACK (tape QIC-02 acknowledge). TACK from QIC-02 is the



06591A 9-10

Figure 9-10. SCSI Advanced Features Upgrade

handshake signal used with XFER from FPC to transfer data (see QIC-02 timing diagrams in Appendix B). With TACK asserted, a jump to RDXFER (read transfer from tape) takes place. All of the QIC-02 processing flow is shown in sheet 2 of Figure 9-7. In a similar fashion to SCSI data transfer, QIC-02 data is a DMA to/from main memory using DMA Request Channel 1 (DREQ1) of the processor. DTREQ is asserted by the FPC (through LADDR) and ARDY is asserted to the processor through CCM. Next is a conditional wait until the processor acknowledges this DMA REQ via DTACK (input to CCM and QIC- 02 Data Bus Transceiver enable). After CC = PASS (i.e. DTACK condition asserted), DTREQ and ARDY outputs are deasserted and the QIC-02 read timing handshake continues with a return to the idle loop.

The next conditional test in the idle loop is for a tape write cycle, indicated by both DTACK and DTREQ being asserted. The WRXFER routine shown in Figure 9-7 matches QIC-02 timing requirements as discussed in Appendix B. The flowcharts for FPC routines include the tape transfer commands and processor interrupts on tape exception conditions.

QIC-02 requires different timing during tape write, read, command, and for tape rewind, which has been divided into separate FPC routines which are interactive with the processor. It begins a tape access by issuing "set on line" (TPONL) valid command and ends tape access with "clear on line" (TPONL). The microprocessing unit section above discusses this interaction.

```
EQU = INTDTA0 * DEVADR0

    + INTDTA1 * DEVADR1

    + INTDTA2 * DEVADR2

    + INTDTA3 * DEVADR3

    + INTDTA4 * DEVADR4

    + INTDTA5 * DEVADR5

    + INTDTA6 * DEVADR6

    + INTDTA7 * DEVADR7
```

Figure 9-11. Node Address Comparator PAL Device Equation

9.3 ADVANCED FEATURES OF SCSI

This design can be upgraded to include SCSI bus arbitration, initiator reselection and operation as target as well as initiator. These features are required in a multiple initiator, multiple target environment.

The logic shown in Figure 9-10, when added to the original design, accomplishes the above. It also provides the means for transferring commands, status, messages, and target selection information via 80188 programmed I/O transfers. For support of target mode operation, it is necessary to provide SCSI bus drivers and addressable latches for the following SCSI signals: REQ, C/D, I/O, MSG, and SEL (not shown).

SCSI bus node addresses are one bit in length. That is, each node is assigned one of eight possible addresses corresponding to one of the eight SCSI bus data lines. During the SELECT phase of bus operation, a node must only test one bit of the data bus to determine if it is being selected. Similarly, during the ARBITRATION phase, the node that is asserting the highest bit on the data bus "wins" control of the bus.

Before allowing SELECTION or ARBITRATION, the 80188 must first load the SCSI "Node Address Register". This register is used as a mask register to determine which bit of the SCSI data bus will be tested during SELECT/RESELECT and which bit will be asserted by this node during the ARBITRATION phase.

9.3.1 Selection (Target reselecting Initiator / selection as Target)

The SCSI bus SEL must now be tested in the Am29PL141's idle loop. If asserted, the Am29PL141 tests the SCSI bus "address compare bit - EQU" (16L8 shown in Figure 9-11) and the SCSI bus BSY signal. If this SCSI node is being addressed and BSY is not asserted; then, the Am29PL141 branches to a routine that will monitor SCSI BSY; else, it returns to its idle loop. To monitor BSY, the Am29PL141 uses one of its internal counters to "time out" a 400 nsec bus free period and then retests SCSI BSY. If the bus is still free, this node is being SELECTED/ RESELECTED and the Am29PL141 will interrupt the 80188 which would then take the necessary action. If the bus is not free, the Am29PL141 returns to its idle loop. The 80188 interrupt handler should test the status of SEL and the "address compare bit" to determine that this is a SELECT/RESELECT interrupt.

9.3.2 Arbitration

To initiate the ARBITRATION cycle, the 80188 issues a command to the Am29PL141 to set an "arbitration request flip-flop ARBRQ". This is another addressable latch bit controlled by the Am29PL141 and subsequently monitored in the Am29PL141's idle loop. If the ARBRQ bit is set, the Am29PL141 will then test SCSI BSY, and if asserted, the Am29PL141 returns to its idle loop. If ARBRQ is asserted and the SCSI bus is not busy, the Am29PL141 will interrupt the 80188, assert the address for this node onto the SCSI bus, assert BSY and begin monitoring SCSI SEL. The address for this node is asserted onto the SCSI bus via the 7438s and a new control bit "ARB". (See Figure 9-10.)

The Am29PL141 will now continuously monitor SCSI SEL and the ARBRQ signal. The asserting of SEL during the arbitration process indicates that another SCSI device has assumed control of the bus and this node should abort the arbitration process. The assertion of SEL causes an "arbitration failed flip-flop" to be set by the Am29PL141. This bit would be added to the status bits readable by the 80188. Also, the deassertion of ARBRQ indicates that the 80188 has terminated the arbitration process. In either case, the Am29PL141 will deassert BSY, remove this node's address from the bus, and return to its idle loop.

The 80188 interrupt processing routine is responsible for reading the SCSI data bus and determining whether this node is the highest currently requesting the bus. If this node has lost the arbitration process, ARBRQ should be deasserted to allow the Am29PL141 to return to its idle loop and then reasserted to begin the process again. If this node appears to have won the arbitration process, the interrupt handler should first check the "arbitration failed flip-flop" before entering the SELECTION phase. This final check is required to insure no other device issued a SEL while the 80188 was responding to the interrupt.

9.4 SUMMARY

This design solves the problem of interfacing older generation tape drives (QIC-02) to modern computer peripherals on the SCSI bus.

The use of the Fuse Programmable Controller and two programmable array logic devices (AmPAL22V10s), allows the implementation of this complex controller with minimum component count, off the shelf standard parts, (see Figure 9-12) and is reconfigurable/upgradable through reprogramming. This design should also give insight into the versatility of the FPC and ease of using this device for new designs.

PARTS LIST

DEVICE	DESCRIPTION	QUANTITY
Am29PL141	Fuse Programmable Controller	1
80188-1	10MHz, 8-bit Microprocessor	1
Am2947	Octal Bidirectional Transceiver	1
Am29843A	9-bit Latch, Non-Inverting	2
Am2958	Octal Buffer, Inverting	2
AmPAL22V10	24-pin Programmable Array Logic	2
Am2950A	8-bit I/O Port with Flags	1
Am29834A	Parity Bus Transceiver, Inverting	1
Am29864	9-bit Transceiver, Inverting	1
Am29828A	10-bit Buffer/Driver, Inverting	1
7438	Open-Collector Drive	2
Am29827A	10-bit Buffer/Drive, Non-Inverting	1
Am27512DC	512K-bit UV EPROM (250 ns)	1
*AmPAL16L8A	20-pin Programmable Array Logic	1

*Use for the five 2-input "OR" gates and for the one 2-input "AND" gate.

Figure 9-12. SCSI and QIC-02 Controller Parts List

CHAPTER 10

HIGH SPEED DMA CONTROLLER USING Am29PL141

10.1 SYSTEM OVERVIEW

In this application, the Am29PL141 Fuse Programmable Controller (FPC) is used to control two hardware blocks that are sequenced at a rate greater than 10 MHz. This application illustrates the power and flexibility of the Am29PL141 in distributed control applications.

The subsystem controlled by the FPC is just a small part of a large computer system. From the viewpoint of the main central processing unit (CPU), this subsystem is an asynchronous peripheral. The peripheral's function is to control a direct memory access (DMA) channel. This channel links the main CPU's memory to a digital signal processor's (DSP) memory. Figure10-1 shows the various hardware blocks which comprise the DMA channel interface. All operations are initiated by the main CPU. Once a command is passed to the subsystem, the main CPU is free to do other tasks. The DMA interface signals the completion of a task by generating an interrupt in the main CPU. A typical command consists of transferring data (totally under the control of the Am29PL141) and/or processing data (controlled by the DSP engine and the Am29PL141).

The overall system can be viewed as a digital signal processor (DSP). It performs high speed data acquisition, digitizing several incoming analog channels. The processor utilizes DSP techniques to modify and/or extract information from this data; the resulting outputs are converted back to analog signals.

By their nature, many DSP algorithms operate on blocks of Data. In this particular application, the incoming channels consist of various speech signals. After digitalization, the speech bandwidth is compressed using linear predictive coding (LPC) techniques. A 64 kbit/sec channel is compressed to a 2.4 kbit/sec data stream using LPC. Six compressed input channels are multiplexed over one serial link. Simultaneously, the processor receives a multiplexed LPC data stream. It demultiplexes this data and expands the compressed data resulting in analog speech output channels.

Real time constraints mandate a high speed DMA controller to orchestrate the filling and emptying of the LPC data RAM. Incoming channels of raw speech data are stored in this RAM. Once available, the processor invokes an analysis routine that extracts the LPC parameters. This parametric information is multiplexed and transmitted over one serial link. In the other direction, received LPC parameters are demultiplexed. A synthesis routine is then invoked which reconstructs the speech signals. These reconstructed speech waveforms are stored in the data RAM. The Am29PL141 not only controls the DMA channel, but also performs a sequencing function assisting the subsystem's DSP engine.

The following sections describe the CPU-FPC interface, the FPC output lines, the use of 27S18 and Am2940 for address generation, and finally the microprogram for this application. A more complete discussion of the Am29PL141 FPC is given in Chapter 1 and Appendix E.

10.2 CPU-FPC INTERFACE

Whenever the CPU desires service from the DSP subsystem, it issues a command by placing it in a 5-bit instruction register. This register's outputs are available to the FPC as T[4:0]. The CPU sets the valid instruction flip flop to indicate the presence of a new command. The flip flop output is connected to the FPC's CC test input. While idle, the FPC interrogates this flip flop. When a new command is detected, the FPC commences execution of the instruction. Upon completion, the valid instruction flip flop is cleared (using P[11]), and a status bit is output to the CPU. Data passes between the main CPU data bus and the DSP data bus via a specialized 16 bit bi-directional I/O port. In addition to buffering data during transfers, the I/O port is used to initialize the DSP data RAM.

There are actually 14 different instructions represented in bits T[3:0]. T[4] is used to tell the DSP engine to perform calculations with the DMA interface generating the addresses.

Three groups of CPU commands are defined:

1. Data Transfer In (to the DSP memory) – 6
2. Data Transfer Out (from the DSP memory) – 7
3. Data Memory Initialize – 1

The number following each group name denotes the number of instructions within that group.

Any instruction in the Data Transfer In group can additionally have T[4] as a qualifier. When T[4] is negated, the DMA interface only transfers data in to the DSP memory. When T[4] is asserted, the DMA interface serves as the address generator for the DSP engine for a particular task after the data transfer is complete. By reexamining the CPU command, the FPC determines how many addresses it needs to generate for the task.

Instruction decoding is a simple task in the Am29PL141 using its multiway branch instruction. In this application T[3:0] are masked and a branch to one of sixteen locations is taken as determined by the pattern present on T[3:0]. Subsequent paths taken are derived from this multiway branch.

10.3 Am29PL141 CONTROLLER

The Am29PL141 is the heart of the DMA interface. Once the CPU passes a command, the FPC takes over. All data transfer operations are under its control. When a new instruction is detected, the 29PL141 decodes it by reading it in on its T0–T4 test inputs. The DONE output of the Am2940 is connected to the FPC T5 test input for signaling the completion of an address sequence. When an input instruction is decoded, control branches to the appropriate control sequence.

A 64 x 32 bit PROM resides on the Am29PL141. The upper 16 bits of each word are used to control the on-board sequencer. The functions of these bits are defined by AMD and are not alterable by the user. The lower 16 bits of each word are brought out through a pipeline register as output lines and are user-defined (P15–P0). Appendix E, the Data Sheets, defines the microinstruction word in detail.

The control data that appears at the outputs (P[00:15]) of the FPC depends on the type of instruction. Five bits (P[00:04]) are used as an address to a 32 x 8 lookup PROM. Four bits (P[06:09]) provide instructions and control to an Am2940 high speed DMA address generator. Two bits (P[10], P[12]) control the specialized bidirectional I/O port between the two processor

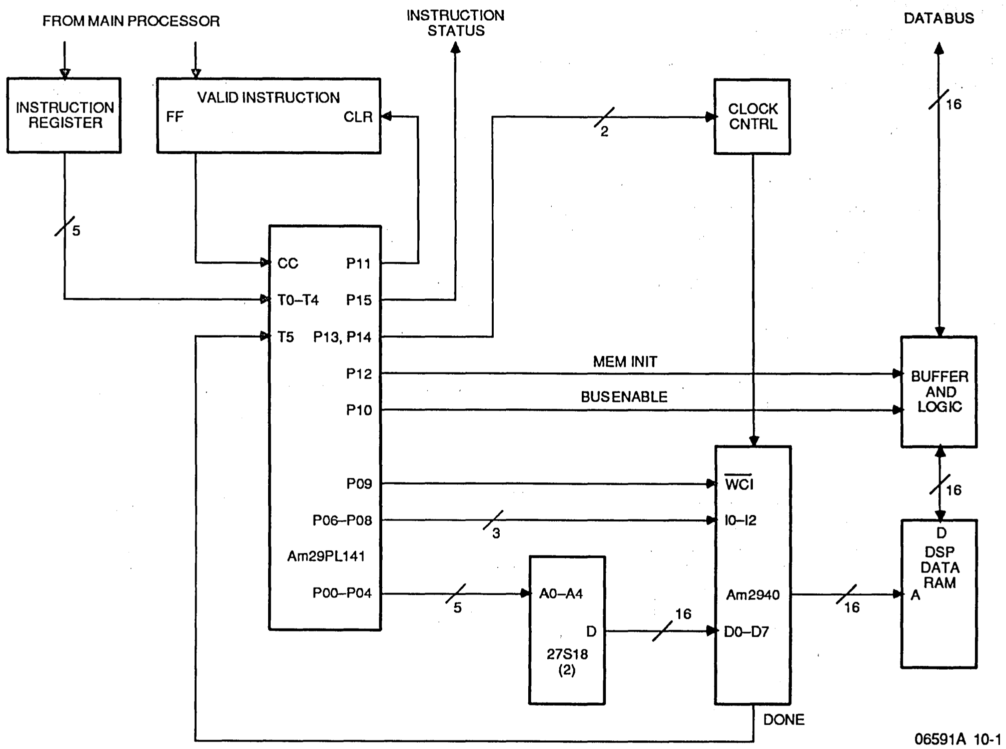


Figure 10-1. DMA Channel Interface

data buses. Finally, two bits (P[13], P[14]) are used to control the clock source to the Am2940 address generator. P[15] signals the main CPU when the execution of a command is complete.

Figure 10-2 illustrates the assignment of the 16 Am29PL141 output lines. These output lines are controlled by the FPC microprogram instructions. One-half of each microinstruction word is used to specify these outputs. All but one of these lines are used in this application. These 16 output lines are grouped into eight fields of varying widths. The specifics of each field, the field width, and the type of micro-operations performed, are as follows:

PROM Address Control

The 5 bit field formed by P[4:0] is named A[4:0]. After a CPU command is decoded, the FPC determines which block of data RAM is to be accessed and its length. The starting address of each block and its length are stored in the look-up PROMs. A[4:0] provide the addresses to the lookup PROMs for each new DMA operation.

DMA Address Generator Control

P[8:6] form a 3 bit field named I[2:0]. These bits are the instructions for the Am2940 address generator. Operations performed by the field include reading and writing various data and control registers on the Am2940.

DMA Count Control

P[9] is a one bit field named CNT wired to the ACI and WCI inputs of the Am2940. The signal enables the counting operation of the address generator. This effectively provides clock control in addition to the external clock circuitry.

Data Bus Interface Control

Bits P[10] and P[12] form two one-bit fields for this function. P[10] is named BEN and controls data transfers between the two CPU data buses. When it is asserted, transfers are allowed. P[12] is named ZEN (Zero Enable). When asserted, it overrides BEN for transfers into the DSP data memory and instead places zeroes on the data bus. This feature is useful for initialization in certain tasks. By having the DMA controller provide this function, the DSP is offloaded and subsequently has more time for performing calculations.

Instruction Status

P[11] and P[15] form two one-bit fields used in conjunction with the CPU instruction interface. P[11] is named CLR. This bit serves as the clear signal to the valid instruction flip flop. This flip flop can only be set by the main CPU and reset by the DMA controller. When an instruction is completed by the DMA controller, it resets this flip flop. The FPC idles until the main CPU sets this flip flop indicating the presence of a new instruction in the instruction register.

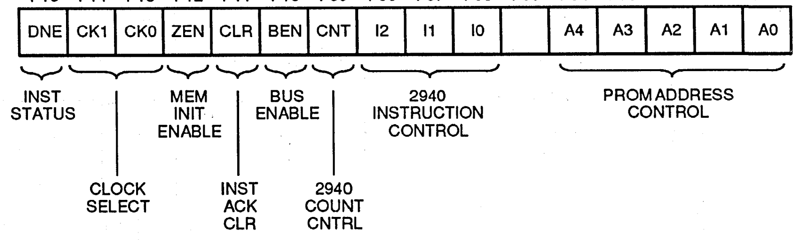
P[15] is named DNE and is sent back to the main CPU. When asserted by the FPC it indicates that the DMA subsystem has completed the execution of a command and is awaiting a new one.

Clock Control

P[14:13] form a two bit field named CK[1:0]. These bits control the source of the clock to the Am2940s. Three selections are possible: 1) system clock; 2) system clock shifted by 180°; and 3) clock inhibit.

10.4 ADDRESS GENERATION

Several channels of data are stored in the DSP data RAM. For each channel, the DMA controller must input to and/or output from the proper section of the memory. Generation of the appropriate addresses is handled by two Am27S18SA PROMs and two Am2940 address generators.



06591A 10-2

Figure 10-2. Format of User Output Portion of Am29PL141 Microcode

The FPC determines a starting address and a block length from a decoded instruction. The actual values of this data are stored in the Am27S18SA lookup PROMs. Two five-bit addresses, representing the starting address and block length are presented to the PROMs. The data outputs of the PROM are routed to the Am2940s on their data inputs (D0–D7) and loaded into the appropriate registers. Once initialized with these "seed" values, the Am2940s provide sequential addresses to the DSP data RAM until the word count expires. The DONE signal from the Am2940s alerts the FPC to this condition.

In addition to providing DMA addresses, this section of the hardware generates addresses for the DSP for certain processing steps that are time critical. Some sections of the LPC algorithm sequentially step through the memory block repeatedly. For these tasks, the FPC keeps track of how many passes are required and issues control data to the address generators. Basically it performs dummy DMA cycles where addresses are generated but no data passes through the data bus interface.

10.5 FPC MICROCODE

Figure 10-3 is the flowchart of the code implemented for this application. A total of 45 words are used. This leaves ample room for future modifications to the interface. Of the 45 locations used, 30 are used for instruction decoding. However, while the FPC is decoding an instruction, its control outputs are simultaneously loading values

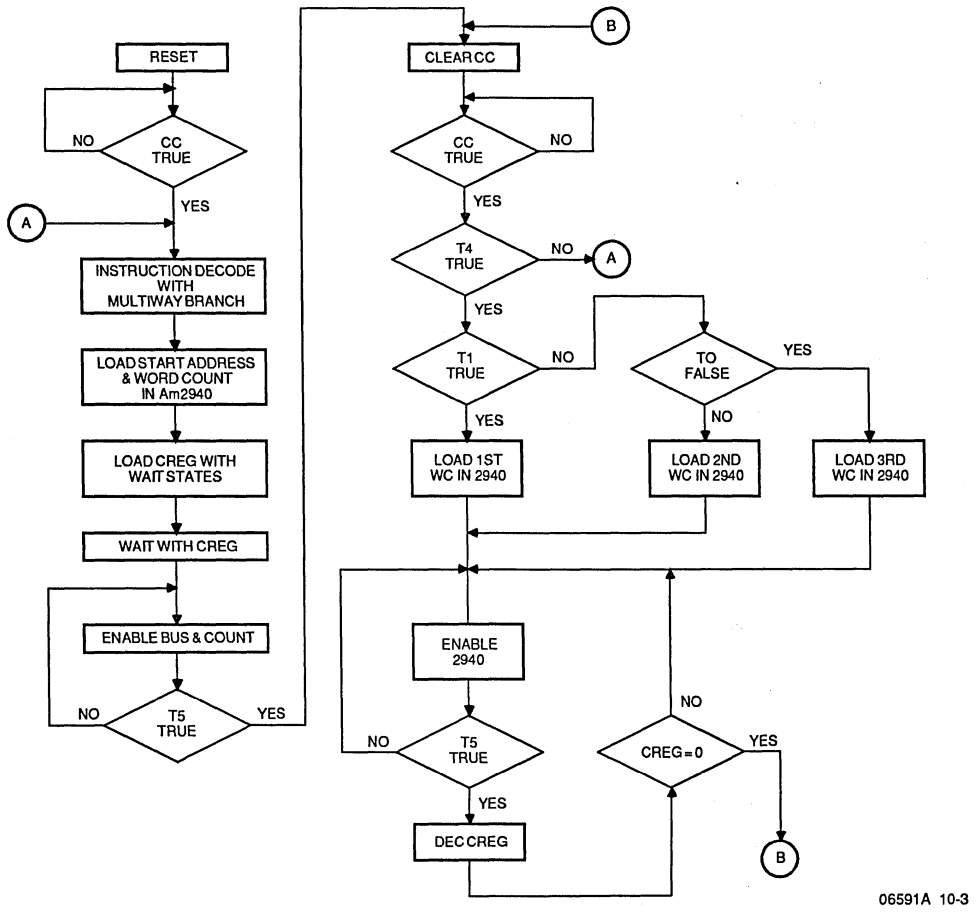


Figure 10-3. DMA Controller Program Flow Diagram

into the Am2940s. This parallel operation allows the data transfers to take place with a minimum of overhead. By the time the instruction is decoded, the Am2940 data and control registers are loaded and ready to start the transfer operation.

After some wait states are executed the data transfer commences. When finished, T[4] is tested. If asserted the FPC goes back and looks at T[3:0] to determine how many passes it must make through the data for the DSP engine. It then commands the Am2940s to start the dummy DMA cycles and runs until its pass count expires. A pass count is easily implemented using the C Register on board the Am29PL141. Between each pass the Am2940s are reinitialized to point at the start of a data block. When all passes are complete, the CPU is notified, and the FPC waits for the next instruction.

Figure 10-4 is a listing of the microcode described above. It is written using an assembler written specifically for the Am29PL141 by AMD. This software runs on an IBM PC/XT and is available gratis to any designer using the Am29PL141. Most of the code in this application was debugged using a companion simulator also available from AMD. Only real time timing aspects could not be evaluated. Having this software available makes the design engineer's job easier by minimizing the amount of time spent translating concept to PROM data for the FPC.

```
"                    A HIGH SPEED DMA CONTROLLER                      "

device (pl141)
default = 1 ;
define

" The following mnemonics are the names assigned to the micro
  operations in the eight different fields defined for P(15:0)

  FIELD NAME = DNE
                                                                      "
        DONE    = 0000#H
        NDONE   = 8000#H

" FIELD NAME = CS(2:0)
                                                                      "
        CLK1    = 0000#H
        CLK2    = 2000#H
        NOCLK   = 6000#H

" FIELD NAME = ZEN
                                                                      "
        IMEM    = 0000#H
        NOIMEM  = 1000#H

" FIELD NAME = ICR
                                                                      "
        CLRINST = 0000#H
        NOCLR   = 0800#H

" FIELD NAME = BEN
                                                                      "
        BUSON   = 0000#H
        BUSOFF  = 0400#H

" FIELD NAME = CNT
                                                                      "
        CNTON   = 0000#H
        CNTOFF  = 0200#H

" FIELD NAME = I(2:0)
                                                                      "
        WRCR    = 0000#H
        REIN    = 0100#H
        LDAD    = 0140#H
        LDWC    = 0180#H
        ENCT    = 01C0#H
```

Figure 10-4. DMA Controller Source Program Listing (Sheet 1 of 4)

```
" FIELD NAME = A(4:0)
                                                                        "
        ADD0     = 0000#H
        ADD1     = 0001#H
        ADD2     = 0002#H
        ADD3     = 0003#H
        ADD4     = 0004#H
        ADD5     = 0005#H
        ADD6     = 0006#H
        WC0      = 0008#H
        WC1      = 0009#H
        WC2      = 000A#H
        WC3      = 000B#H
        WC4      = 000C#H
        WC5      = 000D#H
        WC6      = 000E#H
        WC7      = 000F#H;

begin

" The first 16 locations form the branch table for decoding the
  instruction bits present on T(3:0)                                      "

ZERO:   NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD0,
          IF (CC) THEN GOTO PL(DTI1);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD1,
          IF (CC) THEN GOTO PL(DTI2);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD2,
          IF (CC) THEN GOTO PL(DTI3);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD3,
          IF (CC) THEN GOTO PL(DTI2);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD4,
          IF (CC) THEN GOTO PL(DTI3);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD5,
          IF (CC) THEN GOTO PL(DTI4);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD0,
          IF (CC) THEN GOTO PL(DTO1);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD1,
          IF (CC) THEN GOTO PL(DTO2);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD2,
          IF (CC) THEN GOTO PL(DTO3);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD3,
          IF (CC) THEN GOTO PL(DTO4);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD4,
          IF (CC) THEN GOTO PL(DTO1);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD5,
          IF (CC) THEN GOTO PL(DTO2);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+ADD1,
          IF (CC) THEN GOTO PL(DTO3);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+REIN+001F#H,
          IF (CC) THEN GOTO PL(RESET);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+REIN+001F#H,
          IF (CC) THEN GOTO PL(RESET);
        NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDAD+001F#H,
          IF (CC) THEN GOTO PL(MEMINIT);
```

Figure 10-4. DMA Controller Source Program Listing (Sheet 2 of 4)

```
" The next 4 instructions have identical internal control but different
  outputs on P(15:0).  They are used for instructions in the DATA  TRANS-
  FER IN (DTI) group.  They are also part of the instruction decoding."

DTI1:     NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDWC+WC0,
            IF (CC) THEN GOTO PL(DTIWAIT);
DTI2:     NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDWC+WC1,
            IF (CC) THEN GOTO PL(DTIWAIT);
DTI3:     NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDWC+WC2,
            IF (CC) THEN GOTO PL(DTIWAIT);
DTI4:     NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDWC+WC3,
            IF (CC) THEN GOTO PL(DTIWAIT);

" The next 4 instructions have identical internal control but different
  outputs on P(15:0).  They are used for instructions in the DATA TRANS-
  FER IN (DTI) group.  They are also part of the instruction decoding."

DTO1:     NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDWC+WC4,
            IF (CC) THEN GOTO PL(DTOWAIT);
DTO2:     NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDWC+WC5,
            IF (CC) THEN GOTO PL(DTOWAIT);
DTO3:     NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDWC+WC6,
            IF (CC) THEN GOTO PL(DTOWAIT);
DTO4:     NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDWC+WC7,
            IF (CC) THEN GOTO PL(DTOWAIT);

" This instruction is executed for the DATA MEMORY INITIALIZE (DMI) group"

MEMINIT: NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+LDWC+WC7,
            IF (CC) THEN GOTO PL(ZWAIT);

" Program FPC for DTI wait states using the CREG                   "

DTIWAIT: NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+ENCT+001F#H,
            IF (CC) THEN LOAD PL(4);
          NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+ENCT+001F#H,
            IF (CC) THEN GOTO PL(WAIT1);

" Program FPC for DTO wait states using the CREG                   "

DTOWAIT: NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+ENCT+001F#H,
            IF (CC) THEN LOAD PL(6);
WAIT1:    NDONE+CLK1+NOIMEM+NOCLR+BUSOFF+CNTOFF+ENCT+001F#H,
            WHILE (CREG <> 0) LOOP TO PL (WAIT1);
          NDONE+CLK1+NOIMEM+NOCLR+BUSON+CNTON+ENCT+001F#H,
            IF (T5) THEN GOTO PL(CLEARCC) ELSE WAIT;

" Program FPC for MEMORY INITIALIZE function                        "

ZWAIT:    NDONE+CLK1+IMEM+NOCLR+BUSOFF+CNTON+ENCT+001F#H,
            IF (T5) THEN GOTO PL(CLEARCC) ELSE WAIT;

" Clear VALID INSTRUCTION flip flop (CC input to FPC)               "
```

Figure 10-4. DMA Controller Source Program Listing (Sheet 3 of 4)

APPENDIX A

JEDEC STANDARD No.3

The fuse map generated by the Am29PL141 Assembler adheres to the JEDEC standard No. 3 (October 1983) which is a standard data transfer format between a data preparation system and a programmable logic device programmer.

The information to be sent to the device programmer is divided into the following categories:

1. The design specification identifier
2. The device to be programmed
3. Fuse links that must be blown to implement the design
4. Information to perform a structured functional test
5. Other information (e.g.,sumcheck)

A transmission must consist of the following legal characters. Any other characters present in the transmission file may result in invalid operation.

STX	02 hex	start of text
ETX	03 hex	end of text
LF	0A hex	line feed
CR	0D hex	carriage return
all printable characters	20 hex to 7E hex inclusive	

The Assembler forms the transmission file by putting the STX character at the beginning of the file, followed by the fuse link information, the fuse checksum, the ETX character, and the transmission sumcheck. An example Assembler transmission (fuse map) file is:

```
<STX>F1*
L0000 0 10010 1 111 111111 1111111111111000 *
C02EF*
<ETX>0A94
```

Fuse Link Information

Each device fuse link is assigned a decimal number. Each numbered fuse can have two possible states: a Zero specifying a low-resistance (unblown) link and a One specifying a high-resistance (blown) link.

Fuse information is presented in three fields: F, L and C.

F: This field specifies the state of the unspecified fuses in the logic device. This field corresponds to the DEFAULT section in the program source file. The default for this field is 'F0', all unspecified fuses unblown.

L: Each numbered link is addressed by the 'L' field. The L is immediately followed by a variable length decimal number indicating address of the first link in the following string of data. The string of data can be any convenient length terminated by an '*'. In the Assembler each string is 32 characters long.

C: This is the checksum field for the link information. It is computed by performing a 16-bit unsigned addition of 8-bit words formed from all the fuse link states specified in the file.

The 8-bit words are formed as in the following diagram:

Example: Checksum Computation

```
<STX>F1*
L0000 0 10010 1 111 111111 1111111111111000 *
C02EF*
<ETX>0A94

link   7        0
       1101 0010  →  D 2  hex
       1111 1111     F F  hex
       1111 1111     F F  hex
       0001 1111     1 F  hex
       ________________________
                   0 2 E F  hex = checksum
```

Note:

If the number of fuse links is not a multiple of 8, then the last word will be formed by setting Zeroes for all the bit locations more significant than the last link. The 16-bit checksum is specified as a C followed by 4 hex characters and an '*'.

OTHER INFORMATION

The transmission format is ended by an ETX character followed by the sumcheck. The sumcheck is the 16-bit unsigned addition of the ASCII values of all the characters in the transmission file between and including STX and ETX. The parity bit is excluded in this calculation.

APPENDIX B

QIC-02 AND SCSI INTERFACE SIGNALS AND TIMING DIAGRAMS

QIC-02 INTERFACE

QIC-02 is an industry standard which defines the interface between a host system and Quarter Inch Cartridge Tape Drives. Read/write commands, status and, of course, data are transmitted over this interface, as depicted in Figure B-1. The bus and control signals between QIC-02 and host are all standard TTL levels. Timing diagrams for this interface are shown in Figures B-2 through B-4. This interface handshake timing is duplicated for the host side by the FPC and two AmPAL22V10s.

ACKNOWLEDGE (ACK) is used with Transfer to transfer data across the interface.

READY (RDY) indicates that the tape drive can accept a command. It is used to handshake the command across the interface. In the write mode, READY indicates that the drive's internal buffer is empty and ready to receive new data. In the read mode, READY indicates the drive buffer can now be accessed by the host.

EXCEPTION (EXP) alerts the host that the execution of a command has been terminated. This may be a normal completion or an interrupt due to a fault (hard errors, write protected, etc.). The response by host must be READ STATUS.

DIRECTION (DIRC) indicates direction of data flow. Signal is used to enable/disable the data bus transceivers in the HOST.

ON-LINE signal is deasserted at the beginning of a read (from tape) or write (to tape) operation.

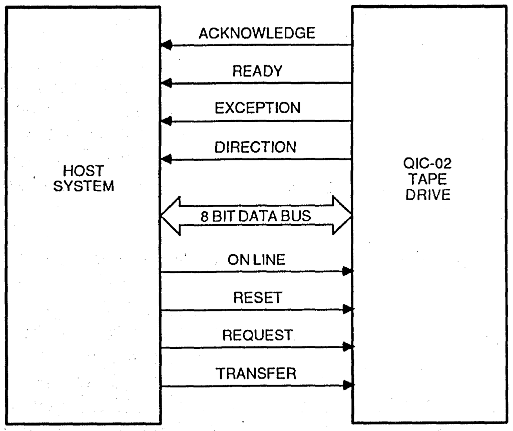
RESET initializes the tape drive. The drive recalibrates the heads to track zero.

REQUEST indicates that a command is on the data bus.

TRANSFER is used with ACKNOWLEDGE to handshake data over the bus, see timing diagram.

SCSI INTERFACE

Small Computer Systems Interface (SCSI) evolved from the disk controller standard developed by Shugart Associates (SASI) in the late 1970s. The SCSI standard was developed by ANSI X3T9.2 subcommittee starting in 1982. SCSI defines an 8-bit parallel bi-directional data bus with parity, plus nine control lines. SCSI protocol allows single or multiple host computers (initiators) to share multiple (expensive) peripherals (targets, i.e. hard disk, floppy disks, etc.), as depicted in Figure B-5. Up to eight Daisy Chained devices can reside on the SCSI bus, with data transfer rates of 4 Mbytes/sec. Synchronous and 1.5 Mbyte/sec. asynchronous. The timing diagrams for the interface are shown in Figures B-6 through B-8.



06591A C-1

Figure B-1. QIC-02 Interface

The interface signals are:

I/O is driven by a target to control the direction of data movement. True indicates input to the initiator.

MSG is drive by a target to indicate "Message Phase". When MSG is asserted, REQ (Request) is also asserted by the target for transfer of data byte indicating the end of the operational phase ("Message").

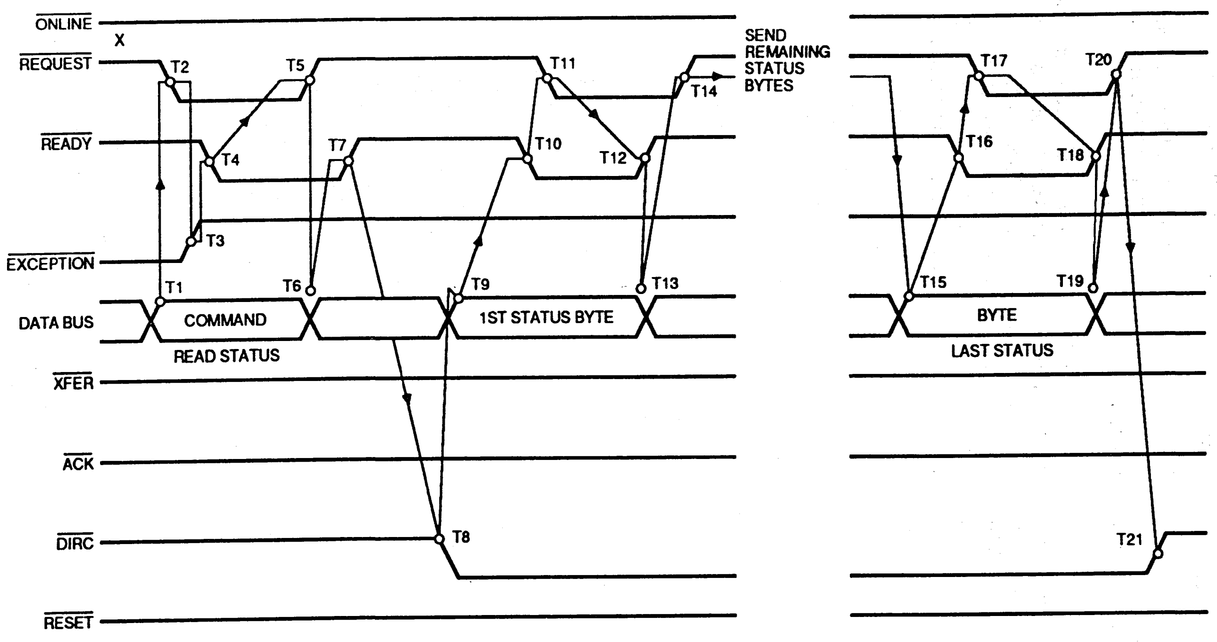
REQ asserted by target indicates that a data byte is to be transferred on the data bus. Data byte is transferred via handshake with ACK (Acknowledge).

ATN (Attention) is driven by an initiator to indicate to target an "attention" condition.

An initiator uses SEL along with appropriate data (address) bits (0-7) being asserted to select a target. Select line is deasserted after the target asserts BSY to acknowledge selection.

RST (Reset) is a pulse asserted by the initiator to stop target's present operation and return same to idle condition.

Data bus and control signals require open collector drivers capable of sinking 48 mA each to support SCSI mode of multiple initiators with multiple targets. SCSI provides for either single ended (6 meter max. cable length) transmission or differential (if a distance up to 25 meters is required).



ACTIVITY	CRITICAL TIMING
T1 – HOST COMMAND TO BUS	N/A
T2 – HOST SETS REQUEST	T1–T2>0 µs
T3 – CONTROLLER RESETS EXCEPTION	T3–T4>10 µs
T4 – CONTROLLER SETS READY	20<T2–T4<500 µs *
T5 – HOST RESETS REQUEST	T4–T5>0 µs
T6 – BUS DATA INVALID	T4–T6>0 µs
T7 – CONTROLLER RESETS READY	20<T5–T7<100 µs
T8 – CONTROLLER CHANGES BUS DIRECTION	N/A
T9 – 1ST STATUS BYTE TO BUS	N/A
T10– CONTROLLER SETS READY	T7–T10>20 µ s

*NOTE: THIS MAY BE > 500 µs UNDER SOME CONDITIONS

T11 – HOST SETS REQUEST	N/A
T12 – CONTROLLER RESETS READY	T11–T12<1 µs
T13 – BUS DATA INVALID	T11–T13>0 µs
T14 – HOST RESETS REQUEST	T11–T14>20 µs
T15 – LAST STATUS BYTE TO BUS	N/A
T16 – SAME AS T10	SAME AS T10
T17 – SAME AS T11	SAME AS T11
T18 – SAME AS T12	SAME AS T12
T19 – SAME AS T13	SAME AS T13
T20 – SAME AS T14	SAME AS T14
T21 – CONTROLLER CHANGES BUS DIRECTION	N/A
T22 – CONTROLLER SETS READY	T20–T21>0 µs
X – DON'T CARE	T21–T22>0 µsN/A
	T11–T12<1 µs

06591A C-2

Figure B-2. QIC-02 Read Status Command Timing Diagram

ERR 59 put a number or a defined name here
Warning: Syntax error
User Action: Put a valid number or predefined name here

ERR 60 put a constant or a number here
Warning: Syntax error
User Action: Put a valid number or predefined constant here

ERR 61 put a '.' after END to terminate the assembler file
Warning: Unexpected end of file
User Action: Include a '.' after the keyword END

ERR 62 put a ':' for labels or ',' for output
Warning: The punctuation symbols ':' or ',' are necessary to separate sections in each statement
User Action: Use ':' or ','

ERR 63 put a ',' to separate the output section
Warning: The ',' symbol is required here
User Action: Put a ','

ERR 64 put a ';' here
Warning: The ';' symbol is necessary to separate program sections or statements
User Action: Put a ';' as a statement separator

ERR 65 put a name here
Warning: A valid predefined constant is necessary here
User Action: put a predefined name here

ERR 66 put a 'TO' here : loop TO PL
Warning: LOOP must be followed by the keyword 'TO'
User Action: put the keyword "TO"

ERR 67 put an operand between logical operators
Warning: Logical expression is incorrect
User Action: Put an operand between '*' and '+'

ERR 68 put an operand between nested operands
Warning: Logical expression is incorrect
User Action: Put an operand after the '('

ERR 69 put an operand here
Warning: Syntax error
User Action: Match an operand with this logical operator

ERR 70 put an operand or ')' to complete the expression
Warning: Unmatched parenthesis or missing operand
User Action: Check logical expression

ERR 71 put an operator between operands
Warning: Logical operators '*' and '+' cannot follow each other
User Action: Check the logical expression/equation

ERR 72 put PL , TM , or SREG here
Warning: Incorrect statement syntax
User Action: Put GOTO PL, GOTO TM or GOTO (SREG)

ERR 73 redefinition of label
Warning: Label has been redefined
User Action: Check label names

ERR 74 separate the output section with a ','
Warning: Syntax error
User Action: Put the necessary ',' here

ERR 75 Severe warning : redefinition of PROM location * See source line *****
Warning: PROM location specified more than once
User Action: Check the flow of your microprogram; some statements may have overlapped due to the use of numbers as labels

ERR 76 SOFTWARE error ... see WRITE WORD module
Warning: The program cannot form the PROM word properly
User Action: None

ERR 77 specify the pipeline data field
Warning: Syntax error
User Action: Specify a data field in PL(data)

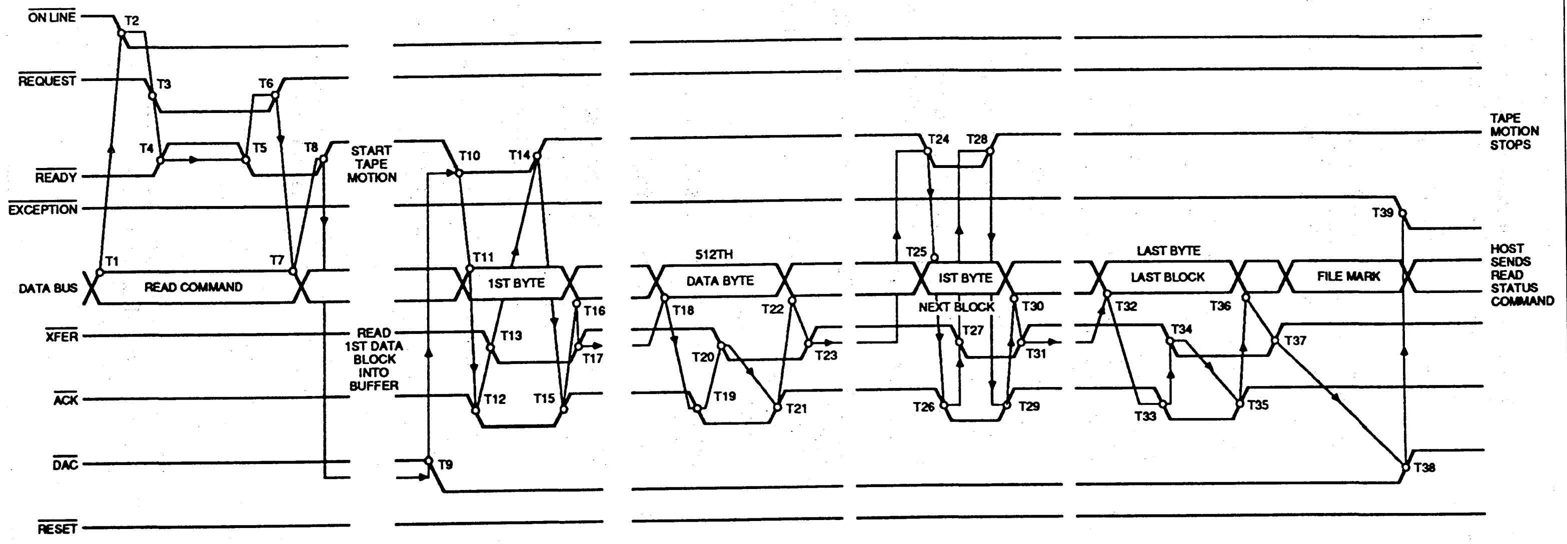
ERR 78 Statement * not supported in *****
Warning: This statement combination does not correspond to any device mnemonic
User Action: Check the list of available statements

ERR 79 this condition has not been defined
Warning: Undefined test condition
User Action: Pair this identifier with a test condition in the DEFINE section

ERR 80 this is a keyword
Warning: Cannot use this keyword in this context
User Action: Use a different variable name

ERR 81 this is not a binary number
Warning: Not a binary number
User Action: use '#b'

ERR 82 this is not a decimal number
Warning: Not a decimal number
User Action: Use 'd'



ACTIVITY	CRITICAL TIMING
T1 – HOST COMMAND TO BUS	N/A
T2 – HOST SETS ONLINE	N/A
T3 – HOST SETS REQUEST	T2–T3>0 µs
T4 – CONTROLLER RESETS READY	T3–T4<1 µs
T5 – CONTROLLER SETS READY	20<T4–T5<500 µs
T6 – HOST RESETS REQUEST	T5–T6>0 µs
T7 – BUS DATA INVALID	T5–T7>0 µs
T8 – CONTROLLER RESETS READY	20<T6–T8<100 µs
T9 – CONTROLLER CHANGES DIRC	N/A
T10– 1ST DATA BYTE TO BUS	N/A
T11– CONTROLLER SETS READY	N/A
T12– CONTROLLER SETS ACK	T11–T12>70 ns
T13– HOST SETS XFER	T12–T13>0 µs

ACTIVITY	CRITICAL TIMING
T14 – CONTROLLER RESETS READY	T13–T14<1 µs
T15 – CONTROLLER RESETS ACK	0.5<T13–T15<3 < µs
T16 – BUS DATA INVALID	T13–T16>0 µs
T17 – HOST RESETS XFER	T15–T17>0 µs
T18 – BUS DATA VALID N/A	
T19 – CONCTROLLER SETS ACK	SAME AS T12
T20 – HOST SETS XFER	SAME AS T13
T21 – CONTROLLER RESETS ACK	SAME AS T15
T22 – BUS DATA INVALID	SAME AS T16
T23 – HOST RESETS XFER	SAME AS T17
T24 – CONTROLLER SETS READY	N/A
T25 – 1ST BYTE TO BUS	N/A
T26 – CONTROLLER SETS ACK	SAME AS T12

ACTIVITY	CRITICAL TIMING
T27 – HOST SETS XFER	SAME AS T18
T28 – CONTROLLER RESETS READY	SAME AS T14
T29 – CONTROLLER RESETS ACK	SAME AS T15
T30 – BUS DATA INVALID	SAME AS T16
T31 – HOST RESETS XFER	SAME AS T17
T32 – LAST BYTE TO BUS	N/A
T33 – CONTROLLER SETS ACK	SAME AS T12
T34 – HOST STES XFER	SAME AS T13
T35 – CONTROLLER RESETS ACK	SAME AS T15
T36 – BUS DATA INVALID	SAME AS T16
T37 – HOST RESETS XFER	SAME AS T17
T38 – CONTROLLER SETS EXCEPTION	N/A
T39 – CHANGE BUS DIRECTION	N/A

*NOTE: THIS TIME MAY BE > 500 µs UNDER SOME CONDITIONS

06591A C-4

Figure B-4. QIC-02 Read Data Command Timing Diagram

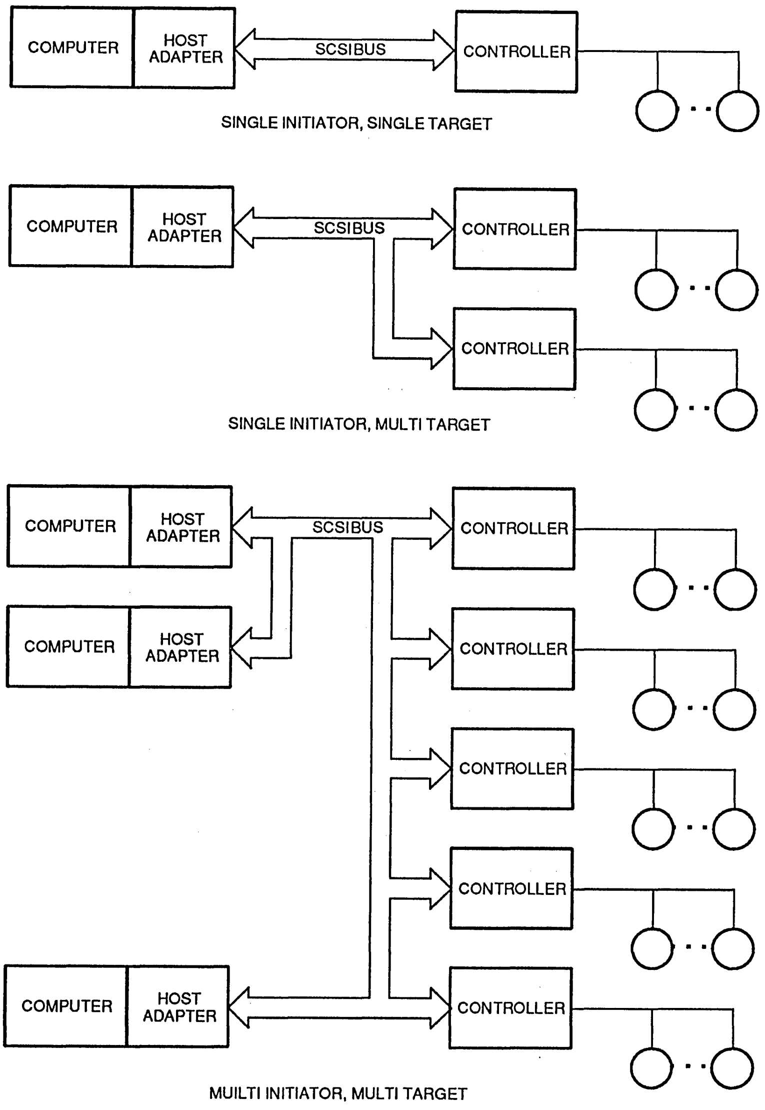
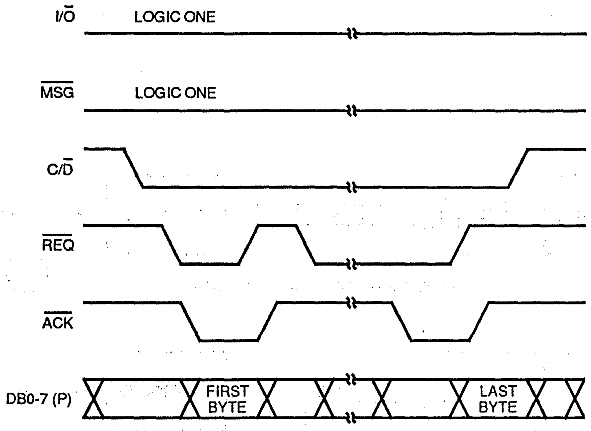
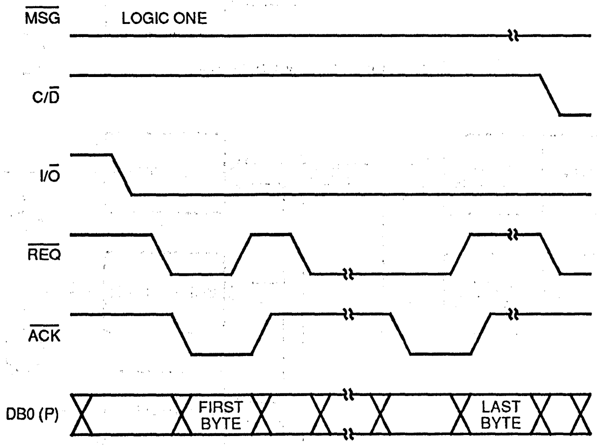


Figure B-5. Possible Bus Configurations



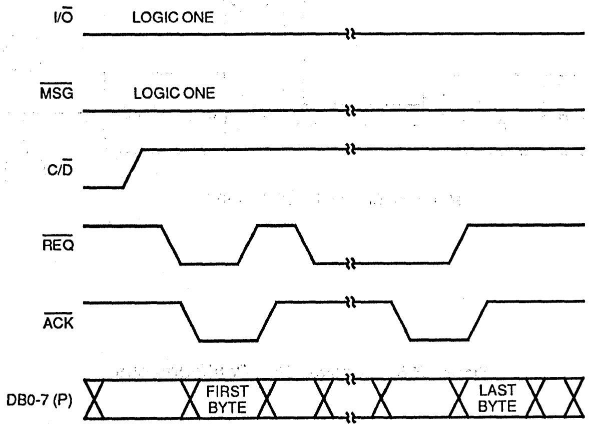
06591A C-6

B-6 SCSI Command Phase Timing



06591A C-7

B-7 SCSI Data Read (from disk) Timing



06591A C-8

B-8 SCSI Data Write (to disk) Timing

APPENDIX C

SOFTWARE SUPPORT

C.1 Am29PL1XX ASSEMBLER

Assembler Features

The Am29PL1XX assembler provides high-level microprogram development support for the Am29PL100 family of parts (Am29PL141, Am29PL142, Am29CPL141, Am29CPL142 and Am29CPL144).

With the inclusion of high-level language constructs, such as IF-THEN-ELSE and WHILE the microprogrammer's task is greatly simplified since the microcode is written in a logical and more natural flowing syntax. In addition, documentation of code is significantly enhanced since the microcode is expressed in a more readable and easy to follow format.

Assembler features include:

- high-level language constructs
  IF-THEN-ELSE
  WHILE
- binary, octal, decimal, and hexadecimal numbers are recognized
- jump/branch to labels
- logic equations for control outputs
- error detection and diagnosis
- default test condition
- JEDEC standard fuse map output
- symbol table output

Error Detection and Diagnosis

Much effort has been made to provide relevant syntax error detection and diagnostic messages in order to facilitate debugging of errors occurring in the microcode. Note that one error may cause spurious errors to propagate through the assembler source file because the assembler expects a certain sequence of symbols. The assembler does not understand the microcode's intent or purpose. Correcting the first error and other meaningful errors will remove spurious errors.

The assembler will check the input file to determine that no conflicts exist in the use of input pins that double as Serial Shift Register (SSR) pins, which are used for customer diagnostics/testing.

System Requirements

The following hardware and software are required to use the assembler:

Hardware (minimum configuration):

- an IBM PC/XT or other PC-compatible, NEC9800, with at least 256K bytes of RAM memory

Software:

- PC-DOS Version 2.0 or higher, or MS-DOS Version 2.11 or higher
- A word processor to create the assembler source file. Any word processor that produces standard ASCII output files is acceptable (example: Wordstar operating in non-document mode).

The following files are on the Am29PL1XX Assembler disk:

Filename	Description
ASM14X.EXE	Am29PL1XX assembler
PL141	Am29PL141 database file
PL142	Am29PL142 database file
CPL141	Am29CPL141 database file
CPL142	Am29CPL142 database file
CPL144	Am29CPL144 database file
COFFEE.EXP	Source file for coffee machine example
MAKE_CPY.BAT	Batch file for making copies and backups

C.2 Am29PL1XX SIMULATOR SIMULATOR FEATURES

The Am29PL1XX simulator provides high-level interactive simulation capability for the Am29PL100 microprograms. Along with the assembler, it helps to verify Am29PL100 designs completely before a device is programmed. The simulator supports functional simulation only. It does not provide any timing simulation.

The Am29PL1XX simulator uses the JEDEC fuse map file (generated by the Am29PL14X Assembler) and a test-vector file as its inputs (Figure C-1).

Based upon the contents of the JEDEC fuse map and the test vector file, it generates "computed output signals" and compares these against expected output values as specified in the test vector file. If any differences are detected, the simulator flags the errors by displaying a "?" under the unmatched output signals.

Am29PL1XX Simulator Distinctive Characteristics

- allows the user to preload or change all internal registers (interactively)
- displays complete status information including all input pin signals, computed and expected output signals, contents of all internal registers
- break point capability
- single step capability
- interactive mode of operation
- another program can be executed during simulation

Simulator Requirements

The following steps ar required to run the simulator:

A. Write and assemble a microprogram source file

Write a microprogram using the Am29PL14X assembler language. Then use the Am29PL14X assembler to assemble the program. The assembler will generate the corresponding JEDEC fuse map file to be used by the simulator.

B. Create test vectors file

The test vectors file can be written in a symbolic format.

Keeping microprogram source and test vector files separate allows one simulation model to have a set of different test vector files.

C. Execute simulation

After the source program is assembled and the test vectors file has been generated, the simulator is ready to run.

The simulator model is designed to reflect the Am29PL100 device as much as possible. Initially, applying a software asserted RESET signal to the simulator is the same as applying a RESET to the Am29PL100 device.

Note that the Am29PL1XX simulator provides functional simulation only - no timing simulation. The simulator assumes 0 propagation delay. However, the clock pulse must be specified as one of the inputs in the test vectors to get register transfers and to compute outputs.

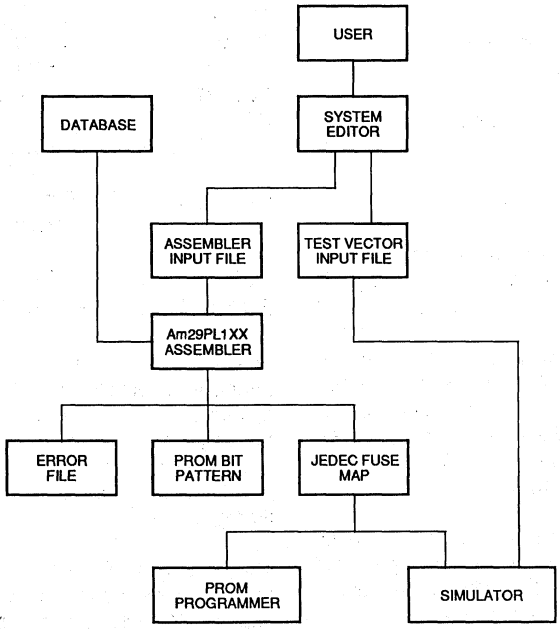


Figure C-1. Simulator/Test Vector Environment

APPENDIX D
REFERENCES

1. *Advanced Micro Devices Programmable Logic Handbook, 1986.*

2. *Advanced Micro Devices Bus Interface Product Specifications, October 1985.*

3. *Advanced Micro Devices Am29PL141 Data Sheet, September 1987.*

4. *Advanced Micro Devices 80188 Data Sheet, October 1985.*

5. *Small Computer Systems Interface (SCSI) Specification* as defined by ANSI X3T9.2 Committee.

6. *Quarter Inch Cartridge (Tape Interface) (QIC-02) Specification.*

7. *PDP-11 Bus Handbook*, Digital Equipment Corporation, 1979.

8. *Microsystems Handbook*, Digital Equipment Corporation, 1985.

APPENDIX E

GLOSSARY OF ABBREVIATIONS/MNEMONICS

141SEL	Am29PL141 Selection (SCSI)
141TPREQ	Am29PL141 Tape Request (QIC-02) Signal
141XFER	Am29PL141 Transfer (QIC-02) Signal
ACK	Acknowledge
ARDY	Asynchronous Ready Line
ARESET	Asynchronous Reset
ATN	Attention
BSYIN	Busy Input (SCSI to FPC)
C/D	Control or Data, SCSI Interface Signal
CC	Condition Code (Input to FPC)
CCMUX	Condition Code MUX to Am29PL141
CMDXFER	Command Transfer Routine
CREG C	Register in Am29PL141 (Count Register)
DACK	Disk Acknowledge (SCSI)
DATN	Disk Attention (SCSI)
DCLK	Diagnostics Clock
DDACK	Disk (SCSI) DMA Acknowledge (to Am29PL141 from 80188)
DDREQ	Disk DMA Request
DIRC	Direction (QIC-02)
DMA	Direct Memory Access
DMAXFER	DMA Transfer Routine
DMSG	Disk (SCSI) Message = MSG C/D (to Int. Status Buffer from Am29PL141)
DREQ	Disk (Data Transfer) Request (to Am29PL141 from SCSI)
DRST	Disk Reset (SCSI)
DSP	Digital Signal Processor
DTACK	DMA Tape Acknowledge (to Am29PL141 from 80188)
DTREQ	DMA Tape Request (to 80188 from Addressable Latch)
EXP	Exception, QIC-02 Interface Signal
FPC	Fuse Programmable Controller
I/O	Input or Output
INT1	Interrupt Number One
ISR	Interrupt Status Register
JEDEC	Joint Electronic Device Engineering Council
LADDR	Addressable Latch
LAN	Local Area Network
LMCS	Lower Memory Chip Select
LPC	Linear Predictive Coding
MCSM	Mid-range Chip Select
MSG	Message SCSI Interface Signal (to Am29PL141 from SCSI)
MSI	Medium Scale Integration
NPR	Non-processor Request
PCS	Peripheral Chip Select
PL	Pipeline
POL	Polarity
RDXFER	Read Transfer Routine
RDY	Ready
RST	Reset
SCSI	Small Computer System Interface
SDI	Serial Data In
SDO	Serial Data Out
SIC-02	Quarter-inch Tape Cartridge Interface
SSR	Serial Shadow Register
TACK	Tape Acknowledge (to Am29PL141 from QIC-02)
TEST41	Am29PL141 Test Vector Generator Program
TOUT	Time Out
TPONL	Tape On Line (QIC-02)
TPRST	Tape Reset (QIC-02)
TRDY	Tape Ready (QIC-02)
TRINT	Tape Ready Interrupt (Addressable Latch to Condition Code MUX)
UMCS	Upper memory Chip Select
VCMD	Valid Command (to Am29PL141 from 80188)
WRXFER	Write Transfer Routine

APPENDIX F

Am29CPL100 DATA SHEETS

The front pages of the latest 29CPL100-family data sheets are reprinted in this section. Complete copies of these technical documents are available from AMD sales offices or authorized representatives.

Preliminary

Am29CPL141/Am29CPL151

CMOS Field-Programmable Controller (FPC)

Advanced Micro Devices

DISTINCTIVE CHARACTERISTICS

- **Implements complex state machines**
- **High-speed, low-power CMOS EPROM technology**
- **Direct plug-in replacement for the bipolar Am29PL141**
- **Seven conditional inputs (each can be registered as a programmable option), 16 outputs**
- **64-word by 32-bit CMOS EPROM**
- **Up to 30-MHz clock rate**
- **Available in a wide selection of 28-pin packages**
  0.6" CERDIP windowed, 0.3" CERDIP windowed, 0.3" plastic DIP OTP, PLCC OTP
- **29 instructions**
  Conditional branching, conditional looping, conditional subroutine call, multiway branch

GENERAL DESCRIPTION

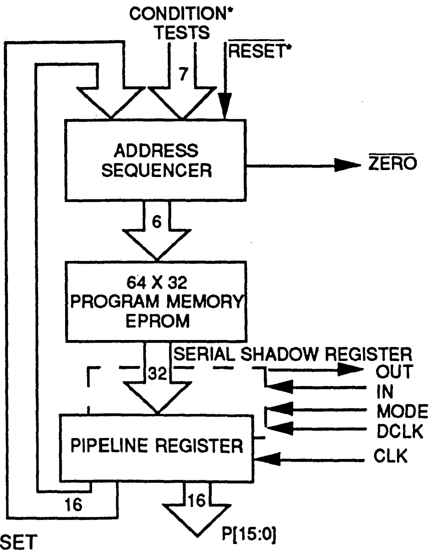
The Am29CPL141, a direct plug-in replacement for the Am29PL141, is a CMOS, single-chip, Field-Programmable Controller (FPC). It allows implementation of complex state machines and controllers by programming the appropriate sequence of instructions. Jumps, loops, and subroutine calls, conditionally executed based on the test inputs, provide the designer with powerful control flow primitives.

Intelligent control may be distributed throughout the system by using FPCs to control various self-contained functional units, such as register file/ALU, I/O, interrupt, diagnostic, and bus control units. An Address sequencer, the heart of the FPC, provides the address to an internal 64-word by 32-bit EPROM.

The Am29CPL151 is electrically and functionally identical to the Am29CPL141 but is manufactured in a space-saving 300 mil DIP package as well as being offered in surface mount packaging.

This UV-erasable and reprogrammable device utilizes proven floating-gate CMOS EPROM technology to ensure high reliability, easy programming, and better than 99.9% programming yields. The Am29CPL141/151 is offered in both windowed and One-Time Programmable (OTP) packages. OTP plastic DIP and PLCC devices are ideal for volume production.

SIMPLIFIED BLOCK DIAGRAM



10135-001A

*Each condition test input can be individually registered as a programmable option; the RESET input can be registered as a programmable option.

Publication #	Rev.	Amendment
10135	B	/0

Issue Date: October 1988

Am29CPL142/Am29CPL152

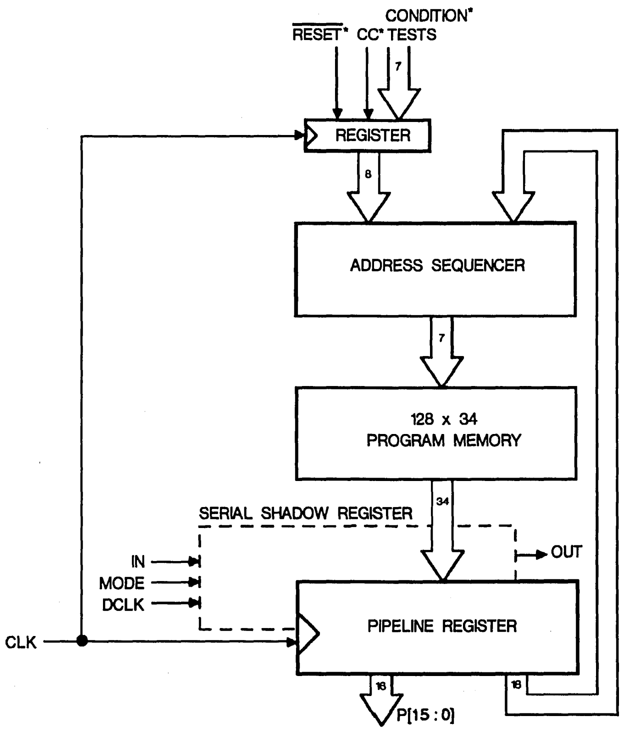
CMOS Field-Programmable Controller (FPC)

ADVANCE INFORMATION

DISTINCTIVE CHARACTERISTICS

- Implements complex state machines
- High speed, low power CMOS EPROM technology
- Eight conditional inputs, 16 outputs
- Each input can be registered or left unregistered as a programmable option
- 128-word by 34-bit CMOS EPROM
- Up to 25-MHz clock rate, 28-pin DIP and PLCC
- 28 instructions
  - Conditional branching
  - Conditional looping
  - Conditional subroutine call
  - Multiway branch
- Output instruction presents counter contents at the control outputs for implementing a larger class of state-machine designs
- A controller-expansion (EXP) cell provides address to external registered PROMs allowing more than 16 outputs
- Am29CPL142 is packaged in a 28-pin 0.6" DIP for upgrade of existing designs
- Am29CPL152 is packaged in a space-saving 28-pin 0.3" DIP or 28-pin PLCC for new designs

SIMPLIFIED BLOCK DIAGRAM



BD007542

* Each test input can be individually unregistered or left registered as a programmable option. The RESET input can also be unregistered as a programmable option.

Publication #	Rev.	Amendment
10796	A	/0

Issue Date: **November 1988**

Am29CPL144/Am29CPL154

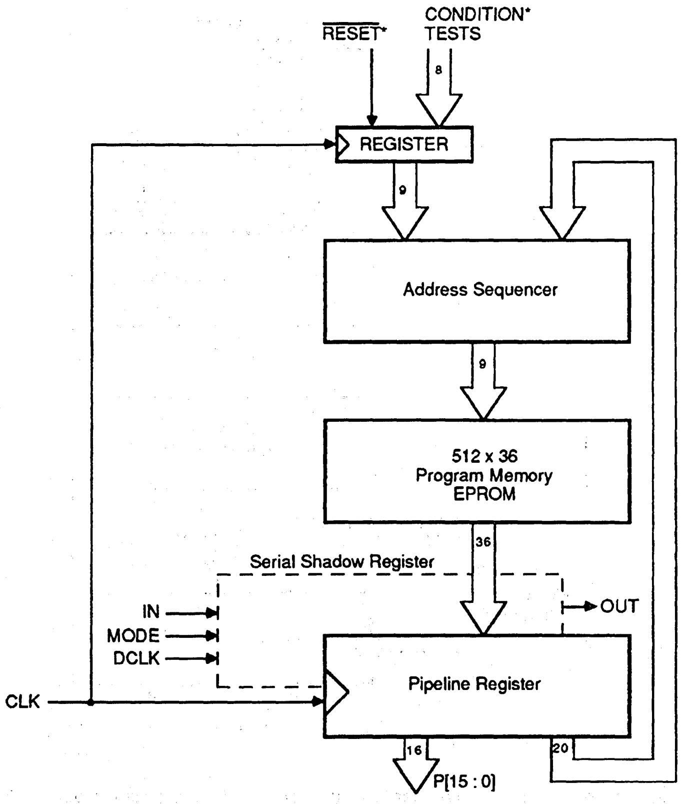
CMOS Field-Programmable Controller (FPC)

ADVANCE INFORMATION

DISTINCTIVE CHARACTERISTICS

- Implements complex state machines
- High-speed, low-power CMOS EPROM technology
- 8 conditional inputs, 16 outputs
- Each input can be registered or left unregistered as a programmable option
- 512-word by 36-bit CMOS EPROM
- 25-MHz clock rate
- Am29CPL144 is packaged in a 28-pin 0.6" DIP for upgrade of existing designs
- Am29CPL154 is packaged in a space-saving 28-pin 0.3" DIP or 28-pin PLCC for new designs
- 28 instructions
  - Conditional branching
  - Conditional looping
  - Conditional subroutine call
  - Multiway branch
- Output instruction presents counter contents at the control outputs for implementing a larger class of state-machine designs
- A controller-expansion (EXP) cell provides address to external registered PROMs allowing more than 16 outputs

SIMPLIFIED BLOCK DIAGRAM



10136A-007A

BD007930

*Each test input can be individually unregistered or left registered as a programmable option. The $\overline{\text{RESET}}$ input can also be registered as a programmable option.

Publication # 10136 Rev. B Amendment /0
Issue Date: **October 1988**

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

APPLICATIONS HOTLINE
(800) 222-9323
(408) 749-5703

Printed in USA

06591B