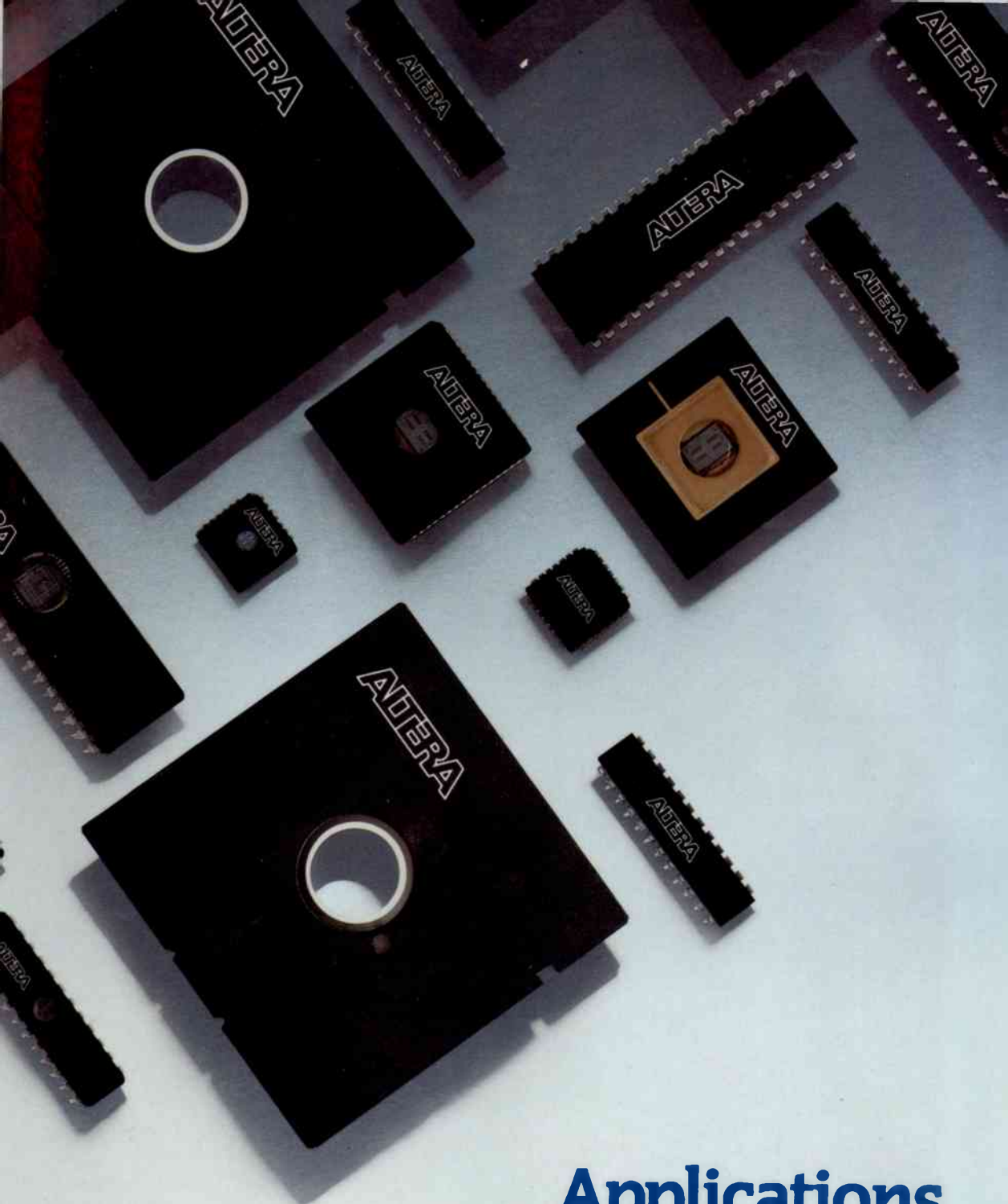


Applications Handbook

USER-CONFIGURABLE LOGIC

APPLICATIONS HANDBOOK

July 1988

ALTERA CORPORATION
3525 MONROE STREET,
SANTA CLARA, CA 95051
(408) 984-2800

SECTION 5: State Machine and Microsequencer Applications — Page No.

SECTION 6: Bus Interface Integration Applications — Page No.

SECTION 7: Multiple Array Matrix EPLDs — Page No.

SECTION 8: Additional Information — Page No.

SECTION 9: Representatives and Distributors — Page No.

General Purpose EPLDs EP-Series

EPLD	PACKAGE	PINS	MACROCELLS (REGISTERS)	BURIED REGISTERS	INPUTS	I/O	Fmax MHz	Icc mA	STANDBY Icc mA
EP1810J	JLCC	68	48	16	16	48	32.0	30.0	0.1
EP1810L	PLCC	68	48	16	16	48	32.0	30.0	0.1
EP1810G	PGA	68	48	16	16	48	32.0	30.0	0.1
EP1800J	JLCC	68	48	16	16	48	20.8	30.0	0.15
EP1800L	PLCC	68	48	16	16	48	20.8	30.0	0.15
EP1800G	PGA	68	48	16	16	48	20.8	30.0	0.15
EP1210D	CerDIP	40	28	4	12	24	26.2	10.0	6.0
EP1210P	OTP DIP	40	28	4	12	24	26.2	10.0	6.0
EP1210J	JLCC	44	28	4	12	24	26.2	10.0	6.0
EP1210L	PLCC	44	28	4	12	24	26.2	10.0	6.0
EP910D	CerDIP	40	24	—	12	24	41.7	20.0	0.1
EP910P	OTP DIP	40	24	—	12	24	41.7	20.0	0.1
EP910J	JLCC	44	24	—	12	24	41.7	20.0	0.1
EP910L	PLCC	44	24	—	12	24	41.7	20.0	0.1
EP900D	CerDIP	40	24	—	12	24	26.3	15.0	0.15
EP900P	OTP DIP	40	24	—	12	24	26.3	15.0	0.15
EP900J	JLCC	44	24	—	12	24	26.3	15.0	0.15
EP900L	PLCC	44	24	—	12	24	26.3	15.0	0.15
EP610D	CerDIP	24	16	—	4	16	47.6	15.0	0.1
EP610P	OTP DIP	24	16	—	4	16	47.6	15.0	0.1
EP610J	JLCC	28	16	—	4	16	47.6	15.0	0.1
EP610L	PLCC	28	16	—	4	16	47.6	15.0	0.1
EP600D	CerDIP	24	16	—	4	16	26.3	10.0	0.15
EP600P	OTP DIP	24	16	—	4	16	26.3	10.0	0.15
EP600J	JLCC	28	16	—	4	16	26.3	10.0	0.15
EP600L	PLCC	28	16	—	4	16	26.3	10.0	0.15
EP512D	CerDIP	24	12	—	10	12	50.0	50.0	0.15
EP512P	OTP DIP	24	12	—	10	12	50.0	50.0	0.15
EP512J	JLCC	28	12	—	10	12	50.0	50.0	0.15
EP512L	PLCC	28	12	—	10	12	50.0	50.0	0.15
EP320D	CerDIP	20	8	—	10	8	50.0	50.0	0.15
EP320P	OTP CIP	20	8	—	10	8	50.0	50.0	0.15
EP310D	CerDIP	20	8	—	10	8	50.0	50.0	0.15

General Purpose EPLDs EPM-Series

EPLD	PACKAGE	PINS	MACRO CELLS (REGISTERS)	BURIED REGISTERS	INPUTS	I/O	Fmax MHz	Icc mA	STANDBY Icc mA
EPM5016	DIP	20	16	8	8	8	TBA		
EPM5024	DIP/JLead	24/28	24	12	8	12	TBA		
EPM5032	DIP/JLead	28/28	32	16	8	16	TBA		
EPM5064	DIP/JLead	40/44	64	36	8	28	TBA		
EPM5127	DIP/JLead	40/44	128	100	8	28	TBA		
EPM5128	JLead/PGA	68/68	128	76	8	52	TBA		

BUSTER Custom Peripheral EPLDs

EPLD	PACKAGE	PINS	MACROCELLS (REGISTERS)	I/O REGISTERS	INPUTS	I/O	Fmax MHz	Icc mA	STANDBY Icc mA
EPB1400D	CerDIP	40	20	32	8	28	40.0	100.0	100.0
EPB1400P	OTP DIP	40	20	32	8	28	40.0	100.0	100.0
EPB1400J	JLCC	40	20	32	8	28	40.0	100.0	100.0
EPB1400L	PLCC	40	20	32	8	28	40.0	100.0	100.0

SAM Stand-Alone Microsequencer EPLDs

EPLD	PACKAGE	PINS	MICROCODE EPROM	BRANCH EPLD	INPUTS	I/O	Fmax MHz	Icc mA	STANDBY Icc mA
EPS444D	CerDIP	24	448x36	768 P-Term	8	12	20.0	120.0	65.0
EPS444P	OTP DIP	24	448x36	768 P-Term	8	12	20.0	120.0	65.0
EPS448D	CerDIP	28	448x36	768 P-Term	8	16	20.0	120.0	65.0
EPS448P	OTP DIP	28	448x36	768 P-Term	8	16	20.0	120.0	65.0
EPS448J	JLCC	28	448x36	768 P-Term	8	16	20.0	120.0	65.0
EPS448L	PLCC	28	448x36	768 P-Term	8	16	20.0	120.0	65.0

Interface Integration EPLDs

EPB2001	JLCC	68	SINGLE CHIP MICROCHANNEL ADAPTOR INTERFACE
EPB2001	PLCC	68	
EPB2002	CerDIP	28	DMA ARBITRATION SUPPORT DEVICE
EPB2002	JLCC	28	
EPB2002	PLCC	28	

Package Abbreviations:

CerDIP = windowed ceramic dual-in-line
OTP DIP = one-time-programmable plastic dual-in-line
JLCC = windowed ceramic leaded chip-carrier
PLCC = one-time-programmable plastic leaded chip-carrier
PGA = windowed ceramic pin-grid array

OVERVIEW **PAGE NO.**

INTRODUCTION

User Configurable (also described as field-programmable or user-programmable) logic devices combine the logistical advantages of standard, fixed integrated circuits with the architectural flexibility of custom devices. These User Configurable devices allow the engineer to electrically program standard, off-the-shelf logic elements to meet the specific needs of each application. This allows in-house design and fabrication of proprietary logic functions without the long engineering leadtimes, high tooling costs, complex procurement logistics, or dedicated inventory problems related to custom Applications-Specific Integrated Circuit (ASIC) approaches (gate arrays, etc.).

Ranging in density from a few hundred to thousands of equivalent gates, Altera's flexible architectures provide cost-effective integration of traditional SSI and MSI TTL functions onto a minimum of board area without the drawbacks of custom circuits. Figure 1 illustrates the wide coverage of design tasks possible with today's advanced User Configurable logic devices from Altera.

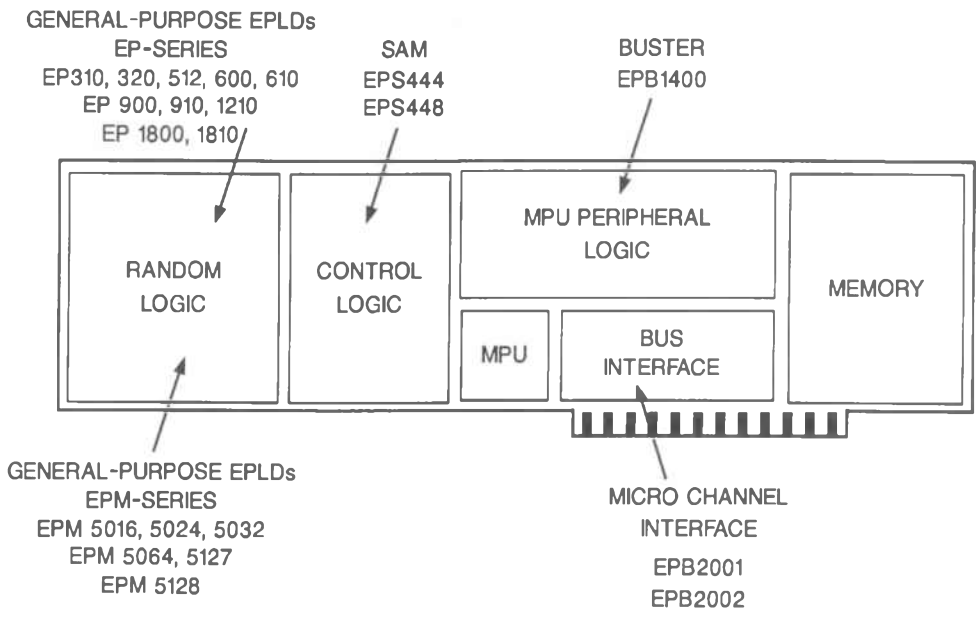
The key to this "off-the-shelf ASIC" capability is use of CMOS EPROM technology. This technology allows creation of a new class of logic elements called EPLDs (Eraseable Programmable Logic Devices). Altera has taken advantage of speed and density advances in CMOS EPROM memory products to create increasingly sophisticated logic devices addressing many logic design problems.

HOW TO USE THIS BOOK

This Applications Handbook is designed for engineers and engineering managers seeking practical ways to reduce design costs, improve design quality and shorten design cycles by taking advantage of advanced User Configurable IC hardware and support software.

Section 1 discusses Altera EPLD's CMOS EPROM technology, EPLD Architecture, and how EPLDs may improve the overall design cycle. Also discussed are the various Altera EPLD families and the class of applications each family addresses. In addition, Section 1 provides background on programmable logic, for those TTL users who are not familiar with this approach to logic design. Section 2 discusses the details of the EPLD design process using Altera software tools. Remaining sections of

Figure 1. Altera User-Configurable Logic Families



this handbook provide detailed applications examples for using each class of EPLD, from simple decoder logic replacement to advanced custom peripheral and state machine designs.

QUICK FUNCTION SPECIFIC IC ENGINEERING

CMOS EPLDs comprise the fastest growing segment of the logic market today. A major reason is the demand for high-density logic solutions with short and easy design cycles, plus the flexibility to support frequent engineering revisions and rapid changes in production schedules. Altera EPLDs satisfy these requirements. In addition to EPLD devices, Altera provides the CAE development tools necessary to take full advantage of the device features.

Altera CAE tools, such as A+PLUS, are easy to learn. They allow mixed format design entry; supporting Boolean, state machine, netlist, and schematic capture input. A full library of over one hundred TTL MacroFunctions allows design input by the method most familiar to many engineers. Design processing using Altera CAE tools is highly automated. Logic minimization, fitting (the equivalent of automatic place and route in PC board or gate array design), utilization reporting, and creation of JEDEC standard programming files are all automatic. This design processing is fast. Less than 15 minutes after completion of design entry (usually 2-3), a programmed EPLD can be in a system for evaluation.

Many Altera CAE tool users have produced functional silicon within hours of receiving their first system. This same development speed is available throughout the product's lifetime to provide rapid enhancements or product variations—often extending the useful life of the product. Hardware design is an iterative process by nature. Altera CAE tools fit this process, allowing easy and rapid design changes. This allows more design refinements before production release to incorporate new ideas, product enhancements, or revisions to eliminate design or specification errors. Altera logic tools and devices can yield better products with dramatically reduced design cycles.

MANUFACTURING SPEED AND FLEXIBILITY

EPLDs also provide benefits to procurement and manufacturing management after the development engineering task is finished. Since ELPDs can be stocked as off-the-shelf devices, they offer the same advantages over "fixed" ASIC solutions which led to EPROMs eventually replacing masked ROMs in the memory arena. Since they can be inventoried as blank units, distributors can easily stock EPLDs to eliminate long procurement lead times and dangers of "line down" situations due to procurement shortfalls. Since customers can stock EPLDs as blank devices, product revisions can be cut into manufacturing without the waste involved in discarding dedicated inventory—a major cost problem common to gate array and other custom chip approaches. The total cost of using EPLDs is often significantly lower than gate arrays, when the real multiple revision, inventory, and other overhead costs are considered.

EPLD FAMILIES

Altera offers families of EPLDs to solve many common board and system integration needs. These User Configurable logic devices are divided into two architectural categories based on a fundamental design decision—should the device architecture yield maximum flexibility for general purpose logic replacement or be specialized to solve a specific system design task:

1) General Purpose EPLDs—provide ideal integration densities for random logic replacement from PAL replacement to thousands of gates. These devices all have the EP prefix (or EPM for MAX devices) plus a number identifying the particular device.

2) Function Specific EPLDs—provide integration of specific system design tasks. These function specific EPLDs are further divided into product families according to their specific system design focus:

- **Custom Peripheral EPLDs**—BUSTER (BUS I/O-regisTER intensive) family EPLDs offer solutions for custom microprocessor peripherals. The devices use an EPB device prefix to identify them as bus-oriented products.
- **Interface Integration EPLDs**—are usually bus-oriented devices designed to provide integration of a specific system interface standard, such as the Micro Channel interface chip set, the EPB2001 and EPB2002.
- **Microcoded EPLDs**—SAM (Stand-Alone-Microsequencer) family devices provide the logic and speed required for complex control logic. These devices use the EPS prefix to indentify them as EPROM programmable sequencers.

As product offerings increase, a wider variety of system design tasks are possible with EPLDs. A variety of package options, including DIP, J-Lead, or PGA are offered. EPLDs are available in windowed ceramic packages for development (erasable), or one-time-programmable plastic versions for high volume production requirements.

GENERAL PURPOSE EPLDs

The General Purpose EPLD family provides a cost-effective means to integrate traditional SSI and MSI TTL functions into a minimum board area without the development delays and expenses incurred with other custom solutions. Members of this family (see Figure 2) range from the EP320 (8 macrocell, PAL replacement), to the EP1810 (48 macrocell, LSI replacement). Each of the General Purpose EPLDs provide dedicated input pins, user-configurable I/O pins, and programmable flip-flop

and clock options. A new High Density EPLD Family, MAX, is previewed later in this Handbook.

CUSTOM PERIPHERAL EPLDs

The EPB1400 was the first member of this family, called BUSTER (BUS I/O-regisTER intensive EPLD). It is a highly integrated, high performance, solution for custom peripheral functions. This family of devices supports direct 24 mA bus drive capability and byte-oriented operation to interface directly to a microprocessor local or system bus.

INTERFACE INTEGRATION EPLDs

The first member of this family is the Micro Channel interface chip set (the EPB2001 and EPB2002). These chips provide the most complete solution available for add-on Micro Channel board development. The EPB2002 provides bus arbitration functions in Micro Channel designs requiring DMA support. These devices provide 24 mA drivers, allowing direct bus connection.

MICROCODED EPLDs

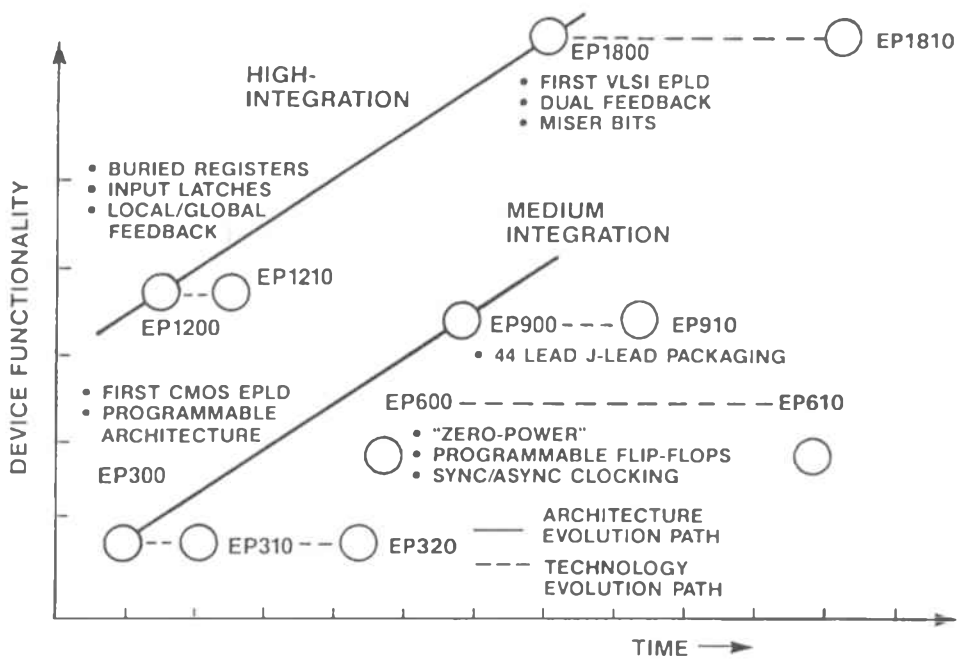
The SAM (Stand Alone Microsequencer) family allows user-configurable solutions to complex state machine and high performance controller applications. These devices offer significant power and space advantages over traditional microsequencers.

DESIGN PROGRAMS USING EPLDS

User configurable logic is very effective for developing custom circuits quickly. Typical applications for these devices include:

- Development and production of systems in the single unit to a few thousand per year volumes. Many examples are found in test equipment, instrumentation, telecommunications and military design.
- Prototype development of complex systems still in the evolutionary stage which may be eventually transfered to ASIC devices for very high volume production.
- Fast time to market products which need VLSI density but cannot wait for multiple gate array iterations.
- Customization of essentially standard systems to meet a specific contract or customer option requirement without incurring expensive PC board or ASIC tooling charges.
- Secure systems that take advantage of the EPLD security cell that acts as a lock to protect the design programmed into the chip. When programmed, it is impossible for a competitor to interrogate the chip in order to reverse engineer the design.

Figure 2. Altera Innovation in General Purpose EPLDs



SOFTWARE TOOLS

Altera silicon and software are developed in concert, rather than starting with a device, then trying to find the development tools needed to support it. This enables Altera to put the features where they're needed—in either software or hardware—and eliminate trade-offs. The goal is efficient software which allows familiar design entry methods and rapid design completion. Software tools are provided on the industry standard IBM PC-AT platform (or compatible). In addition, support for PS/2 machines is planned.

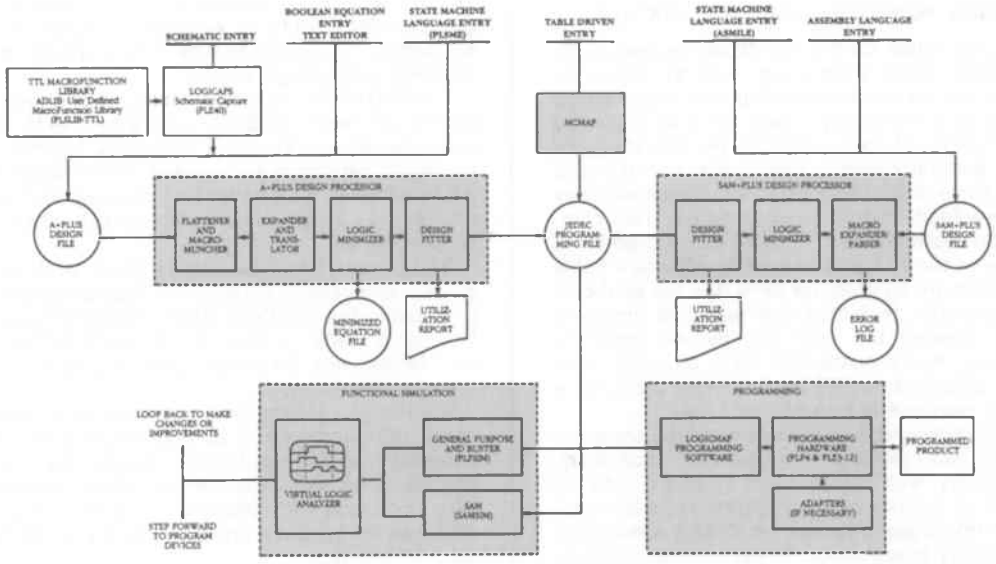
Design entry is available in several forms. Schematic entry allows use of familiar TTL logic symbols—from basic gates to complex MacroFunctions, representing large blocks of logic. State machine entry provides high level entry of control logic tasks. Boolean equation input is convenient for straightforward logic expressions. Altera's design tools allow the engineer to mix and match design entry methods as appropriate within a given design (see Figure 3).

Designers familiar with TTL design may use LogiCaps, the Altera schematic capture program when designing with general purpose or BUSTER EPLDs. LogiCaps provides access to multiple symbol libraries: a complete set of gate constructs, flip-flop and EPLD I/O architectures, and a growing 7400 series TTL/CMOS MacroFunction Library containing over 100 SSI and MSI elements. Since MacroFunctions are familiar, pre-designed and simulated TTL building blocks, using them saves time. Custom symbols may also be created using the ADLIB software. At run time, software automatically eliminates unused portions of a standard MacroFunction to provide minimum logic utilization. LogiCaps also provides design editing features like ten levels of zoom, split-screen capability, real-time orthogonal rubberbanding of connections, and bus and multipage support, to reduce the effort needed for edits and enhancements.

For control logic designs that are best described with state diagrams or state tables, a high-level language called ASMILE was created. It supports a common syntax for building simple state machines (for BUSTER or general purpose devices) or complex state machines and controllers (using SAM).

Once the design is entered, the file is converted to the Altera Design File (ADF) format, then processed into a JEDEC file by the Altera Design Processor (ADP). The design processor provides automation of the remaining design details. Automatic part selection is a user-controlled option.

Figure 3. Altera CAE Design Environment



This, combined with heuristic logic reduction and fitting algorithms, minimizes the delay usually seen between design entry and generation of a testable prototype. A Functional Simulator is also available to allow simulation during the design cycle. Finally, the LogicMap II program translates the JEDEC file into a working part via the Logic Programmer unit.

Altera software tools run on an IBM PC XT, AT or compatible under MS-DOS Version 3.0 or greater. Hardware requirements are one 360K floppy drive, 10-Mbyte hard disk, 640K of RAM, and a full length expansion slot for the programmer card. Only design stations used for device programming need this expansion slot.

DESIGN CYCLE IMPACT

A unique advantage of the user-configurable approach to logic design is the very short product design cycles which are possible. This section describes three examples which use the combination of Altera CAE software tools and EPLD device flexibility to create custom logic functions more efficiently and rapidly than by other means. The first example demonstrates how quickly a first-time user can take advantage of the technology, whereas the others describe capabilities available to an experienced designer. Examples two and three are discussed in detail within their own applications notes later in the book.

1) Comparison of an EPLD vs. a standard cell ASIC solution in general-purpose logic design.
2) A custom microprocessor peripheral design.
3) A high-performance microcoded controller.

GENERAL PURPOSE LOGIC INTEGRATION

In preparation for this Handbook, several Altera customers were interviewed, with the focus on customers familiar with ASIC approaches, such as gate arrays and standard cells. Integration density (the number of logic functions per board square inch) generally favors gate arrays and standard cells, since user-configurable solutions must carry the overhead of programming arrays. This comparison resembles the EPROM vs. masked ROM issues of the 70's and early 80's. Although initial unit costs were higher for EPROMs, the total cost of ownership, including overhead and inventory costs, favored the user configurable solution. Similarly, many customers have changed from ASIC solutions to EPLDs for many projects. A typical response is summarized here.

An engineering group in a Fortune 500 company already had extensive in-house ASIC experience, particularly with "quick turn" standard cells. A new, high priority design program required functional prototypes in six months. Design constraints, particularly board space, power, and manufacturability issues demanded a solution that would integrate most of the random logic on the board into one or two packages.

In order to meet the needs of the project, a high cost, "crash program" to create a single-chip standard cell was selected. Other projects awaiting similar in-house standard cells were delayed as this program gobbled up time in the ASIC design department. Expected throughput time for production-ready parts was nine months.

On reviewing Altera literature, the engineer noticed that two EP1800s could possibly hold the design. Since he was unfamiliar with EPLD design, the engineer called the Altera Applications department. After several phone conversations, EP1800 suitability for the design was established. An A+PLUS development system was ordered on Monday. By Friday of the same week, the engineer demonstrated a functional prototype, using just two EP1800s to contain the entire TTL section of the design. The standard cell program was put on "indefinite hold", as soon as it was realized the EP1800s could compete on cost, when NRE and other overhead charges were spread over the manufacturing lifetime of the product.

CUSTOM MICROPROCESSOR PERIPHERAL

The second example of a custom microprocessor peripheral illustrates the advantages of TTL MacroFunction design entry and highlights the flexibility of the EPB1400, a BUSTER device (see section 4 of this book, for complete details). The design is a buffered, serial data communications transmitter with an integral bit-rate scaler, transmit state machine, and interrupt handshake logic. The configuration operates as a custom communications peripheral to a microprocessor operating at 25 MHz with no wait states at the bus interface. Since the BUSTER family was specifically designed for custom microprocessor peripheral design, the EPB1400 is the target device.

The EPB1400 combines the features of a dense, flexible 20 macrocell EPLD with a byte-wide I/O and internal bus structure optimized for microprocessor peripheral applications. The EPB1400 is designed as an 8-bit peripheral, however multiple devices may be used in parallel to create 16 or 32 bit peripherals.

Design entry took two hours using BUSTER-specific and general purpose TTL MacroFunctions provided in the LogiCaps library. Design processing, using Revision 5.0 of the A+PLUS software, took 90 seconds. Silicon programming was completed in 20 seconds.

Modification of a peripheral function using fixed ASIC technology would be both costly and time consuming at this point in the design. User configurability of BUSTER devices make changes easy. For example, changes to control signal polarities on the microprocessor interface can be made in minutes.

COMPLEX STATE MACHINES AND HIGH PERFORMANCE CONTROLLERS

The SAM, Stand-Alone-Microsequencer, function-specific EPLD architecture is optimized for design

of complex state machines and high performance controllers. SAM combines the high-performance, multiway branch capability of an EPLD with the complex control features of an EPROM-based microcoded sequencer. The first SAM device (the EPS448) is a 28-pin function combining 768 p-terms of branch control logic with an on-chip 448-word by 36-bit EPROM sequencer (see the SAM section of the Altera DataBook for more details).

SAM makes an ideal control element in a custom graphics engine (see section 5 for the detailed design). This example was designed using a micro-assembler design entry approach and SAM+PLUS software. Once the program was defined, initial assembler source file entry took three hours using a standard text editor. Design compilation took 30 seconds. Four entry errors were discovered and corrected in five minutes. Successful compilation took 45 seconds. Programming the two EPS448s required by the design took 55 seconds each.

This application required two EPS448 devices due to output capacity requirements. A significant feature of the SAM design processor is that the user can describe the application as a single control block and the software will partition the function over as many cascaded devices as required. The complex nature of the graphics control microcode required many iterations to create a functional system. As in the BUSTER example, early ability to change is a capability inherent to the EPLD concept, which supports practical engineering needs throughout design program development cycles.

EPLD ARCHITECTURE

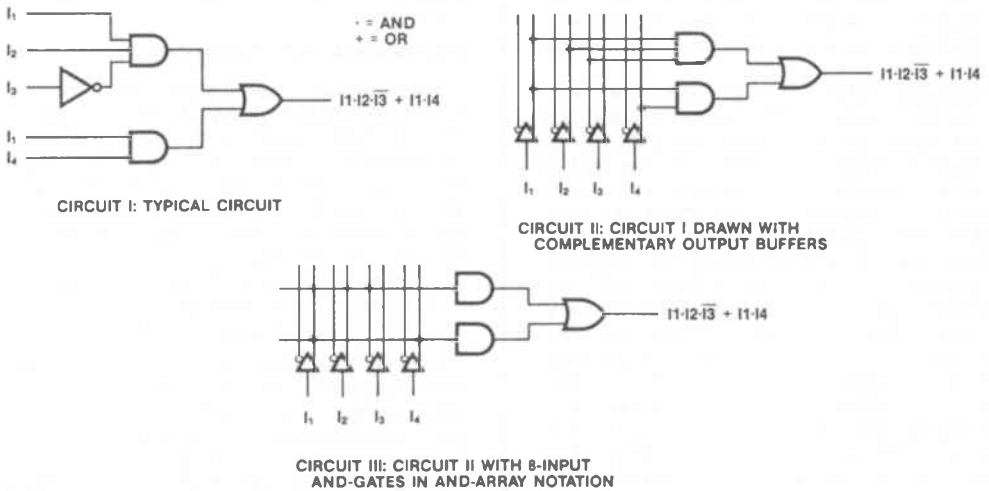
The following architectural discussion is provided for the interested reader. However, an important aspect of the Altera approach to design tool support is to eliminate the necessity of mastering the inner complexities of EPLD architectures. The user may work in a familiar environment, such as TTL MacroFunctions or a high-level state machine language, and the software automatically translates the design into the format required to fit the EPLD architecture.

BASIC CONCEPTS

Altera General Purpose EPLDs provide dedicated input pins, user configurable I/O pins, and programmable flip-flop and clock options to insure maximum flexibility for the integration of random logic functions. Many of these features are also present on Altera function-specific EPLDs such as BUSTER. This section provides a basic discussion of these common architectural features. Refer to the Altera DataBook for detailed architecture and pinout descriptions for each device.

Within an EPLD, there exists an AND array which creates product terms. A product term is simply an n-input AND gate, where n is the number of connections. EPLD schematics use a shorthand "AND-array" notation to represent several large AND gates with common inputs. Figure 4 shows three different representations of the same logic function. Circuit I is presented in classic logic notation, Circuit II has been modified to a sum-of-products notation, and Circuit III is written in

Figure 4. AND-array Notation



CIRCUIT I: TYPICAL CIRCUIT

CIRCUIT II: CIRCUIT I DRAWN WITH COMPLEMENTARY OUTPUT BUFFERS

CIRCUIT III: CIRCUIT II WITH 8-INPUT AND-GATES IN AND-ARRAY NOTATION

AND-array notation. Dots represent connections between vertical wires and an input to one of the 8-input AND gates on the right. No dot implies no connection, the AND gate input is unused and floats to a logic 1.

The 2 by 8 AND-array of Circuit III, can produce any Boolean function of four variables (provided only two product terms are required) when expressed in sum-of-products form. Outputs of the two AND gates in the example are called "product terms" or "p-terms". Any Boolean expression—no matter how complex—may be written in sum-of-products form.

MACROCELL ARCHITECTURE

An Altera EPLD Macrocell consists of such a programmable AND-array, supporting typically 8 product terms, combined with an eight input OR gate, a tri-state buffer driving the I/O pin, with a sequential logic block between the OR gate and output buffer. This sequential logic block consists of a flip-flop (either toggle or D-type, depending on device) combined with programmable multiplexers used to control output, and selection of registered or combinatorial output and feedback. Programmable feedback is supported, either from internal logic or from the I/O pin. Some EPLDs provide a dual-feedback option allowing both internal logic and the I/O pin to feedback simultaneously. This allows the logic to be buried within the Macrocell and simultaneously use the I/O pin as an additional dedicated input pin. Since each Macrocell can have a different configuration, Macrocell groups provide a powerful logic design base.

LOGIC ARRAY

The Logic Array uses a programmable AND, fixed OR PLA structure. Inputs to the AND array come from the true and complement of the dedicate input and clock pins and from the internal feedback paths.

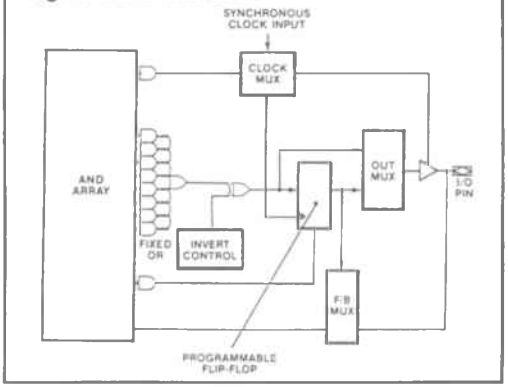
Connections within the Logic Array are made using the AND array. At the intersection point of an input signal and a product-term (AND gate) there exists an EPROM connection. In the erased state, all connections are made. This means both the true and complement of all inputs are connected to each product-term.

Connections are opened during the programming process. Therefore, any product term may be connected to the true and complement of any array input signal. When both the true and complement of any signal is left intact, a logical false results on the output of the product-term. If both the true and complement connection are open, then a logical "don't care" results for that input. If all inputs for the product-term are programmed opened, a logical true results on the output of the product term.

OUTPUT/FEEDBACK SELECTION

For increased flexiblity, General Purpose EPLDs provide programmable I/O architectures. Both the macrocell output and feedback selection can be independently configured. Their operation is controlled by a multiplexer whose selection is determined by the programmed state of EPROM cells as shown in Figure 5. Each macrocell I/O architecture can be individually configured.

Figure 5. I/O Macrocell



A Macrocell and its associated I/O pin can act as a combinatorial output, a registered output, an input, a combinatorial output with combinatorial feedback, or a registered output with registered feedback (the EP310 provides combinatorial output with registered feedback and registered output with combinatorial feedback). In addition, certain devices allow dual feedback on the outputs, allowing the Macrocell to be used as an embedded register, and at the same time function as an input pin.

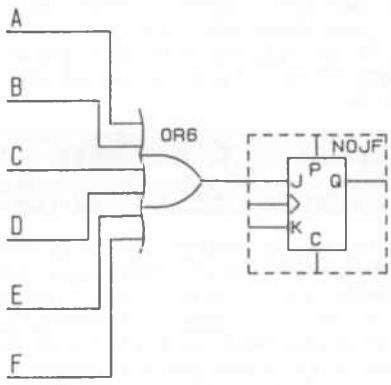
PROGRAMMABLE FLIP-FLOPS

Programmable flip-flops are used in many Altera EPLDs to create a variety of logic functions in an efficient manner. Each flip-flop can be programmed to provide a conventional D-type, JK, Toggle, or RS function. This is coded into the JEDEC file by A+PLUS design software. The designer merely specifies the type desired. Programmable flip-flop bits perform three functions:

1) Inversion control (De Morgan's theorem).
2) Selection of register type.
3) Distribution of product terms.

Selection of inversion controls is determined from results of A+PLUS computations to decide whether true or complement forms of logic will yield reduced demand for product-terms. Figure 6 illustrates a function which would require six product-terms, as drawn. Fortunately, De Morgan's theorem allows reduction to just one product-term.

Figure 6.



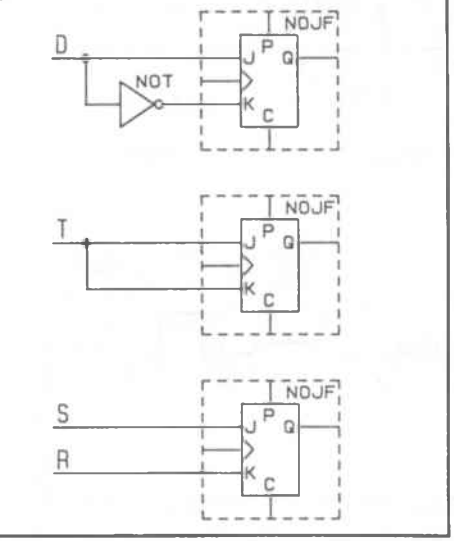
The OR gate can be transformed into a NAND gate using De Morgan's conversion:

A+B+C+D+E+F = /(/A*/B*/C*/D*/E*/F)

This conversion from "OR" to "AND" allows translation of the desired equation and reduces the number of fixed OR terms required in the logic array. This conversion must be done by hand when using most programmable logic tools, before the equation can be entered for device programming. Altera software tools automatically apply De Morgan's translations to optimize use of the EPLD array.

The next programmable function is selection of the flip-flop type. A classical view of flip-flop types presents the JK flip-flop as the master configuration from which many others can be constructed. Figure 7 illustrates Toggle, D-type, and RS flops

Figure 7.



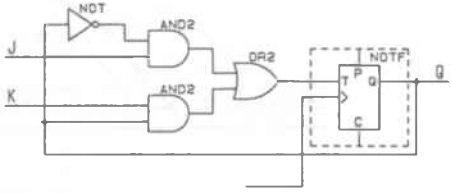
derived from the JK type. From this it would be a natural approach to build a basic JK flip-flop and program it to provide the simpler derivations. This technique has been used in some early generation PLDs, but suffers from a major limitation since there is no provision to distribute product-terms between the J and K inputs. An additional programmable OR array provided this product-term allocation. However, this carried the penalty of additional propagation delay.

Altera EPLDs use a flip-flop that is programmed by a single EPROM bit to operate either as a D or Toggle flip-flop. This allows the EPLD to be programmed as a JK with provisions for inversion control and product-term distribution. Figure 8 shows the Toggle flip-flop used to produce the JK function. The additional gating (for this or other possible flip-flop types) is automatically selected by the A+PLUS design processor, which also handles distribution of terms automatically. This approach does not require the additional programmable logic array, thus eliminating a speed penalty.

Figure 8.

Function Table

J	K	Qn	Qn+1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



PROGRAMMABLE CLOCK

In general purpose EPLDs (except the 310 and 1210) each internal flip-flop may be clocked independently or in user-defined groups. Any input or internal logic function may be used as a clock. These clock signals are activated by driving the flip-flop clock input with a clock buffer primitive (CLKB). In this mode, flip-flops can be configured for positive or negative edge-triggered operation.

External pins which can be programmed as input clocks are provided in certain devices. Direct connection gives enhanced clock to output delay times compared to internally generated clock signals. System clocks are positive edge-triggered with data transitions occuring on the rising edge of the clock.

TRI-STATE OPERATION

The output configuration EPROM bits include support for an output enable function for each output pin. Microcoded SAM devices provide a bit in the microcode to enable or disable the output on each microcode step.

ZERO-POWER/TURBO OPERATION

CMOS technology generally implies lower power dissipation than older bipolar technology. However, the Altera EP600 pioneered true "zero-standby" power operation. Utilizing a unique input-transition detection scheme, the device requires only micro-amp currents during quiescent periods. Using this feature saves power in applications clocked at low to medium frequencies. Each input is connected to a transition-detection circuit consisting of an exclusive-OR, delay element and OR gate. The trigger output of the OR gate activates logic array power on any transition, allowing new input conditions to propagate to device outputs. The logic array is then automatically powered-down to await the next transition. The transition-detection circuitry does add a 30-40% additional delay in the device input/output path. A programmable TURBO-bit is provided to disable the input transition detection circuitry and permanently enable the logic array, giving the user the choice of extra speed versus power consumption. The device also exhibits better system noise rejection characteristics in the "TURBO" mode, and this mode should be used where noisy environments are a problem. These TURBO bits are included in the JEDEC programming file and programmed like any other EPROM bit.

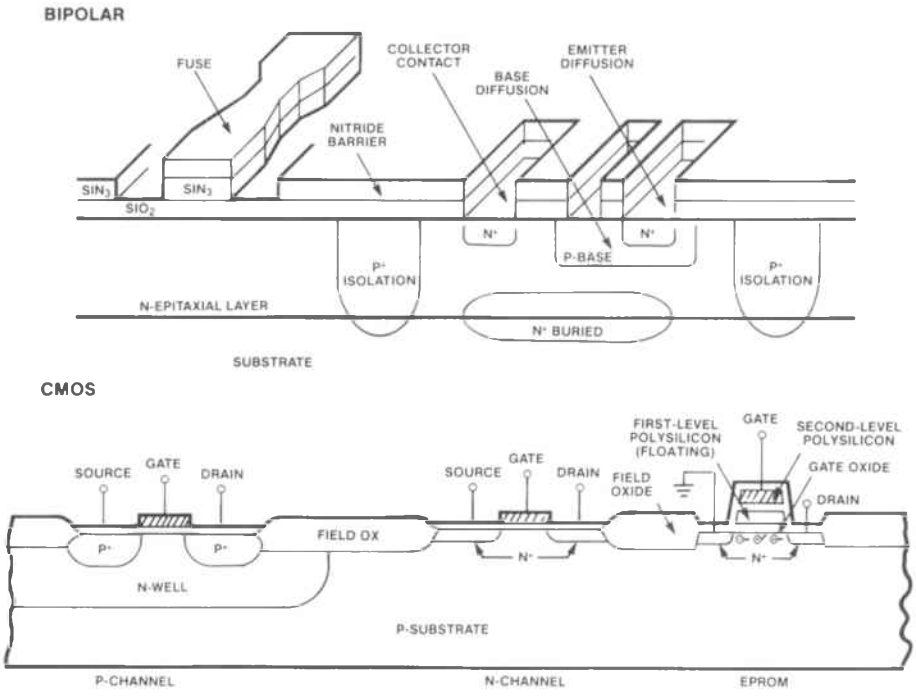
ALTERA CMOS EPROM TECHNOLOGY

Until the invention of the first EPLD by Altera in 1984 (the EP300), the only technology used for Programmable Logic Devices was bipolar and fuse based. The active elements on these devices were constructed from traditional bipolar transistors (ala TTL) with arrays of fuses providing programmable interconnect structures. (See Figure 9). These fuse elements were constructed from a variety of exotic metal alloys and/or polysilicon structures, but all relied upon the physical destruction of the fuse by passing large currents through their small geometries to open connections.

WHY USE CMOS EPROM TECHNOLOGY?

The melting process is difficult to control and resulted in poor and unpredictable programming yields. Since the process is irreversible, guaranteed

Figure 9. Bipolar PLD and CMOS EPLD Technology



results are impossible. The power-hungry nature of bipolar also limited integration levels severely. Altera's pioneering efforts replaced bipolar technology with CMOS, and fuses with reprogrammable EPROM bits. These bits are much smaller in size, electrically programmable and UV-erasable. They allow full factory testing, guaranteeing 100% programming yield at the customer site. CMOS technology also provides low-power operation, allowing higher integration levels.

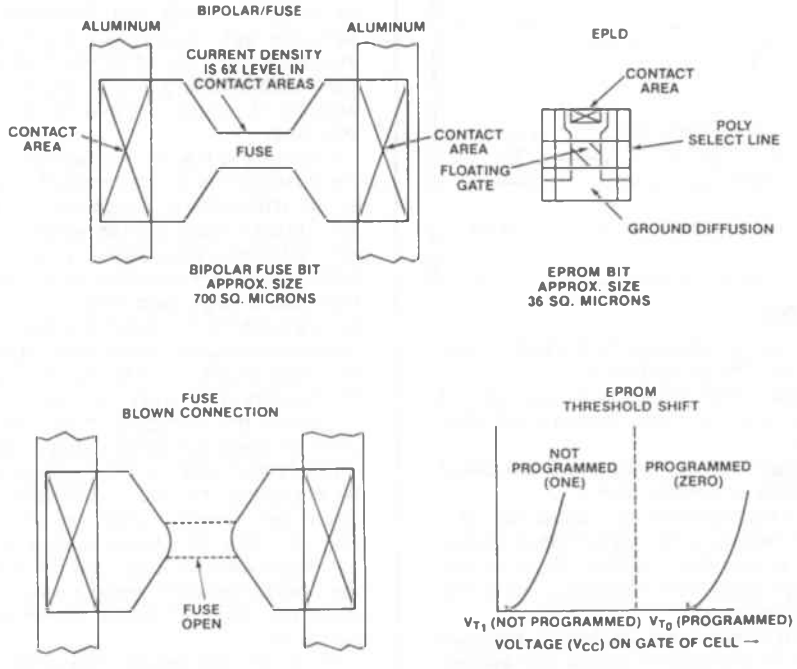
Fig. 10 compares fuse and EPROM cell programming technology. The fuse destructively opens, while the EPROM cell operates via floating-gate charge injection. The programming process consists of placing sufficient voltage (typically in excess of 12 Volts) on the drain of the transistor to create a strong electric field and energize electrons to jump from the drain region to the floating gate. Electrons are attracted to the floating gate and become trapped when the voltage is removed. If the gate remains at a low voltage during programming, electrons will not be attracted and the floating gate will remain uncharged. Trapped charge changes the threshold of the EPROM cell from a relatively low value with no charge present ("erased") to a higher value when programmed.

Within the EPLD programmable array, a sense amplifier or comparator is placed at the end of each product-term line, and by setting a reference voltage into it mid-way between the programmed and unprogrammed levels, the state of the EPROM cells along the product term is sensed and used to select the desired logic function. Low-threshold cells with a logic "one" placed on their select gates (associated input) will tend to pull the product-term line down and cause the logic term to go to a "zero". Transistors with high thresholds will not conduct even when their gates are at a logic "one", and effectively represent a no-connect. This technology, pioneered with EPROM memory in the early 70's, provides reliable, testable programming and operation of Altera EPLDs. Altera devices currently use state-of-the-art 1.0 micron, CMOS EPROM technology, with work underway to move to submicron geometries. As the basic logic array is composed of N-Channel EPROM transistors, device characteristics are optimized to maximize performance of the N-Channel device. This approach minimizes overall input to output delays on the chip.

CELL TECHNOLOGY

Figure 9 shows a basic cross-section of the cell technology. The gate oxide of the transistors is 200 angstroms thick and programmed cell threshold exceeds 6 volts. Basic logic cell size is 24 square microns.

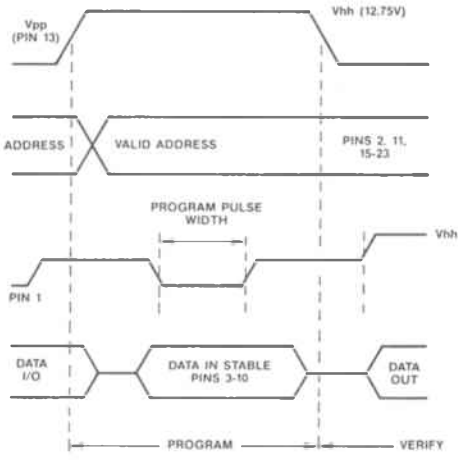
Figure 10. PLD Programmable Elements and Programming Technique



OUTPUT DRIVE CHARACTERISTICS

The CMOS push-pull output stages used on Altera EPLDs provide good AC and DC load driving capability in a system environment. Iol and Ioh specs for general-purpose devices are guaranteed at 4mA or 8mA (depending on the device), adequate levels for all but system bus driving applications. New custom peripheral devices such as the EPB1400 (BUSTER) and EPB2001/2002 (Micro Channel Adapter Chip Set) are designed to directly drive the system bus, providing full 24mA drive capability. AC output characteristics for Altera EPLDs are typically specified with 50pF output loads. Additional output capacitive loading effects the device output delay. The timing parameter used is Tpd (input-output combinatorial delay). Incremental delay per picofarad of capacitance is typically 0.1 ns or less at room temperature.

Figure 11. Program Waveforms

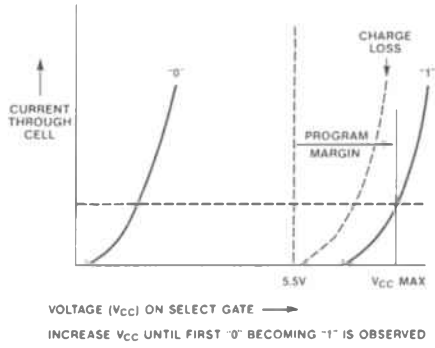


PROGRAMMING

A typical programming cycle for Altera EPLDs is shown in Fig. 11. The procedure is:

1) Raise input select lines for a given row of logic bits to a high Vpp voltage (12.5 Volts nominally).
2) Place either 0 or 12.75 Volts on associated product-term or column lines.
3) Programming algorithm is a sequence of 1 ms pulses separated by program verify cycles.
4) Program pulse width controlled by a Vhh (super-high input level) signal applied to one of the device pins.
5) Once a byte has been verified to program correctly, "overprogram" pulses are applied to doubly insure device programming.

Figure 12. EPROM Cell Margin



The programming operation is done 8-bits at a time. Either Altera-supplied or other reviewed PLD programming hardware may be used. Altera devices also feature a Security (verify-protect) bit which may be programmed to inhibit any verification or interrogation of the device's contents. This may be done as the final stage in the programming process to insure device code security.

DEVICE MARGIN

To insure reliable operation in the user's system, substantial factory testing is performed on all devices prior to shipment. Foremost among these tests are cell margin tests which guarantee the in-service retention of EPROM bit programming. Cell margin testing is a means to determine the amount of charge trapped on the floating gate structure.

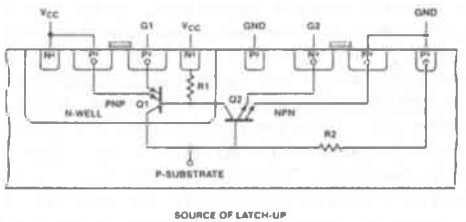
Charge loss results from electrons leaking from the floating gate structure over time and results in a net reduction in programmed cell threshold. Charge gain results from an accumulation of charge on the floating gate, usually as a result of electric fields caused by operation of the device. Charge loss and charge gain mechanisms could effect program retention. To avoid such problems, Altera reliability evaluation has included burn-in of devices at temperatures of up to 250 degrees Centigrade for periods of a week or more. As a point of reference, this corresponds to well over 100,000 years of operation at 70 degrees Centigrade.

Fig. 12 illustrates the concept of program margin. As mentioned previously, EPROM arrays depend upon cell threshold shifts for correct operation. Zero and One I-V characteristics for the EPROM cell are shown. Program margin is a measure of the spread between actual device threshold and minimum required device threshold for correct operation.

To detect cell margin, Altera devices are subjected to special test modes which allow the

external control of EPROM bit gate voltages. By varying this voltage, a measure of cell margin can be obtained. This can accurately monitor cell charge and retention.

Figure 13. Parasitic Bipolar Transistors in CMOS



LATCH-UP

Due to the fundamental structure of CMOS devices, parasitic bipolar transistors are present in the device structure. Typically, the base-emitter and base-collector junctions of these transistors are not forward biased and as a result the transistors are not turned-on. Fig. 13 shows a cross-section of a CMOS wafer with the primary parasitic transistors indicated. By connecting the p-type substrate to the most negative voltage available on-chip (Vss) and the n-type well structure to the most positive voltage on-chip (Vcc), all junctions should in theory remain reverse-biased. However, two factors can alter this ideal state.

As shown in the diagram, besides parasitic transistors, parasitic resistors also occur in the CMOS structure. These resistors are of no concern as long as currents do not flow through the structure laterally. But if any of the associated diodes turn-on for any reason, I-R drops may occur in the structure. The initial turn-on of these diodes usually is the result of power-supply or I/O pin transients which exceed the limits of Vss and Vcc. These transients may be induced by signal ringing and other inductive effects in the system.

The potential problem is that once parasitic structures begin to conduct, the effect is regenerative, reinforcing itself until potentially destructive currents flow. This is the SCR effect called latch-up. Referring back to the diagram, as current flows through the parasitic transistor, the I-R drop through the resistor increases, further forward-biasing the base-emitter junction. The cycle continues until current is limited by drops in the primary current path. However, this current could be at such a level that internal circuitry is permanently damaged.

Altera components have been designed to eliminate the effects of latch-up inducing transients. Under reasonable system operating conditions, all devices are guaranteed to withstand input extremes between Vss-1 Volt and Vcc+1 Volt, and which force currents of 100mA or less through the device pins. To minimize the possibility of inducing latch-up, some general system design guidelines on sequencing of power and inputs to the device are recommended. For example, the normal application sequence of voltages to the device should be:

1. Vss or GND
2. Vcc (+5V)
3. Inputs

When removing power from the device, the sequence should be executed in reverse: first remove or take inputs low, then remove or lower Vcc. Simultaneous application of inputs and Vcc to the device, as might occur as a power supply ramps during power-up, should be safe. Care should be taken to insure inputs cannot rise faster than supply under extreme conditions.

In some applications, boards are "hot-socketed" in the field. To insure under these conditions that latch-up-inducing levels are not applied to the EPLD, the circuitry shown in Fig. 14 may be used. This normally would be required only if the EPLD has inputs tied directly to the edge connector. The diodes clamp the inputs at acceptable levels and the series resistor further limits the injection of current into the EPLD input and clamp diodes. The result is an interface giving maximum protection.

Figure 14. Hot Socket Protection

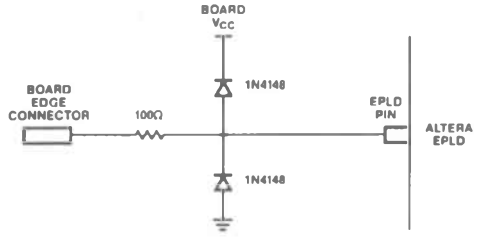
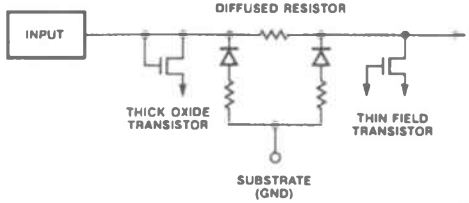


Figure 15. EPLD Input Protection Structure



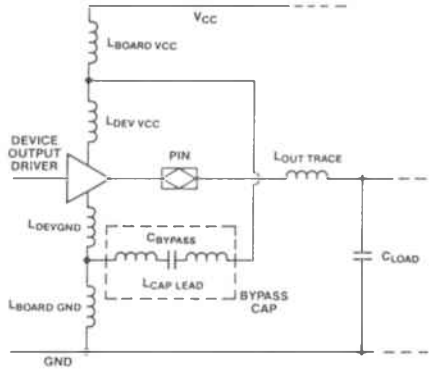
ELECTROSTATIC PROTECTION

Electrostatic Discharge (ESD) can cause device failure when improper handling occurs. EPLD handling during the programming cycle increases exposure to potential static-induced failure. Volt-

ages into the tens of kiloVolts can be generated by the human body during normal activity. Wearing ground straps during device handling and grounding all surfaces which come in contact with components reduces the likelihood of damage.

Altera components employ special structures to reduce the effects of ESD at the pins. Fig. 15 shows a representation of a typical input structure. Diode structures, as well as specialized field-effect transistors shunt harmful voltages to ground before destructive currents will flow. Altera devices typically withstand ESD voltages in excess of 2 kiloVolts, and are thus safe under normal handling conditions.

Figure 16. Board-Level Noise Problem



SYSTEM NOISE

Large switching currents can flow through supply and output pins during high-performance operation. If a 50pF capacitor is charged from 0 to 5 Volts in 10 ns, a dynamic current of 25mA will flow. If 24 outputs on an EPLD switch simultaneously, as might be the case with an EP900, the total transient current can exceed 600mA! This can severely degrade Vcc supply voltage due to inductive effects in the device and system environment. Fig. 16 shows the distribution of typical inductances which can contribute to the problem.

The key to controlling these inductive effects is to adequately decouple the Vcc supply to ground at each device by a suitable capacitor or combination of capacitors. This capacitor can then act as a reservoir of charge to supply the transient switching needs of the device. It is recommended that a 0.1 to 0.2 microfarad capacitor (depending on device, see the appropriate device Data Sheet) be connected from each Vcc pin to ground at the device. High quality capacitors with low internal and lead inductance should be used (monolithic ceramic or tantalum), and leads must be kept short to limit series inductance which degrades capacitor effectiveness. Careful decoupling of the power supply is just good design practice. Decoupling is particularly important in devices capable of large current drive, such as the EPB1400, a BUSTER device.

2

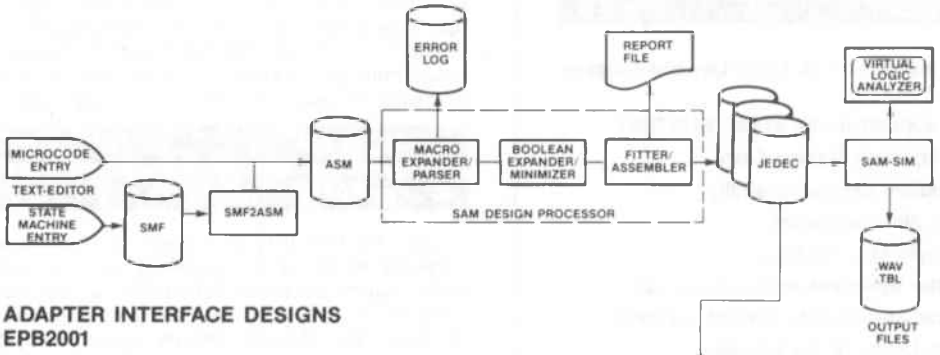
DEVELOPMENT TOOLS	PAGE NO.

Installed on an IBM-XT, AT or compatible machine, Altera EPLD development tools provide a fast, flexible and easy to learn CAE development environment.

They may be purchased as complete development systems or as individual software and hardware products.

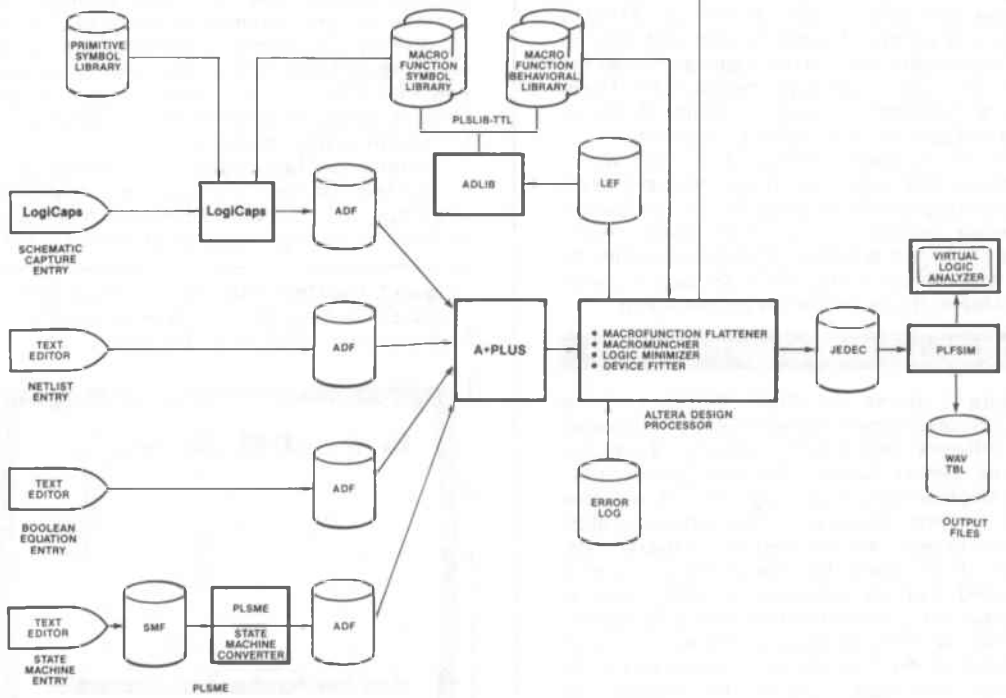
EPLD designs may be entered in many convenient formats. These include schematic capture (basic gates through TTL MacroFunctions), Boolean equations, State Machine and microcode assembler entry. Design Compilation performs logic minimization, automatic device fitting and generation of programming data in the standard JEDEC format. Device fitting is the PLD equivalent to an automatic place and route capability and is accomplished on a typical design in minutes. Design verification and device programming capabilities are also available. Altera development systems permit the use of many third party software and hardware products via appropriate interface programs.

STAND ALONE MICROSEQUENCER DESIGNS
EPS444, 448



ADAPTER INTERFACE DESIGNS
EPB2001

GENERAL PURPOSE EPLD DESIGNS
EP310, 320, 512, 600, 900, 1800
EP610, 910, 1810
EPB1400



FEATURES

- **Software support for all Altera General-Purpose EPLDs.**
- **Software support for EPB1400 (BUSTER).**
- **Boolean Equation Design Entry.**
- **MacroFunction design capability.**
- **Automatic pin assignments.**
- **SALSA Logic Minimization.**
- **Device fitter optimizes device resources.**
- **Support for user-defined MacroFunctions.**
- **Schematic Design Entry interface.**
- **State Machine entry interface.**
- **Functional-Simulator interface.**

GENERAL DESCRIPTION

A+PLUS, Altera Programmable logic user software, contained in the PLS2 and PLS4 products, is a series of software modules that transform a logic design into a programming file for Altera's general-purpose and function-specific EPLDs. A+PLUS supports a variety of input formats that may be used individually or combined together to meet the needs of a particular logic design task. These include Schematic Capture, State Machine, Boolean Equation, and Netlist Design Entry.

A+PLUS includes a Design Processor which transforms the input format to optimized code used to program the targeted EPLD. The Design Processor implements logic minimization, automatic EPLD part selection, architecture optimization, and design fitting. A+PLUS also includes LogicMap software for device programming.

FUNCTIONAL DESCRIPTION

Figure 2 shows the Block Diagram of the A+PLUS development software. A+PLUS accepts four different design entry formats: Schematic Capture, Netlist Capture, Boolean Equations, or State Machine input. The designer is not restricted to just one entry method but may 'mix-and-match' methods to best meet the needs of the overall logic design. If necessary, the design entry format is converted to an Altera Design File (ADF) which is the common entry format for the A+PLUS software. The ADF is then submitted to the Altera Design Processor (ADP). The ADP is composed of a set of modules integrated together that produce an industry standard JEDEC code used to program the EPLD. The Design Processor also produces documentation showing minimized logic and EPLD utilization. Once the JEDEC file is produced, the user may functionally simulate the design. Finally, the user can program the EPLD with the LogicMap programming software and Altera programming hardware or qualified third party programmers.

DESIGN ENTRY

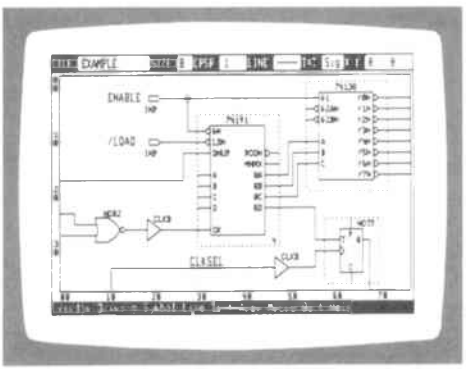
SCHEMATIC CAPTURE

Logic Designs may be entered from schematic drawings by using the LogiCaps or other schematic capture packages. Schematic capture design entry allows the user to quickly construct a wide range of logic circuits. Designs entered with this method use library primitives in the form of low-level functions (input, basic gates, flip-flops, and I/O primitives) to high level TTL MacroFunctions. LogiCaps is mouse-driven and supports hardcopy printout and plots. As required, the schematic representation is converted to an ADF file and processed by the A+PLUS software. For a more detailed description see the PLE40 LogiCaps data sheet.

LogiCaps is a high performance schematic capture package that has been optimized for entering designs destined for Altera EPLDs. It is the primary design entry platform for any member of the Altera EPLD family. When used in conjunction with TTL and user defined MacroFunction libraries, LogiCaps becomes the essential tool for the design of high density EPLDs.

The optional Altera design library is a collection of high level MSI building blocks which allow the LogiCaps user to enter designs in a "block manner". An initial primitive symbol library contains

Figure 1. LogiCaps—*MacroFunction Libraries allow EPLD designs to be entered in LogiCaps using popular 7400 series symbols.*



basic gate, flip-flop and I/O symbols as well as the most commonly used TTL SSI and MSI functions. Other design libraries include an extensive TTL 7400 series symbol library, and user-defined libraries. In addition, each library also contains logic functions not available in standard TTL or CMOS devices. Examples include counters implemented with toggle flip-flops, combination up/down counter with left/right shift register, and inhibit gates.

NETLIST ENTRY

The A+PLUS software directly supports netlist entry via the Altera Design File (ADF). Using a standard text editor, a netlist which describes the circuit is created by using a simple, high-level, design language. The netlist may contain basic gates, I/O architectures, boolean equations, and TTL MacroFunction descriptions. In addition, user defined comments and white space may be freely employed throughout the ADF file. The completed file is then submitted to the Design Processor. This entry method also permits circuit designers to utilize netlist outputs (e.g. from workstations or schematic capture packages not support by A+PLUS) that have been translated into ADF format.

BOOLEAN EQUATION

The Altera Design Processor compiles Boolean equation designs that are written in a simple design language. The source for the design may be created with any convenient text editor. The language supports free-form entry of all syntactical elements. Boolean equations need not be entered in sum-of-products form since the Design Processor will expand equations automatically. The multi-pass design processor/compiler has the ability to support intermediate equations. This feature permits significant reduction in the size of the Boolean equation source code and allow the designer to define the logic in the most natural conceptual manner.

STATE MACHINE

Designs that are easily represented with state diagrams may be entered via the State Machine approach. This method uses a high-level language description featuring IF-THEN constructs, Case statements and Truth Tables. This design entry supports both Mealy and Moore state machines. Outputs of the state machine may be defined conditionally or unconditionally allowing flexible output structures that can be merged with other portions of the design. In addition, multiple state machines may be linked within the same design. Boolean equations are allowed offering the definition of high level intermediate logic expression. The software will also selects the optimum flipflops for the particular design. For more information on State Machine design entry see the PLSME data sheet.

DESIGN PROCESSOR

The Altera Design Processor (ADP) consists of a series of modules that translate design information from a variety of input sources into a JEDEC Standard File used to program the EPLD. This process is automatic and requires minimal assistance on the part of the circuit designer.

DESIGN FLATTENING

A+PLUS accepts design files from one or more of the design entry methods. Once the design has been submitted, the first function of the Altera Design Processor is to "flatten" the design from high-level MacroFunctions to low-level gate primitives. In order for designs to e flattened, information from the MacroFunction Behavioral Library is transferred to the design flattener, which in turn decomposes all MacroFunctions to their primitive gate equivalents.

MACROMUNCHING AND DEFAULT MODES

Once the design is flattened, the design processor analyzes the complete logic circuit and removes unused gates and flip-flops from any MacroFunction employed. This "MacroMuncher" allows the logic designer to freely employ high-level building blocks from the MacroFunction Symbol Libraries without the headaches of optimizing their use.

When MacroFunctions with unconnected inputs are detected, the design processor assigns "intelligent" default values. In general, active-high inputs default to GND and active-low inputs default to V_{CC} when left unconnected. This default mode is activated simply by leaving unused inputs without connections, thus eliminating "busy work" and enhancing productivity.

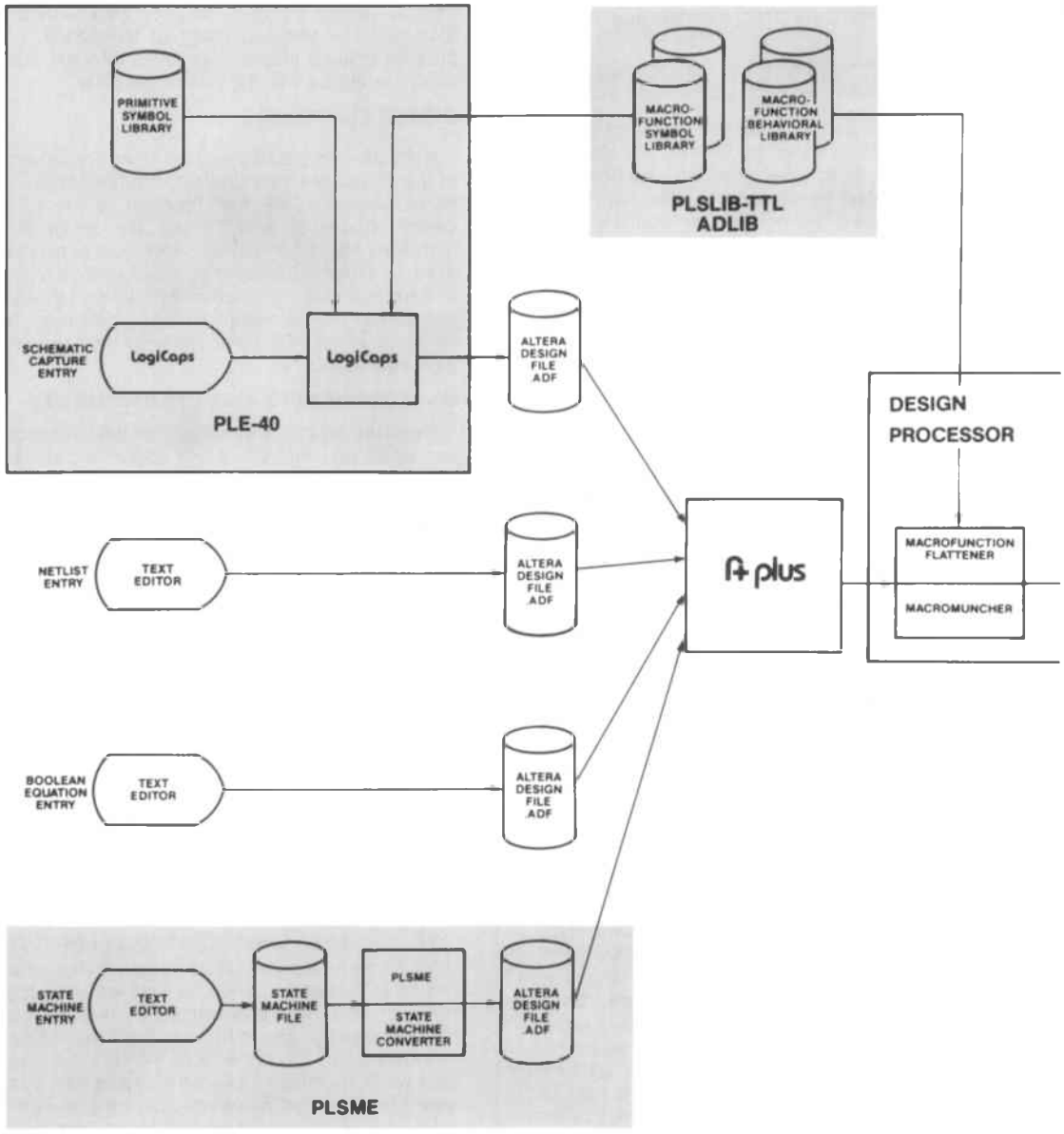
Once the design has been flattened, "munched", and all default values have been assigned, a secondary design file (SDF file) is produced for further processing.

TRANSLATION/MINIMIZATION

The Translator takes the SDF file and checks for logical completeness and consistency. For example, the Translator validates that no two logic function outputs are shorted and that all logic nodes have an origin. In the event that the designer has chosen an EPLD name of "AUTO", the Translator will automatically select the appropriate EPLD based on the logic requirements of the design.

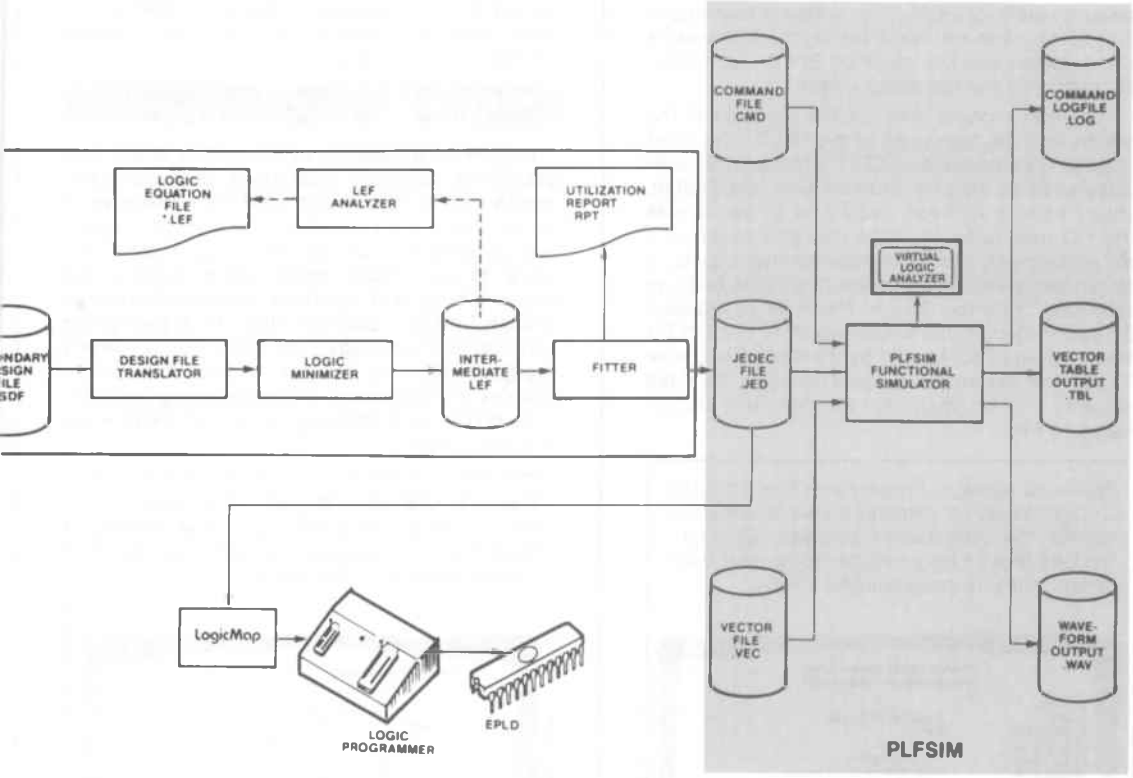
Logic minimization of designs is provided by the Minimizer module. Minimization phases include Boolean minimization with a SALSA (Speedy Altera Logic Simplifying Algorithm) that yields superior results to other heuristic reduction tech-

Figure 2. A+PLUS EPLD Design Environment.



RECOMMENDED COMPUTER CONFIGURATION:

IBM AT and compatible machines
Color graphics or Enhanced graphics display (with extended memory)
640k bytes of main memory
20M byte hard disk drive and floppy-disk drive
MS-DOS or PC-DOS versions 3.2 or later releases
Full-card slot for programming card



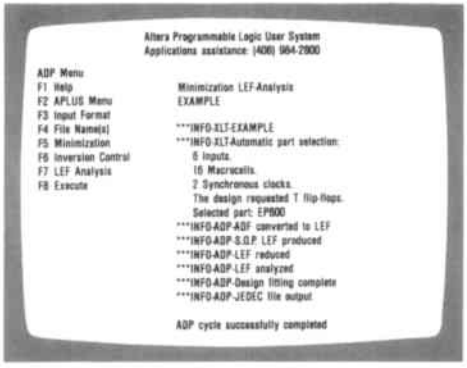
niques. DeMorgan's theorem inversion can be applied automatically to equations. The processor contains algorithms based on artificial intelligence techniques to select candidate equations that will best be represented by a complemented AND/OR function. This feature significantly reduces product-term demands that can be generated by complex logic functions. For Altera EPLDs with selectable flip-flops, the Minimizer checks which type of flip-flop yields a more efficient solution and converts architecture if necessary. The minimized logic can then be passed to the Analyzer module which converts the file into human-readable format allowing the designer to examine the minimized logic.

DESIGN FITTING

The fully minimized design is now transferred to the Fitter. This fitting routine relies on algorithms based on artificial intelligence software techniques in order to place and route the logic requirements of the design into the specified EPLD, automatically providing full pin assignments.

The Fitter module matches the requests of the design with the resources of the EPLD. The Fitter process encompasses all EPLD architectural attributes such as variable product term distribution, programmable flipflops, local and global busses and I/O requirements. If the designer specifies a pin assignment, the Fitter matches the request. If no pin assignments are made, the Fitter finds an optimized fit for the design. The Fitter produces a Utilization Report that shows which of the EPLD's resources were consumed by the design and how. Finally, the Assembler module converts the fitted requests into an image for the part in a JEDEC Standard File.

Figure 4. Design Processor—*The A+PLUS Design Processor displays status information during the compilation process. Complex designs require only minutes to convert from design entry to programmed EPLDs.*



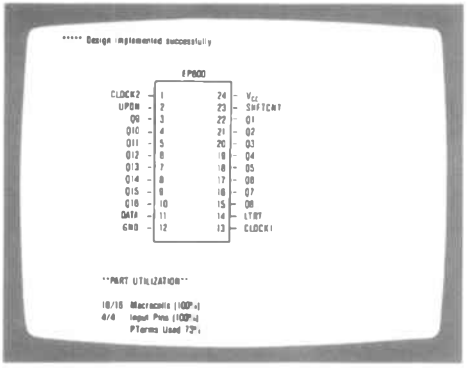
LOGICMAP II

LogicMap II is the interface software that programs EPLDs from JEDEC files created by the Altera Design Processor. The program uses the Altera Super Adaptive Programming algorithm ASAP which significantly reduces device programming times. LogicMap II fully calibrates the programming environment and checks out the programming hardware when initiated. In addition, the program allows the designer to review the JEDEC object code generated by the Altera Design Processor in a structured manner. The program is fully menu driven and provides views of the device object code through a series of hierarchial windows. This feature permits low-level observation and editing of the design, viewed from a perspective similar to that of the logic diagram of the device in the datasheet. Individual EPROM bits within each Macrocell structure may be examined or changed if desired.

HARDWARE

LogicMap software is used to drive Altera programming hardware comprised of a software-configured programming card that occupies a single slot in the computer. Programming signals are transmitted to an external programming unit via a 30 inch ribbon cable and connector. The programming unit contains zero-insertion-force sockets for easy device insertion. All programming waveforms and voltages are derived by the Altera programming card so that no additional power sources are necessary. A programming indicator lamp on the programming unit is illuminated when the unit is active.

Figure 5. Utilization Report—*The Utilization Report documents which of the EPLDs resources have been utilized. Shown below is a small portion of the report.*



FEATURES

- **Graphical entry of General Purpose EPLD designs.**
- **Powerful editing features such as tag and drag, rubberbanding, split screens, and multiple zoom levels.**
- **Supports multiple design libraries including TTL and User-Defined MacroFunctions.**

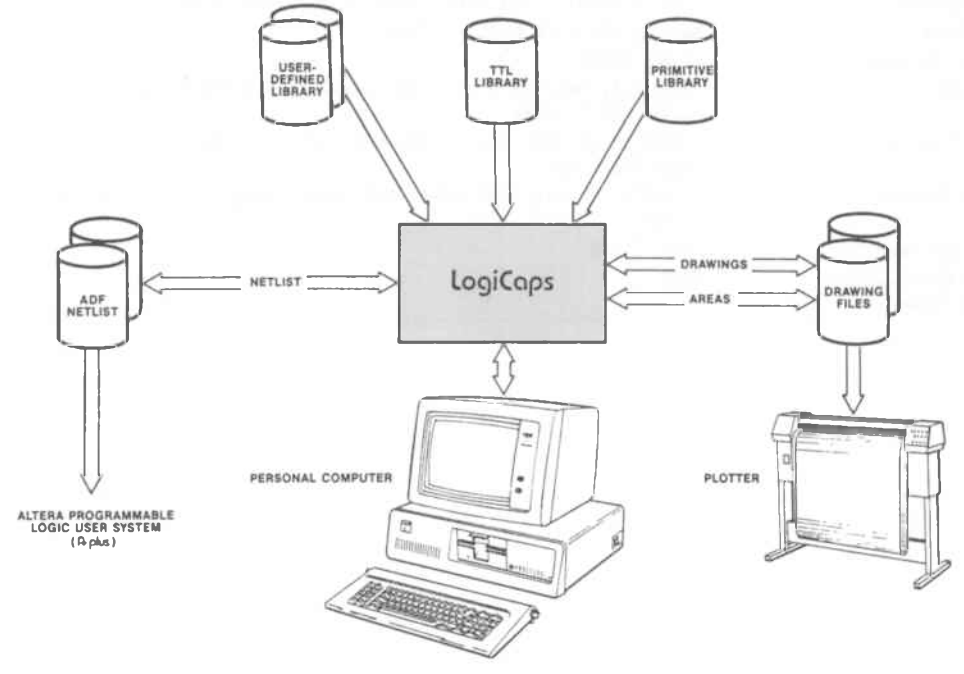
INTRODUCTION

Logic designs are most often conceived and expressed as schematic diagrams, consisting of SSI and MSI TTL components. These designs then need conversion to Boolean equations to be placed within an EPLD. Graphical entry of Altera EPLD designs is accomplished with LogiCaps schematic capture program. With LogiCaps, a personal computer, and a 3 button mouse, EPLD designers can enter schematics using basic logic gates, 7400 TTL symbols, or "user-defined" custom libraries. LogiCaps automatically converts the schematic into a netlist (ADF file). The ADF is then submitted to A+PLUS for design processing. Schematics can be plotted or printed for documentation. See PLE40 data sheet in the Altera Databook for a complete description of LogiCaps.

LOGICAPS FEATURES

- Graphical Entry of Logic Schematics.
- Runs on IBM PC or compatible.
- Directly Interfaces to Altera A+PLUS software system.
- Easy Mouse, Key, and Menu commands.
- Extensive on-line documentation.
- Dual Window display mode.
- Multiple ZOOM levels.
- Orthogonal Rubberbanding of lines.
- Draw schematics up to 90" by 90".

Figure 1.



LOGICAPS FEATURES (cont'd)

- Tag and Drag editing.
- Area edit, save and load.
- Schematic plotting with HP plotters or EPSON printers.
- Support for CGA, EGA, VGA, and Hercules Graphics Cards.

MULTIPLE LIBRARIES

LogiCaps supports multiple design libraries:

1. Primitive Library	Basic gates, Flip-Flops, Input and I/O pins.
2. TTL MacroFunctions	120 7400 TTL functions which have been optimized for EPLD architecture.
3. User-Defined MacroFunctions	Custom logic functions built from the libraries above.

FUNCTION	AVAILABLE TTL MACROFUNCTIONS
AND-OR Gate	7452
ALU	74181
BUS MacroFunction	COLI8, CORI8, RBUSI__A, RINP8__A
Comparator	8MCOMP, 7485, 74518
Converter	74184, 74185
Counter	16CUDSLR, 4COUNT, 8COUNT, FREQDIV, GRAY4, UNICNT2, 7493, 74160, 74160T, 74161, 74161T, 74162, 74162T, 74163, 74163T, 74190, 74190T, 74191, 74191T, 74192T, 74193T, 74393
Decoder	7442, 7443, 7444, 7445, 7446, 7447, 7448, 7449, 74138, 74139, 74154, 74155, 74156
Full Adder	8FADD, 7480, 7482, 7483, 74183
I/O MacroFunction	CO2F, INPN, JO2F, RO2F, SO2F, TO2F
Latch	NANDLTCH, NORLTCH, 7475, 74116, 74259, 74279, 74373
Multiplexer	21MUX, 74147, 74148, 74151, 74153, 74157, 74158, 74298
Multiplier	MULT2, MULT24, MULT4, 74261
Parity Generator	74180, 74280
Register	7470, 7471, 7472, 7473, 7474, 7476, 7477, 7478, 74173, 74174, 74175, 74178, 74273, 74374
SSI Function	CBUF, INHB, 7400, 7402, 7404, 7408, 7410, 7411, 7420, 7421, 7427, 7430, 7432, 7486
Shift Register	BARRELST, 7491, 7494, 7496, 7499, 74164, 74165, 74166, 74179, 74194, 74198
Storage Register	7498, 74278
True Complement/ 0/1 Element	7487

FEATURES

- **Simple syntax rules.**
- **Supports all EPLD architectures.**
- **Standard logic operators.**

INTRODUCTION

Boolean equations have been the standard way of entering designs for low density programmable logic. Altera continues to offer this means of creating designs, and has added specific enhancements in order to take advantage of the advanced architectures in Altera EPLDs.

ADF SYNTAX

Using an ASCII text editor (in non-document mode), Boolean design files are created by following the Altera Design File (.ADF) syntax. The ADF file is broken down into sections, as described below. Each section is defined by a keyword followed by a colon, written in all capitals, and starting at the extreme left hand side of the page. An example is shown in Figure 1.

The ADF file in submitted to the A+PLUS Design Processor which checks all syntax, performs logic minimization, and design fitting into the EPLD. A+PLUS then produces a JEDEC file used to program the device.

HEADER SECTION

The Header section includes all the bookkeeping information, such as design title, revision, designer, and any other desired information. This section is optional and has no bearing on the design itself.

OPTIONS SECTION

The Turbo-Bit and Security Bit is controlled here.

PART SECTION

This section specifies the targeted EPLD. "AUTO" can be used, which allows the A+PLUS to automatically select the best EPLD for the design. If pin assignments are given, then the AUTO option cannot be used since the pin assigments must correspond to a particular EPLD.

INPUTS SECTION

All EPLD inputs for the design are specified here. Pin numbers can be assignd by appending an "@" to the end of the pin name. See Figure 1.

OUTPUTS SECTION

The Outputs section declares all outputs used in the design. Output pin names may also be assigned pin numbers in the same way as input pins, by using the '@' symbol. Bi-directional pins are also declared in this section and must not be declared in the INPUTS section.

NETWORK SECTION

The Network Section defines the programmable I/O architecture of the EPLD. This sections allows you to access all of the advanced I/O architecture features such as programmable flip-flops.

EQUATIONS SECTION

This section contains the Boolean equations, using standard operators (AND = * or & , OR = + or # , NOT= / or ') to implement the desired function. These equations do not need to be in sum-of-products form, and parentheses are used to indicate grouping and establish precedence. Intermediate equations are allowed for subsituting entire equations and for ease of design.

END SECTION

All ADF files end with the END$ statement.

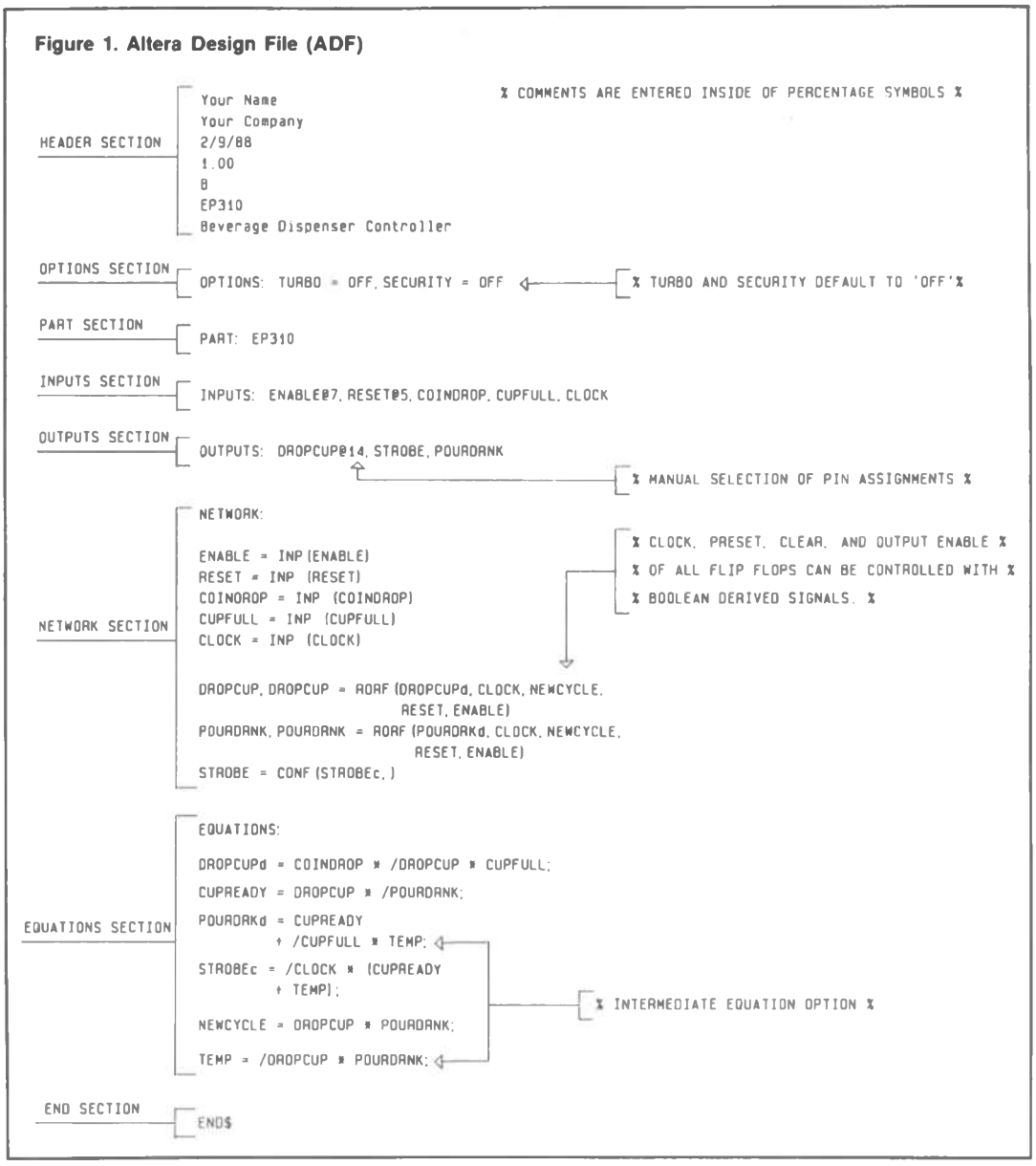
ADDITIONAL FEATURES

DUAL FEEDBACK

The Dual-Feedback capability of the EP1800/EP1810 Global MacroCells and the EPB1400 Generic MacroCells is accessed by declaring the buried logic in the OUTPUTS section, and the dedicated input in the INPUTS section. An example using a buried register and input pin with the same macrocell is shown below:

```
PART: EP1800

INPUTS: DUAL1@10

OUTPUTS: DUAL2@10

NETWORK: .
         .
         .
      DUAL1 = INP(DUAL1)
      DUAL2 = NORF(DUAL2d,CLK,CLR,GND)
         .
         .
         .
EQUATIONS:

      DUAL2d = A * B + C;
         .
         .
         .
END$
```

Figure 1. Altera Design File (ADF)



If Output Enable control is required, the syntax is as follows:

```
PART: EP1800

INPUTS: DUAL1

OUTPUTS: DUAL1

NETWORK: .
          .
          .
      DUAL1 = INP(DUAL1)
      DUAL1,DUAL1f = RORF(DUAL1d,CLK,
                          CLR,GND,OE)
          .
          .
          .
EQUATIONS:

     DUAL1d = A * B + C;
     OE     = D * E;
          .
          .
          .
END$
```

ACTIVE LOW INPUTS

Active low inputs are specified by inverting the signal in the NETWORK or EQUATIONS section. Examples are shown below:

(1) Active low input specified in NETWORK Section:

```
INPUTS: /X

% The "/" has no logical meaning in the %
% INPUTS section %

NETWORK:

Xn = INP(/X)
X = NOT(Xn)
% X is the internal acitve low signal %
```

(2) Active low input specified in EQUATIONS Section:

```
INPUTS: /Y

NETWORK:

Yn = INP(/Y)

EQUATIONS:

Y = /Yn;
% when pin is low Y is true %
```

ACTIVE LOW OUTPUTS

Active low outputs are specified by 'inverting' the left hand side of the equation. If active low outputs are required by sequential function, the feedback nodes must also be inverted. An example is shown below.

```
PART: EP610

INPUTS: CLOCK

OUTPUTS: /QC, /QB, /QA, /RCO
%The "/" has no logical meaning in the OUTPUTS
                                    Section%

NETWORK:
 CK      = INP(CLOCK)

 /QA,QAf = RORF(QAd,CK,,,)
 /QB,QBf = RORF(QBd,CK,,,)
 /QC,QCf = RORF(QCd,CK,,,)
 /RCO    = CONF(RCOc,)

EQUATIONS:

 QA = /QAf; % Invert feedback nodes %
 QB = /QBf;
 QC = /QCf;
 RCOc' = QC*QB*QA; %Invert left hand side
                                of equation%

 QCd' = QC*QA'
        QC*QB'
        QC'*QA*QB;

 QBd' = QB*QA'
        QB'*QA;

 QAd' = QA';

END$
```

FEATURES

- **High-Level design entry.**
- **Supports IF-THEN constructs and CASE statements.**
- **Optional Truth Table entry.**

INTRODUCTION

The Altera State Machine Input Language (ASMILE) provides a high-level design approach for state machine designs. The description language allows state machine diagrams to be created with simple Truth Tables and IF-THEN constructs or CASE statements. State machine design entry can be used as a stand-alone design entry mechanism for an EPLD, or it can be used in conjunction with LogiCaps schematic capture and TTL Macro-Functions for added power.

The ASMILE syntax is a superset of the Altera Design File (ADF) format. Users who are unfamiliar with the ADF format should first read the Boolean Equation section in this Handbook.

SMF SYNTAX

Using an ASCII text editor (in non-document mode), State Machine design files are created following the Altera State Machine File syntax (.SMF). The SMF file is broken down into sections, defined by a keyword followed by a colon (in all capitals and starting at the extreme left hand side of the page). Each section is described below. An example is shown in Figure 1.

The A+PLUS Design Processor converts this SMF design file into a Boolean equation representation. In the process, it chooses the optimal register (T or D) for each state variable. Then A+PLUS minimizes the resulting logic and creates a JEDEC file for device programming.

HEADER SECTION

The Header section includes all bookkeeping information such as design title, revision, designer, and any other desired information. This section is optional and has no bearing on the design itself.

OPTIONS SECTION

The Turbo-Bit and Security Bit are controlled here.

PART SECTION

This section specifies the targeted EPLD. "AUTO" can be used, which allows A+PLUS to automatically select the best EPLD for the design. If pin assignments are given, then the AUTO option cannot be used since the pins assigned must correspond to a particular EPLD.

INPUTS SECTION

All EPLD inputs for the design are specified here. Pin numbers can be assigned by appending an "@" to the end of the pin name.

OUTPUTS SECTION

The Outputs section declares all outputs used in the design. Output pin names may be assigned pin numbers in the same way as Input pins.

NETWORK SECTION

The Network section for a state machine file is only used to define machine outputs that are not also state variables. A+PLUS automatically adds the Network statements required for each input and each state variable. This section is optional for SMF files.

EQUATIONS SECTION

This section uses standard Boolean operators as in Boolean Equation entry. Equations can be used to define outputs which are not state variables, or to define intermediate equations. Intermediate equations might be used as a transition condition or as a clock for a state machine. This section is optional.

MACHINE SECTION

The Machine section specifies the state machine name. Several state machines can be placed into the same file.

CLOCK SECTION

The Clock section defines the clock for a given machine. The clock may come directly from an input pin (listed in the INPUTS section), or it can be defined as an any internal node within the design. Internal nodes are defined by the Equations section, the Truth Table section, or from other state machines.

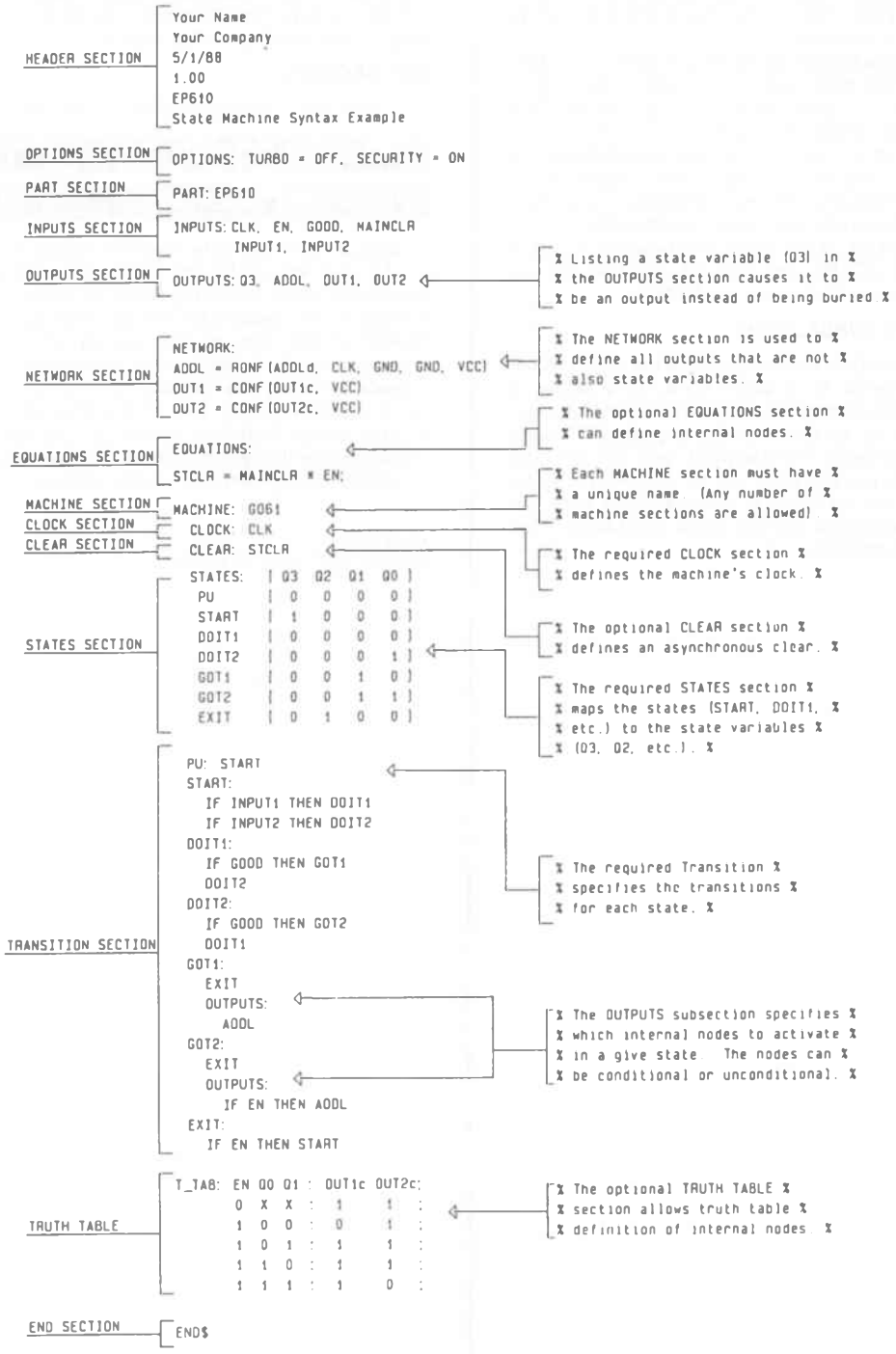
CLEAR SECTION

The Clear section defines a master asynchronous clear for the machine. It may come directly from an input pin or from any internal node within the design. This section is optional.

STATES SECTION

The States section defines all machine states. Each state is listed in a column down the page. The state variables of the machine are listed across the page and enclosed in square brackets. Each state must be assigned a unique definition of state variables.

Figure 1. State Machine File (SMF)



TRANSITION SECTION

This section defines the transitions for each machine state. This is the only section that does not have a keyword.

Each state should be listed followed by a colon. An IF-THEN construct is then used. The first IF statement has precedence over all others. This insures no ambiguity exists if 2 conditions are true. For unconditional branches, the IF statement should be deleted. If none of the conditions are true, and no unconditional transition is specified, the machine state will remain unchanged.

For example, if the machine shown in Figure 1 is in state START, and INPUT1 and INPUT2 are both 0, the next state will be START.

OUTPUTS SUBSECTION

The Transition section provides the ability to set internal nodes to a given state. This ability is accessed by listing the keyword OUTPUTS: after the transition specification for any state. All node names following the keyword, will be set high during the current state. An IF statement may be included so the node is a function of both the current state and current input conditions. This section is optional.

TRUTH TABLE SECTION

A final method of defining internal nodes is with a Truth Table format. Both sides of the table consists of internal nodes. See Figure 1.

END SECTION

All state machine files must terminate with END$.

DEFINING OUTPUTS FROM THE MACHINE

Outputs from the state machine may consist of any internal node within the design. Internal nodes include all of the following: State variables, nodes defined in the Equations section, inputs, nodes defined in the Outputs sub section, and, nodes defined in the Truth Table. State variables become outputs by listing the variable name in the OUTPUTS section at the top of the file. All other internal nodes must first go through the Network section where the type of output architecture must be specified (registered or combinatorial).

AB14A Rev 1.0

FEATURES

- **Allows logical simulation of EPLD designs from JEDEC file.**
- **On-screen Waveform Analyzer provides functional verification of a design in an interactive environment.**
- **Hard copy output of waveforms and tables for documentation.**
- **Flexible definition of input vectors using State Tables or Pattern format in hexadecimal, decimal, octal, or binary base.**
- **Powerful commands including Break, Force, Save, and accessing buried nodes provide unique debugging ability.**

INTRODUCTION

Functional simulation is the ability to verify that the design created operates correctly. This is done by applying vectors to the input pins of the device under test, then observing the outputs and comparing them to the expected results.

PLFSIM, the Altera Functional Simulator, provides an interactive tool to test EPLD designs. PLFSIM also provides a Virtual Logic Analyzer for observing waveforms within the design. By interfacing PLFSIM with the A+PLUS environment, the designer can enter, compile, and verify logical operation before programming a device. The simulator gets its information from the file used to program the device, therefore the designer is assured that the simulation accurately mirrors the functionality of the device.

PLFSIM uses three input files: a JEDEC file, a Vector file, and a Command file. The JEDEC file is the design file created by the A+PLUS processor and contains all the logical information about the design. The Vector file is created by the designer using a text editor, and contains logic values to apply to the input pins of the device. The Command file is a list of instructions which tell the Functional Simulator what to do and when to do it. The Command file is entered from the keyboard in an interactive environment or may be pre-created with a text editor and submitted to the simulator for batch operation.

The Waveform Analyzer is an on-screen logic analyzer, allowing interactive examination of up to 256 signals for 512 timesteps. Single key-stroke commands provide powerful debugging tools such as split screens, multiple zoom levels, and bussing of signals which help to quickly analyze a logic design.

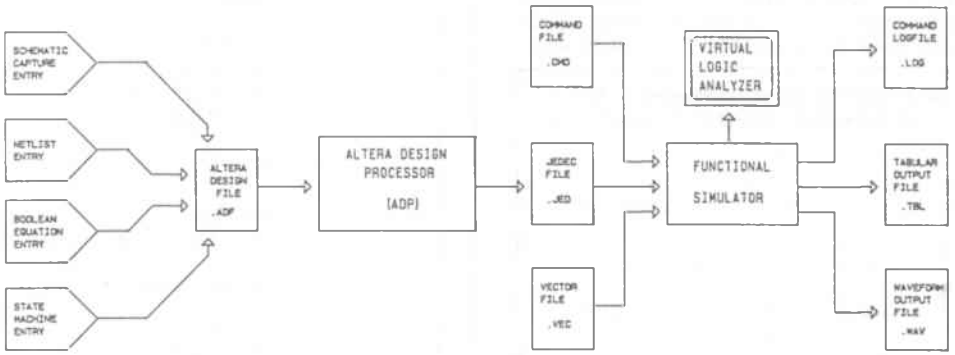
PLFSIM also provides both tabular and waveform output files wich can be stored on disk or printed out.

GETTING STARTED

There are two steps that must be done before beginning to simulate the design, generating the JEDEC file of the design, and developing input vectors to exercise the design.

Actual execution of the simulation is done by entering commands in the simulation environment. PLFSIM provides a set of commands that tells the simulator how long to simulate, where to find the input vectors, and what nodes to observe. In addition, there is a complete set of commands designed to ease the debug process.

Figure 1. Functional Simulator Block Diagram — *Functional diagram of the design process, from design entry to functional simulation of a processed design.*



After executing the simulation, the VIEW command invokes the Virtual Logic Analyzer, which allows on-screen examination of the simulation results.

STEP 1: Creating the JEDEC file

As shown in Figure 1, any one of four design input techniques can be used to enter a design into the Altera Design Processor. Once entered, the Processor automatically generates JEDEC object code which is used to program an EPLD. The Functional Simulator uses this same JEDEC file to obtain information about the design. Therefore, the design being simulated is the actual design programmed into the device.

STEP 2: Creating the input vectors

The next step in the simulation process is to define the input values to apply to the EPLD input pins during simulation. For example, suppose a design is known to behave a certain way when inputs A, B, and C are given the waveform shown in Figure 2. To verify this design, use this waveform as the input vectors, and then observe the simulation to verify that the design is working correctly.

This waveform needs to be translated to input vectors in a format that the simulator can understand. There are two formats that can be used, the Tabular format or the Pattern format.

The Tabular format, as indicated by the keyword 'TABLE', is the most straightforward. The input node names are listed across the top of the table with the desired values of each input listed in the columns below. Each row corresponds to a single timestep in the simulation process. Simulating the 10 timesteps shown in Figure 2 would require 10 rows of values. Figure 3a shows how to convert the waveform in Figure 2 into a table formatted Vector file. Note that the Tabular format cannot be used interactively in the simulation environment, and must be created with a text editor and saved as a file with the extension VEC.

For long simulations the Tabular format can be quite cumbersome. In these cases the Pattern format makes vector entry more convenient by using the PATTERN command. In this format input nodes are equated to a string of desired values with each value corresponding to a single timestep. For example, the waveform for input A would be entered as "A=0 1 0 1 0 1 0 1 0 1;", as shown in Figure 3b.

Instead of entering the input value for each and every timestep, the asterisk "*" can be used to indicate repetition of any pattern enclosed in parenthesis for any length of time. For example, the input signal A shown in Figure 2 could be designated by the pattern "A=(0 1)*5;". This means that the pattern within the parenthesis (in this case 0 1) will be repeated 5 times. An asterisk without a number immediately following it means that the pattern inside the parenthesis will be repeated continuously.

Nested repetition is also allowed, so that the C input in Figure 2 could be entered as:

"C=((0)*4 (1)*4)*;"

which means that the C input will repeat the string

Figure 2. Desired Input Waveform — *This is the input waveform that is to be applied to the design.*

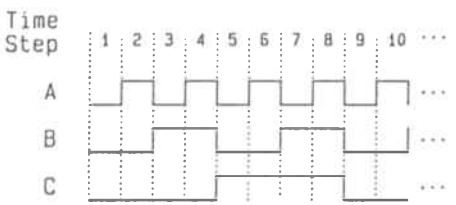


Figure 3. Input Vector Definition — *Shown are 5 different ways of defining the waveform in Fig. 2.*

3a) Tabular definition.

```
TABLE:  A  B  C ;
        0  0  0
        1  0  0
        0  1  0
        1  1  0
        0  0  1
        1  0  1
        0  1  1
        1  1  1
        0  0  0
        1  0  0
```

3b) Simple pattern format.

```
PATTERN:
A = 0 1 0 1 0 1 0 1 0 1;
B = 0 0 1 1 0 0 1 1 0 0;
C = 0 0 0 0 1 1 1 1 0 0;
```

3c) Pattern format with nested repetition.

```
PATTERN:
A = (0 1)*;
B = (0 0 1 1)*;
C = ((0)*4 (1)*4)*;
```

3d) Group vector input.

```
PATTERN:
IN = (0 1 2 3 4 5 6 7)*;
```

3e) Predefined input waveforms.

```
PATTERN:
IN = COUNT;
```

of four 0's followed by four 1's continuously. Figure 3c shows how to enter the desired waveform using nested patterns.

To further ease input definition, a set of related inputs can be defined as a group and listed in the Vector file using hexadecimal, decimal, octal, or binary numbers. For example, if the three inputs A, B, and C are defined as a decimal group named IN, with C being the most significant bit, then the single line "IN = (0 1 2 3 4 5 6 7)*" would be equal to the desired input waveform. Group definitions occur in the Command file. This is useful when a set of inputs represents an address or data bus.

Finally, pre-defined input waveforms exist to make input definitions even easier. The binary counting sequence shonw in Figure 2 could be entered with the line "IN = COUNT". In this case COUNT indicates that the group should simply count from 0 to 7 continuously. Other pre-defined waveforms include: GREY, a grey code counter; ROTATE, a rotating bit sequence; and, GLITCH, a glitch creation sequence where the maximum number of bits change each time step.

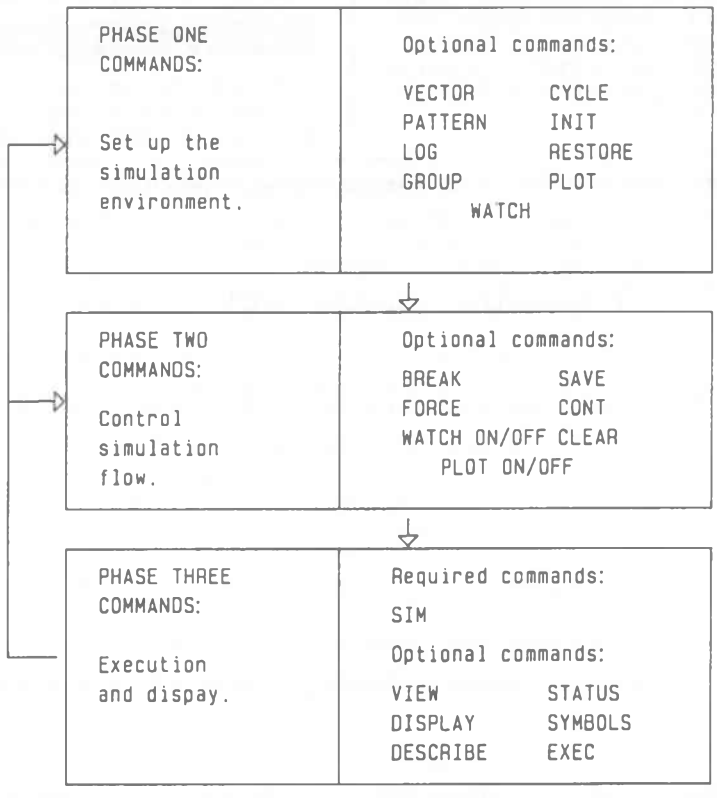
STEP 3: Executing the simulation

After generating a JEDEC file describing the design and defining the simulation input values, the next step is to execute the simulation. Execution involves entering commands that tell the simulator how long to simulate, which vector file to use (or define the input vectors), and what signals to observe.

PLFSIM can run in one of two modes: interactive or batch mode. In the interactive mode, commands are entered within the PLFSIM program itself. In the batch mode, a pre-created list of commands are used to control the simulator so that the entire simulation is done automatically.

PLFSIM features a selection of powerful commands that can be divided into 3 different phases as shown in Figure 4. The first phase of commands sets up the simulation environment by initializing values and defining output formats. The second phase of commands controls the simulation flow based on various simulation conditions. The third phase executes the simulation and displays the results with the Virtual Logic Analyzer.

Figure 4. Functional Simulator Command Groups — *These are selected basic commands used to set up the simulation, control the simulation, and execute the simulation.*



Of course, the HELP command is always available to provide information on the entire set of commands.

PHASE 1: Setup Commands

Phase 1 commands are used to initialize the simulation environment. They are typically the first commmands entered to the simulator and they tell it what the input vectors are or where to find them, how to initialize the nodes, and how to format the output.

The LOG command allows logging all the commands entered interactively and saves them in a text file. This running log of commands can be used later as a command file for using the batch mode option of the simulator.

The VECTOR command tells the simulator which file is to be used to generate the input vectors. It is possible to switch from one Vector file to another during the simulation so that more than one set of input conditions can be used to simulate a given design.

While the VECTOR command can be used to change the input vectors for the entire set of inputs, the PATTERN command can be used to alter the input vector of a single node. The PATTERN command overrides the Vector file and thus can be used to modify input vectors interactively in the middle of a simulation run. For example, input vectors for the waveform shown in Figure 2 can be defined interactively as "PATTERN IN=(0 1 2 3 4 5 6 7)*".

In applications where a set of inputs or nodes is best described as a single value, the GROUP command allows refering to this set of signals using hexadecimal, decimal, octal, or binary format. This command means hex can be of binary values to refer to input or output busses.

The waveform output is formatted with the PLOT command. Waveforms for all the listed input or output nodes will appear in the plot output file (denoted as a .WAV file) in the order specified. This file is used by the Virtual Logic Analyzer for on-screen viewing, as shown in Figure 5, or this file may printed with a dot-matrix printer for permanent documentation.

Similarly, the WATCH command formats a tabular output file. This file allows the listing of groups with their hex, decimal, octal, or binary values. This file may also be printed out, as shown in Figure 6.

The CYCLE command defines the number of vector clocks in a single cycle. This cycle number, as opposed to the vector clock, is used in all references to time found in the commands.

Finally, the INIT command will initialize input values of the simulation before applying any inputs. This should only be used prior to simulation. Initialized values will be overidden by the input vectors during the first vector clock.

PHASE 2: Control Commands

Phase 2 commands are used to control the flow of execution and to change conditions during simulation. All these commands are optional.

Figure 5. Sample screen from Virtual Logic Analyzer showing waveform and bus output.

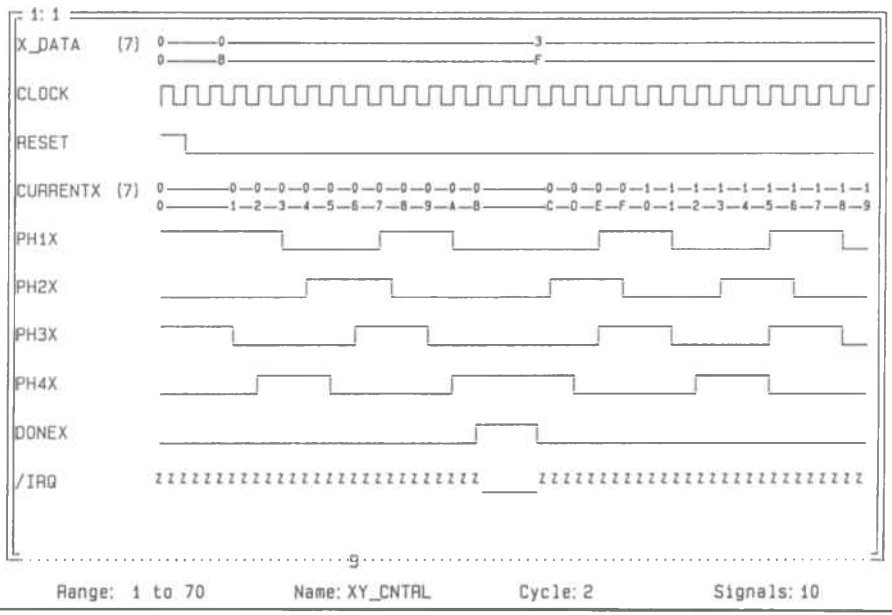


Figure 6. An example of a Tabular output file.

```
          C
          Y
          C
          L I Y Y Y Y Y Y Y Y
          E N 0 1 2 3 4 5 6 7

          1 0 0 1 1 1 1 1 1 1
          2 1 1 0 1 1 1 1 1 1
          3 2 1 1 0 1 1 1 1 1
          4 3 1 1 1 0 1 1 1 1
          5 4 1 1 1 1 0 1 1 1
          6 5 1 1 1 1 1 0 1 1
          7 6 1 1 1 1 1 1 0 1
          8 7 1 1 1 1 1 1 1 0
          9 0 0 1 1 1 1 1 1 1
         10 1 1 0 1 1 1 1 1 1
```

During a simulation run, it may be desired to verify certain logic conditions or check for faulty ones, to monitor the state of nodes during a certain time period, or to make changes to input vectors depending on the logic levels or cycle values reached during a simulation. The BREAK command allows checking for these and other simulation conditions by causing the simulation to stop when a break condition is met. The break condition can be a function of cycle number, node values, or a combination of both.

To make a break dependent on the cycle value, the "RANGE" sub-command is used. For example, "BREAK RANGE 15" will cause the simulator to break at cycle 15.

Similarly, the sub-command "NODES" is used to define a break at certain node values. For example, "BREAK NODES A=0 B=0 C=1" will cause the simulator to break when the listed nodes have the specified value.

Conditions based on both node values and cycle value can be created by including both the RANGE and NODES sub-commands in the command. Thus, "BREAK RANGE 5 to 15 NODES A=0 B=0 C=1" will cause a break to occur if the node conditions are met between cycle 5 and 15.

Not only may the points be set to interrupt the simulation, but commands can be defined to determine what should happen when the breakpoint is met. A command sub-list can be specified after the word "DO" for each break point which will be executed when the break conditions are met. For example, the command "BREAK RANGE 20 DO VECTOR @NEW.VEC" will stop the simulation at Cycle 20 and change the active vector file.

Within this command sub-list a certain node may be forced to a specified level with the FORCE command. This command is similar to the INIT command, but while INIT should be used only at cycle 0, FORCE may be used at anytime.

To control the range of simulation that is written to the output files, use the PLOT ON and PLOT OFF commands. This prevents unwanted time cycles from cluttering the output file. Similarly, WATCH ON and WATCH OFF controls the tabular output file.

Finally, the CONT command tells the simulator to continue the simulation process after a breakpoint is met and the CLEAR command will remove a breakpoint so that it won't interrupt again.

As an example of the power of the BREAK command, consider the following command.

```
BREAK RANGE 5 TO 12
      NODES OUT = 4 ENABLE = 0
      DO
         FORCE CLEAR = 1;
         CONT;;
```

The command causes a break to occur if the group OUT = 4 and the node ENABLE = 0 between Cycle 5 and Cycle 12. If this break condition is met, the sub-command list following DO is executed. In this case, the tabular output file is turned off, the CLEAR input is forced to 1 for one clock cycle, and the simulation continues.

PHASE 3: Execution Commands

Phase 3 commands are used to execute the simulation and display the results to the screen.

After the simulation environment is set up by entering the Phase 1 and Phase 2 commands, the SIMULATE command executes simulation up to the indicated number of cycles. For example, "SIMULATE 30" does simulation up to cycle 30. "SIMULATE +30" simulates for 30 cycles from the present cycle.

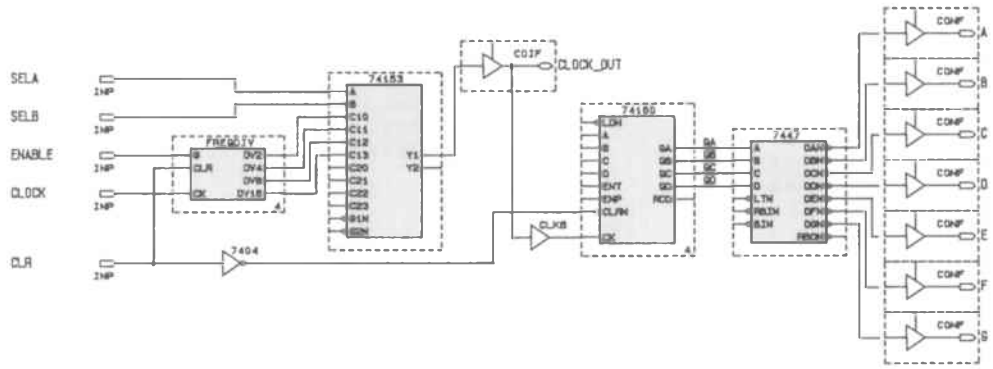
During simulation the input values are computed from the Vector file, PATTERN commands, and any active FORCE commands. The signals are constantly scanned to see if any BREAK conditions are met, and the output values are written to the output files as indicated by the PLOT and WATCH commands. After execution, the Functional Simulator prompt will be updated to show the current cycle value and further commands can be entered.

The VIEW command invokes the Virtual Logic Analyzer. Figure 5 shows how this allows the results of the PLOT command to be examined on-screen. This is a powerful environment, allowing split screens, control 2 cursors and multiple zoom levels. There is also a complete set of single stroke commands that allow, for example, grouping of signals into busses, and searching for a given simulation state.

The SYMBOL, STATUS, and DESCRIBE commands display useful information about nodes or the simulation environment. The DISPLAY command will list any file to the screen for on-line examination.

The BEGIN command returns to the beginning of the whole process. The Phase 1 and Phase 2 command can be re-entered to modify the simulation environment for further execution.

Figure 7. Schematic of a Variable Frequency Counter with Seven Segment Driver — *An example design entered using LogiCaps schematic capture package. This design was then processed with the Altera Design Processor to produce a JEDEC file used for simulation and programming.*



The QUIT command will leave the simulation environment and return to A+PLUS.

EXAMPLE—Counter-decoder

For an example of the simulation process, consider the example in Figure 7. This design is a selectable frequency counter which drives a seven-segment decoder at the output. This is the actual schematic created in the Logicaps Schematic Capture program and then processed into a JEDEC file by the Altera Design Processor. The file produced was called EXAMPLE.JED.

The next step is to develop input vectors to place on the input pins of the device, in order to verify the logic is correct. A good way to test this design is to first select the counter clock using SELA and SELB, clear the device, then enable continuous counting. After determining the waveform, a Vector file is created using a text editor. This file is shown in Figure 8. Notice that the clock is a continuous stream of two 0's followed by two 1's. The CLEAR signal is asserted for two timesteps, and then after 5 timesteps the ENABLE signal is asserted, allowing the device to operate.

Now it must be determined which nodes to monitor. For example, suppose it is desired to observe the output of the counter in a decimal format, and also observe the outputs in both a tabular and waveform format. This is done by first grouping the outputs of the counter using the command "GROUP DEC OUT=QD QC QB QA;". Then, the PLOT and WATCH commands are used to select the nodes to be observed and their order of presentation.

The simulator must be told where to look for the input vectors, and this is accomplished by the VECTOR command, specifying which vectors are wanted.

The CYCLE command defines a cycle as four vector clocks. A vector clock corresponds to a

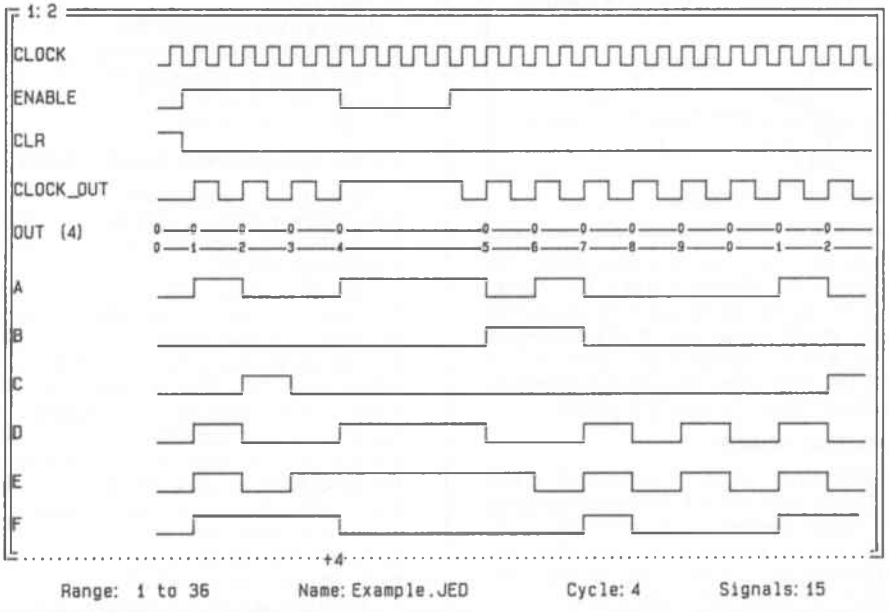
Figure 8. The Input Vector File Used to Simulate the Example in Figure 7 — *Pattern definition is used in this example, and the "*" indicates repetition.*

```
PATTERN:

 CLOCK  = (0011)*;
 CLR    = (1)*2  (0)*;
 ENABLE = (0)*5  (1)*;
 SELA   = (0)*;
 SELB   = (0)*;
```

Figure 9. The Command File Used to Simulate the Example in Figure 7 — *This command file was used to simulate the design in Batch mode.*

```
GROUP DEC OUT = QD QC QB QA ;
VECTOR @Example.VEC ;
PLOT   CLOCK ENABLE CLR CLOCK_OUT
       QA QB QC QD A B C D E F G ;
WATCH  CLOCK OUT A B C D E F G ;
CYCLE 4 ;

BREAK RANGE 5 TO 12
      NODES OUT = 4
      DO
        WATCH = OFF ;
        FORCE ENABLE = 0 ;
        CONT ;;

 SIM 36 ;
 VIEW ;
 QUIT ;
```

Figure 10. Virtual Logic Analyzer Display of the Simulation of Example Design — *The Virtual Logic Analyzer can be used to interactively debug designs.*



single clock period in the vector file.

The BREAK command is an example of a Phase 2 command. In this case it is used to stop if the condition OUT=4 between cycle 5 and 12 is met.

If this condition is met, the listed sub-command is executed. The WATCH OFF command turns the tabular file off so that subsequent node values are not written to that output file. This prevents unneeded simulation data from cluttering the output table. (Values are still written to the waveform output file as specified in the PLOT command). The FORCE command is then used to disable the counter by driving ENABLE low. Finally, the CONT command causes the simulation to resume execution after servicing the BREAK command.

The third group of commands are Phase 3 commands. The "SIM 36" command executes simulation until cycle number 36 is met.

The VIEW command causes the simulator to enter into the Virtual Logic Analyzer environment. A screen from this environment is shown in Figure 10. While in this environment it is possible to examine the waveform using split screens, multiple zooms and data busses to assist in debugging.

Finally the QUIT command returns to the A+PLUS environment.

ADVANCED FEATURES

The Functional Simulator has several advanced features which are described below. The features are all accessed with special commands or variations of the standard commands.

SAVING AND RESTORING

PLFSIM has the ability to save a current simulation state and restore that state as a initial condition in a future simulation run. The command "SAVE @filename.SAV" will cause the current simulation state to be saved in the file named "filename.SAV. The information saved includes cycle value, node values, break points, and the state of the Virtual Logic Analyzer. The RESTORE command "RESTORE @filename.SAV" will restore the saved information.

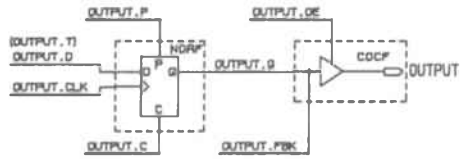
EXECUTING HIERARCHICAL COMMANDS

The EXECUTE command can be used to run any command file from any other file. This allows one command file to transfer execution control to another command file. The syntax for that command is "EXECUTE @filename.CMD". Encountering a QUIT in the second command file will act like a return from subroutine call and return execution control to the first command file.

ACCESSING BURIED REGISTERS

In addition to providing easy access to all output and input pins, the functional simulator provides access to all buried output primitives such as "NORF"'s and "NOCF"'s, which do not actually drive a pin. In the example from Figure 7, the outputs of the 74162 were named QA through QD.

Figure 11. Legal Extensions for Referencing Subnodes — *Extensions allow the observation of all inputs to I/O primitives during simulation.*



These outputs really correspond to buried NORF's which are inside the MacroFunction. The nodes QA through QD can be called up during simulation just as easily as an output pin. In the command file from Figure 9, the "GROUP DECIMAL OUT = QD QB QC QA" command was used to combine all these buried nodes into a single group.

ACCESSING SUB-NODES

PLFSIM also allows access to sub-nodes of I/O Primitives. Figure 11 shows the nodes surrounding an output primitive and their extensions. The nodes are accessed by appending a dot and then the extension to the output pin name. So, for example, "WATCH OUTPUT OUTPUT.D OUTPUT.Q OUTPUT.CLK OUTPUT.OE" would display in the tabular output file the pin output, D-input, Q-output of the flip flop, the flip flop clock input, and the output enable of the tri-state buffer.

When a design passes through A+PLUS, all JK and SR flipflops are emulated by D and T flipflops. Therefore, JK or SR inputs which cannot be referenced directly and J, K, S, and R subnode extensions are not allowed.

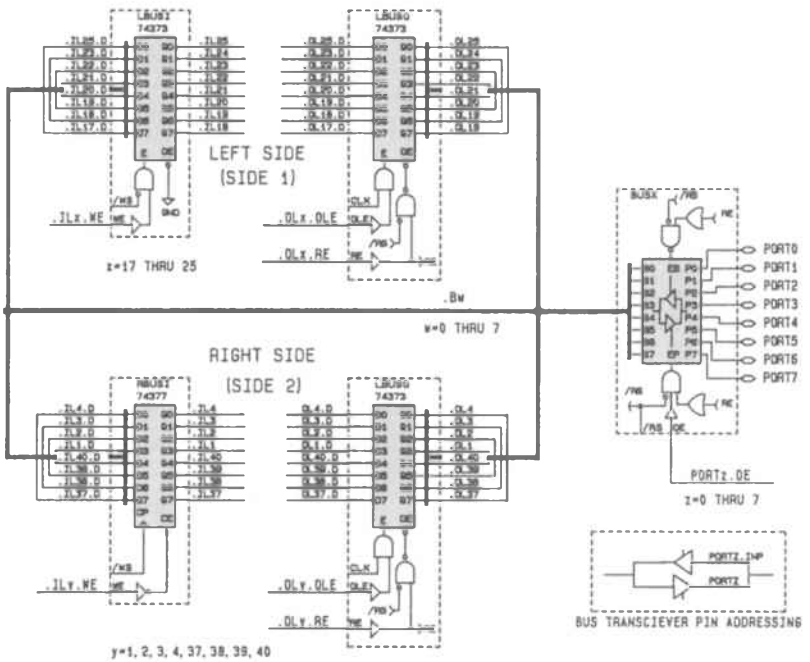
USING PRE-DEFINED NODE NAMES

Besides pin names defined by the user, the functional simulator recognizes pre-defined names for each pin, macrocell, and special structures within each EPLD.

Each pin can be referred to by its pin number. This name is is ".P" and the pin number. For example, an output named "OUT" at pin number 17 can be called out in the functional simulator as "OUT" or ".P17". This is especially useful if the report file indicates that it is necessary to tie two clock input pins together on the device itself. In this case it is necessary to "tie" these pins together for simulation as well. Since only one pin has been named in the input file the second clock pin must be accessed with its pre-defined name. Both of these values must appear in the vector file and must have the same vectors.

Nodes can also be accessed by using the macrocell number. This number is read from the report

Figure 12. Pre-Defined Node Names in the EPB1400 — *The pre-defined names in the EPB1400 allow thorough observation of all nodes within the microprocessor interface section.*



file generated by A+PLUS. Therefore, if "OUT" is at pin 17, macrocell 6, it can be refered to as "OUT", ".P17", or ".M6".

SIMULATING THE EPB1400

The EPB1400 is the first in Altera's line of Bus I/O-Register Intensive Architecture (BUSTER) devices. Besides a block of general purpose macrocells, it also has a microprocessor interface section, consisting of an 8-bit bus transceiver, 2 input register/latches, 2 output latches, and an internal byte wide bus connecting them together. These elements have predefined node names to aid in the simulation of this device. The architecture of this device is symmetrical (see Figure 12), with each side having an output register and an input register.

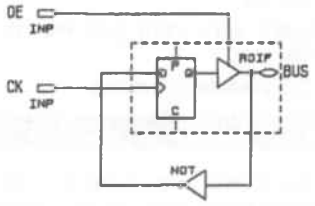
The input registers are referenced using the extension ".IL" and a number. The number used correlates to a pin number for the side of the device on which the register is located. In this way each bit of a byte wide register can be accessed. The input registers also have predefined subnodes. The inputs to the registers use the extension ".D", and the write enable (WE) signal can be observed using the ".WE" extension. For example, to see the input, enable, and output of the third bit of the input register on the left side of the EPB1400, the command would be "WATCH .IL23.D .IL23.WE .IL23". Note that each bit of an input register uses the same write enable signal, so that observing the WE for one bit, that same WE signal is also being applied to the other seven bits.

The output latches use the same numbering scheme, but are denoted by ".OL". Output latches have 2 subnode extensions, ".OLE" for the output latch enable signal, and ".RE" for the read enable. Therefore, if the input, latch enable, read enable, and output of the 3rd bit on the right side of the chip wanted to be observed, the command would be "WATCH .OL2.D .OL2.OLE .OL2.RE .OL2". Again note that the .OLE and .RE extensions are global per output latch.

The internal bus of the EPB1400 also uses predefined node names. These lines can be individually accessed with the name ".B0, .B1, ... , .B7". The .B0 line is connected to the Q0 outputs of the output latches, the D0 input of the input registers, and the B0 I/O of the BUSX transceiver. .B1 is connected to Q1, D1, and B1, and so on. So, if it were desired to observe the internal bus in a hexadecimal bus format, the command would be "GROUP HEX IBUS= .B0 .B1 .B2 .B3 .B4 .B5 .B6 .B7".

The BUSX Macro is a bi-directional bus transceiver similar to like the 74245. It can be observed in the functional simulator by using its pin names. In Figure 12, the pin names are PORT0 through PORT7. If different names are used, these user defined names must be used in the simulator. The output enable signal OE, which is common to all 8 bits in the transceiver, can be accessed using the subnode extension ".OE". Also, any input vectors applied to the port can be observed using the extension ".INP". This brings up the point of simulating bi-directional pins.

Figure 13. Sample Schematic and Vectors for Bi-directional Pin Simulation — *The Functional Simulator has the capability to simulate bi-directional pins. Notice that the input vectors to the 'BUS' pin need to be 'Z' whenever OE = 1.*




```
PATTERN:
CK = 0011 0000 0011 0000
OE = 0000 1111 0000 1111
BUS= 1111 ZZZZ Z000 ZZZZ
```

SIMULATING BI-DIRECTIONAL PINS

Bi-directional pins can be handled with proper use of the 'Z' input value. If an input vector is being applied to a bi-directional pin, a 'Z' must be applied whenever the output enable of the pin is active, which means the pin is being driven by the EPLD. When the output enable of the pin is disabled, the desired value can be placed into the device by declaring it in the vector file. Figure 13 shows how this can be done on a simple I/O macrocell.

The ".INP" extension is used to observe the value being applied into a bi-directional pin. This way the input vector being applied to the device pins can be observed, because the pin name alone gives the pin output. For example, observing the BUS pin in Figure 13, only the Q value would be seen when OE is asserted, and 'Z' when OE is negated. But since an input vector is being applied to the pin, the actual value is the one denoted by the input vector. This also holds true for the pins of the BUSX transceiver in the EPB1400.

AB24 Rev 4.0

FEATURES

- **Generic testing.**
- **Automatically appending test vectors to the JEDEC file.**
- **AC and DC parametric testing.**

INTRODUCTION

Post-program testing is functional testing of a PLD after program and verify cycles are completed. It was introduced as a means of reducing the failure rates of fuse-based PLDs. The high failure rate of those devices is directly attributable to a fundamental problem with the fuse technology: there is no way to test a fuse. Unable to complete IC testing themselves, fused-based PLD manufacturers are forced to pass the testing burden onto the user.

Altera's EPLDs, on the other hand, use EPROM cells in place of fuses to implement logic. The erasable nature of these cells allows Altera to fully test devices within the normal production flow BEFORE shipping parts to the customer. This type of pre-program testing is called "Generic Testing" and is the primary reason Altera can guarantee 100% programming yield of its components. If additional testing is desired, this Application Brief shows how to perform post-program testing of EPLDs.

GENERIC TESTING

Altera's generic testing manufacturing flow programs and tests each and every EPROM cell in every device. At the same time, Altera tests functionality of the entire device, including every architectural configuration of each MacroCell. Finally, Altera completes full A.C. testing based on the performance of the worst case test pattern as well as full D.C. parametric testing of every pin. Contact Altera marketing for copies of Application Brief 45, "Testing of Plastic OTP EPLD's", and the Altera Quality and Reliability Handbook, which describe the manufacturing test flow of EPLDs.

After programming the EPLD, a simple verify test assures that no faults were introduced due to poor device contact or out-of-spec programming equipment. Altera guarantees that 100% of the EPLDs that pass the verify test will function as programmed when using Altera programming hardware.

POST-PROGRAM TESTING

When desired, the customer can perform post-program testing by using test vectors generated from the output of Altera's Functional Simulator and a Data I/O programmer.

In order to perform post-program testing, simulation results must be made available to form the expected data. Altera's Functional Simulator, PLFSIM, provides a useful means of generating this information. For a complete description of how to use PLFSIM, refer to "Altera Functional Simulator — PLFSIM" in the development system section of this book.

Simulation input vectors for PLFSIM are defined with the PATTERN command or placed in a separate vector file. To simplify input vector definition, PLFSIM provides pre-defined input waveforms. To use these waveforms, first use the GROUP command to define all the inputs to be in a single group. Next use the PATTERN command to define this group to equal one of the pre-defined waveforms, ROTATE, COUNT, GREY, or GLITCH.

If 100% fault coverage is desired, the inputs should be carefully constructed to insure that 100% of the internal nodes are toggled. To assist with intelligent vector definition, PLFSIM automatically calculates the percentage of nodes toggled after each simulation.

Figure 1. Sample Command (CMD) File— *This command file for PLFSIM defines the input vectors, lists the values to be watched, and simulates for 20 cycles.*

```
% Define the inputs %

PATTERN CLOCK = (01)*;
PATTERN ENABLE = 1100 (1)*;
PATTERN CLR = 0001 (0)*;
GROUP HEX INPUTS = SELA SELB ;
PATTERN INPUTS = COUNT ;

% List the inputs and outputs
  you want to watch. %

WATCH SELA SELB CLOCK ENABLE CLR
  A B C D E F G CLOCKOUT;

% Simulate for 20 time steps %

SIMULATE +20;
QUIT;
```

One of the output formats of PLFSIM is a tabular ASCII file that lists all input or output signals specified by the user. The WATCH command specifies the order and radix of signals to be listed. Figure 1 shows a sample command file for PLFSIM and Figure 2 shows the resulting tabular output file.

Figure 2. FSIM Tabular Output—*The tabular output file from PLFSIM for the command file given in Figure 1.*

```
JEDEC file : macexam.JED
EPLD part  : EP600




                                         C
                                         L
            E                            O
C       C   N                            C
Y  S  S L   A                            K
C  E  E O   B  C                         O
L  L  L C   L  L                         U
E  A  B K   E  R  A  B  C  D  E  F  G    T

 1 0  0 0   1  0  0  0  0  0  0  0  1    0
 2 0  1 1   1  0  0  0  0  0  0  0  1    0
 3 1  0 0   0  0  0  0  0  0  0  0  1    0
 4 1  1 1   0  1  0  0  0  0  0  0  1    0
 5 0  0 0   1  0  0  0  0  0  0  0  1    0
 6 0  1 1   1  0  0  0  0  0  0  0  1    0
 7 1  0 0   1  0  0  0  0  0  0  0  1    0
 8 1  1 1   1  0  0  0  0  0  0  0  1    0
 9 0  0 0   1  0  0  0  0  0  0  0  1    0
10 0  1 1   1  0  0  0  0  0  0  0  1    0
11 1  0 0   1  0  1  0  0  1  1  1  1    1
12 1  1 1   1  0  1  0  0  1  1  1  1    0
13 0  0 0   1  0  1  0  0  1  1  1  1    0
14 0  1 1   1  0  0  0  1  0  0  1  0    1
15 1  0 0   1  0  0  0  1  0  0  1  0    0
16 1  1 1   1  0  0  0  1  0  0  1  0    0
17 0  0 0   1  0  0  0  1  0  0  1  0    0
18 0  1 1   1  0  0  0  0  0  1  1  0    1
19 1  0 0   1  0  0  0  0  0  1  1  0    1
20 1  1 1   1  0  1  0  0  1  1  0  0    1

Simulation cover : 85%
```

All standard I.C. testers accept some kind of tabular input to describe input vectors and expected output vectors. Although the tabular input syntax varies from tester to tester, conversion from the PLFSIM output is straightforward. It may be completed by hand with a text editor, or with a format conversion utility program.

Conversion to the JEDEC format accepted by Data I/O PLD programmers and conversion to a format used by an Integrated Measurement Systems (IMS) Logic Master tester will be examined. The principles covered may be used for other standard testers.

JEDEC FILE TEST VECTORS

For those customers who wish to perform post-program functional testing on a Data I/O PLD programmer, it is possible to append PLFSIM simulation results to the end of the JEDEC file to serve as test vectors. Altera has written a utility program called AVEC that appends test vectors to the JEDEC file automatically. AVEC is available from Altera's Electronic Design Support Service (refer to the Utility Software Programs section of this book).

The JEDEC standard requires that the value for each pin be listed in pin number order. Output pins must have the expected data specified with an "H" or "L" instead of the standard "0" and "1" listed by PLFSIM. Also, a column for the VCC and GND pins, which are not listed by PLFSIM, must be filled with the value "N". Finally, any pins that are listed as GND in the report file (.RPT), must have the "X" listed by PLFSIM changed to a "0". AVEC makes all these changes automatically.

Figure 3 shows the JEDEC test vectors created by AVEC from the simulation results of Figure 2.

Figure 3. JEDEC File with Test Vectors—*Altera's AVEC program automatically appends test vectors to the end of a JEDEC file.*

```
N 20 test vectors *
V1      00XLHLLXLL0NX1LXXXXXXXX0N*
V2      11XLHLLXLL0NX1LXXXXXXXX0N*
V3      00XLHLLXLL1NX0LXXXXXXXX0N*
V4      11XLHLLXLL1NX0LXXXXXXXX1N*
V5      00XLHLLXLL0NX1LXXXXXXXX0N*
V6      11XLHLLXLL0NX1LXXXXXXXX0N*
V7      00XLHLLXLL1NX1LXXXXXXXX0N*
V8      11XLHLLXLL1NX1LXXXXXXXX0N*
V9      00XLHLLXLL0NX1LXXXXXXXX0N*
V10     11XLHLLXLL0NX1LXXXXXXXX0N*
V11     00XHHHHXHL1NX1LXXXXXXXX0N*
V12     11XLHHHXHL1NX1LXXXXXXXX0N*
V13     00XLHHHXHL0NX1LXXXXXXXX0N*
V14     11XHLHLXLH0NX1LXXXXXXXX0N*
V15     00XLLHLXLH1NX1LXXXXXXXX0N*
V16     11XLLHLXLH1NX1LXXXXXXXX0N*
V17     00XLLHLXLH0NX1LXXXXXXXX0N*
V18     11XHLHHXLL0NX1LXXXXXXXX0N*
V19     00XHLHHXLL1NX1LXXXXXXXX0N*
V20     11XHLLHXHL1NX1LXXXXXXXX0N*
```

2

IMS LOGIC MASTER TESTER

If more rigorous testing than is possible with a PLD programmer is desired, a standard ASIC tester can be used. The Logic Master Series of ASIC testers from Integrated Measurement Systems, Inc. of Beaverton, Oregon, is one such system. It provides the ability to do complete functional testing of any device with up to 68 pins. In addition, it can do D.C. parametric testing and A.C. testing up to 40 MHz.

To perform post-program testing with the IMS system requires three steps. The first step is to create a test board. Typically this consists of a ZIF (Zero Insertion Force) socket wire-wraped to a PC board. The test board can be customized to fit the user's particular needs. For example, the ouput load can be designed to replicate the actual load expected on the final production board.

In our example, a separate load board was configured for each of the EPLD packages. All the dedicated input pins were wired directly to a force channel (an output channel from the IMS) and all the I/O pins were wired both to a force channel and an aquisition channel (an input channel of the IMS). The capacitive loads for each output were identical to that specified as the test conditions in the Altera Databook.

The second step is to configure the tester to the proper pinout, D.C. parameters, and timing waveforms for the given EPLD. With the Logic Master, configuration is accomplished on a personal computer (P.C.) or terminal that is connected to the tester through a series of pop-up menus. The resulting configuration file can be saved as a text file for future use.

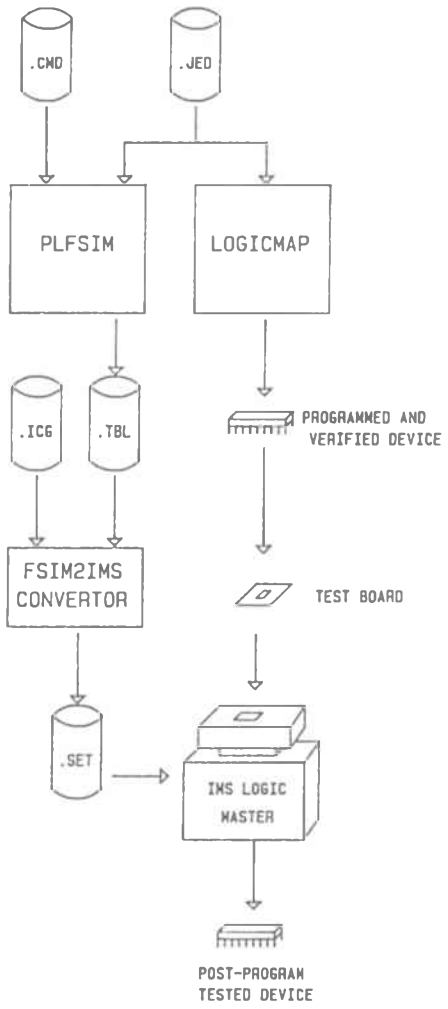
In our example, a separate configuration file for each EPLD was hand created and saved with the extension .ICG. The .ICG file contains all the information about the particular device type under test. Once the configuration files were completed, moving from one device to another simply involved switching the test board and loading the desired configuration file.

Finally, actual test vectors must be loaded into the tester. The IMS Logic Master is able to read tabular files from the P.C.'s disk.

Altera Applications has written a utility program to automate the conversion process for the IMS Logic Master as shown in Figure 4. The utility, called FSIM2IMS, combines any valid PLFSIM .TBL output with a valid .ICG file and creates a complete setup file for the IMS. The final setup file, which has the extension .SET, has all the information needed to test a given pattern in a given device. This utility is available to any IMS user through Altera Electronic Design Support Service.

Once the .SET file with the configuration and simulation information is loaded into the IMS, complete testing of the device can begin. A testing procedure may include functional testing, A.C. testing, D.C. parametric testing, or all of the above.

Figure 4. IMS Conversion Routine—*A conversion routine automates test vector generation for the IMS Logic Master. Customers wishing to automate test vector generation on other ASIC test hardware should write interface software analogous to that shown above.*



AB62 Rev 1.0

FEATURES

- **Development software supporting Altera's Stand Alone Microsequencer (SAM) series of EPLDs.**
- **State Machine Design Entry.**
- **Assembly Language Design Entry.**
- **User Definable Macros.**
- **Interactive Functional Simulator with Virtual Logic Analyzer user interface.**
- **Disassembler for examination of Assembly Code during simulation.**
- **Fully supports Horizontal Cascading of multiple SAMs.**
- **Runs on PC-XT, PC-AT, or compatible computers.**
- **Complete support of device programming through Altera programming hardware.**
- **Also available as a software only extension to existing Altera Development systems.**

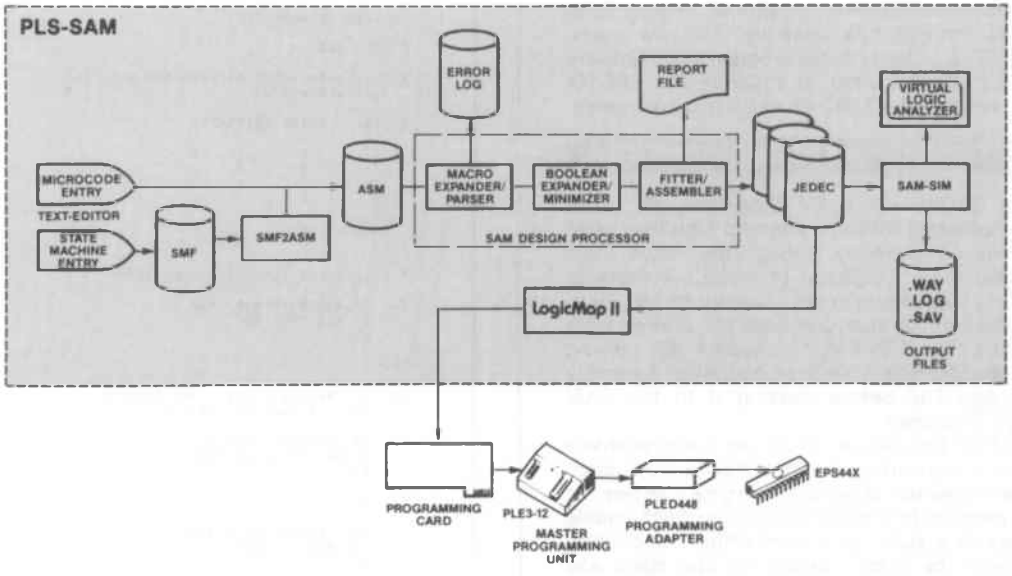
GENERAL DESCRIPTION

The Altera PLDS-SAM (Programmable Logic Development Software) represents a complete software solution to implementing State Machine and Microcoded applications into Altera's SAM family of Function-Specific EPLD's. PLS-SAM is a comprehensive, easy to use system that encompasses design entry with SAM+PLUS, design debugging with SAMSIM, and device programming with the Altera programming hardware.

The SAM+PLUS processing software accepts two forms of design entry and automatically generates an industry standard JEDEC file object code. SAMSIM is an interactive functional simulator created specifically for verification of State Machine and Microcoded designs implemented in SAM EPLDs.

For existing Altera PLDS or PLCAD users, PLS-SAM (Programmable Logic Software) is available as a software enhancement to their current system.

Figure 1. PLDS-SAM Block Diagram



FUNCTIONAL DESCRIPTION

Figure 1 shows a Block Diagram of the SAM+PLUS development system. Design entry with SAM+PLUS is done with either the Altera State Machine Input Language (ASMILE) or the Altera Assembly Language (ASM). With either method, a text-editor is used to create the input file. If the ASMILE language is used, a State Machine to Assembly converter will produce an Assembly Language file (ASM). The ASM file is passed on to the various modules that make up the SAM Design Processor (SDP). The SDP produces 3 outputs: an industry standard JEDEC file used to program the SAM EPLD, an Error Log file, and a Utilization Report file showing how the resources within the device have been used.

Once the JEDEC file is produced, the user may simulate the design using the SAMSIM functional simulator. SAMSIM provides an interactive design debugging environment. SAMSIM's Virtual Logic Analyzer provides on-screen examination of input and output waveforms and the Disassembler converts object code back into the original Assembly Language source code during simulation.

Horizontal cascading (using multiple SAM devices to increase the number of outputs) is fully supported including design entry, processing, simulation, and programming. The multiple SAM parts are listed in a single source file, and separate Report and JEDEC files are created for each.

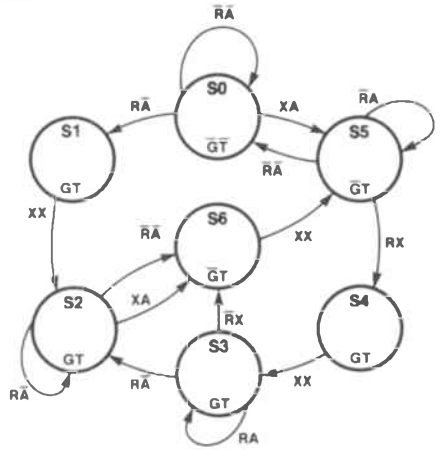
Finally, the user can program the SAM device with the LogicMap software and programming hardware. Users who have access to an Altera development system may use the same hardware together with PLS-SAM software to program SAM devices through new adapters. For new users, PLDS-SAM, includes all the programming hardware and software required to program the EPS44x parts using a PC-XT, PC-AT, or compatible system.

STATE MACHINE DESIGN ENTRY

The SAM+PLUS (SAM Programmable Logic User Software) software supports high-level state machine design entry through the Altera State Machine Input Language (ASMILE). A designer uses this language with any standard text-editor to create a text file that describes the desired state machine. The SMF2ASM convertor will convert the State Machine File into an equivalent Assembly Language File before passing it to the SAM Design Processor.

ASMILE provides a simple yet comprehensive means of converting a conceptual state diagram into a simple text description. Figure 2 shows the state diagram for a 68020 bus arbiter. Each bubble represents a state, the values within the bubbles represent the output values for that state, and the expressions on the arrows represent the conditional branches between states.

Figure 2. State Diagram for 68020 Bus Arbiter



R — BUS REQUEST INPUT
A — BUS GRANT ACKNOWLEDGE INPUT
G — BUS GRANT OUTPUT
T — THREE-STATE CONTROL TO BUS CONTROL LOGIC
X — DON'T CARE

Figure 3. State Machine Input File

```
DESIGNER NAME
COMPANY NAME
4/1/87
68020 Bus Arbitration Controller for SAM


PART:EPS444

INPUTS: REQUEST ACK

OUTPUTS: GRANT TRISTATE

MACHINE: BUSARBITER

CLOCK: CLK

% The State table defines the outputs
  for each state                      %

STATES: [GRANT TRISTATE]

  S0    [  0      0    ]
  S1    [  1      1    ]
  S2    [  1      1    ]
  S3    [  1      1    ]
  S4    [  1      1    ]
  S5    [  0      1    ]
  S6    [  0      1    ]

% Transition Specifications follow %

S0: IF REQUEST*/ACK THEN S1
    IF ACK THEN S5
    S0

S1: S2

S2: IF /REQUEST*/ACK + ACK THEN S6
    S2    % Implied ELSE  %

S3: IF /REQUEST THEN S6
    IF REQUEST*/ACK THEN S2
    S3

S4: S3

S5: IF /REQUEST*/ACK THEN S0
    IF REQUEST THEN S4
    S5

S6: S5

END$
```

Figure 3 shows the Altera State Machine Input Language representation of the same state machine. Notice that the states and their respective outputs have been defined in the STATES section using a truth table format. The transitions between the states have been defined with a simple IF-THEN construct. Once this file is created, it can be passed on to the SMF2ASM converter with no further modifications.

ASSEMBLY LANGUAGE DESIGN ENTRY

Direct Assembly Language design entry is also available for those who prefer to approach SAM as a microcoded controller. This entry method provides access to the advanced features of the SAM family including the on-chip Stack and Loop Counter. There are 13 instructions that directly control such functions as multi-way branching, sub-routines, nested for-next loops, and dispatch calls (jumping to an externally specified address).

In addition, user-defined Macros are available which allow users to define their own instruction mnemonics. This provides a higher level design entry approach. Macros are also useful for defining output values for various output fields so that the designer does not have to work at the binary level.

Figure 4 shows an example of an Assembly Language file. In this file, Macros have been used to define the 7 new instructions "GOTOS0" through "GOTOS6".

DESIGN PROCESSOR

The SAM Design Processor (SDP) takes an Assembly Language file and creates an optimized JEDEC file for the targeted device. The SDP first expands Macros that have been defined by the user. It then parses the design, listing any syntax or connection errors in an Error Log file. Next it minimizes the Boolean expressions that define the transition conditions. Finally, it fits the design into the SAM EPLDs, generating a separate JEDEC file for each. A Utilization Report file is also produced showing how and where the various instructions were implemented.

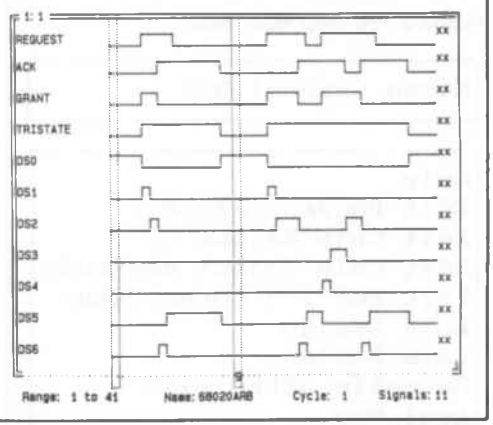
FUNCTIONAL SIMULATION—SAMSIM

Once a design has been processed and a JEDEC file created, it can be simulated with the SAMSIM Functional Simulator. SAMSIM provides a comprehensive design debugging environment. The Virtual Logic Analyzer displays the input and output waveforms interactively providing such features as multiple zoom levels, split screen, and differential time display. The internal state of the SAM device, including the Stack and Counter, can be examined and modified at will. In addition, an on-line disassembler converts the actual object code back into the original Assembly Language source code.

Figure 4. Assembly Language Input File

```
DESIGNER NAME
COMPANY NAME
4/1/87
68020 Bus Arbitration Controller for SAM

PART: EPS444

INPUTS: REQUEST ACK

OUTPUTS: GRANT TRISTATE

MACROS:
GOTOS0 = "[00] JUMP S0"
GOTOS1 = "[11] JUMP S1"
GOTOS2 = "[11] JUMP S2"
GOTOS3 = "[11] JUMP S3"
GOTOS4 = "[11] JUMP S4"
GOTOS5 = "[01] JUMP S5"
GOTOS6 = "[01] JUMP S6"

PROGRAM:

0D:   GOTOS0;

S0:   IF REQUEST#/ACK THEN GOTOS1;
      ELSEIF ACK THEN GOTOS5;
      ELSE GOTOS0;

S1:   GOTOS2;

S2:   IF /REQUEST#/ACK+ACK THEN GOTOS6;
      ELSE GOTOS2;

S3:   IF /REQUEST THEN GOTOS6;
      ELSEIF REQUEST#/ACK THEN GOTOS2;
      ELSE GOTOS3;

S4:   GOTOS3;

S5:   IF /REQUEST#/ACK THEN GOTOS0;
      ELSEIF REQUEST THEN GOTOS4;
      ELSE GOTOS5;

S6:   GOTOS5;

END$
```

Figure 5. Screen from Virtual Logic Analyzer



FEATURES

- **Supports all programmable options of EPB2001.**
- **Table-drive Design Entry.**
- **Real time error checking.**
- **Automatic report generation for documentation.**
- **Runs on IBM-PC and PS/2 computers or compatibles.**

GENERAL DESCRIPTION

Altera's MC Map development software supports all programmable options contained in the EPB2001 EPLD. MC Map features interactive, table-driven design entry with real-time error checking and automatic report generation for documentation. The designer is prompted for information concerning the programmable portions of his design: Board I.D., Chip Select Ranges, POS register control for address remapping, and POS I/O crosspoint configuration.

The MC Map entry table automatically validates the information entered and will make any necessary corrections to comply with the underlying hardware in the EPB2001 device. Address decode chip select ranges may be entered with either I/O memory or binary format. Once the design has been entered the MC Map compiles the information in minutes and generates a JEDEC file used to program the EPB2001.

Figure 1. MC Map Block Diagram

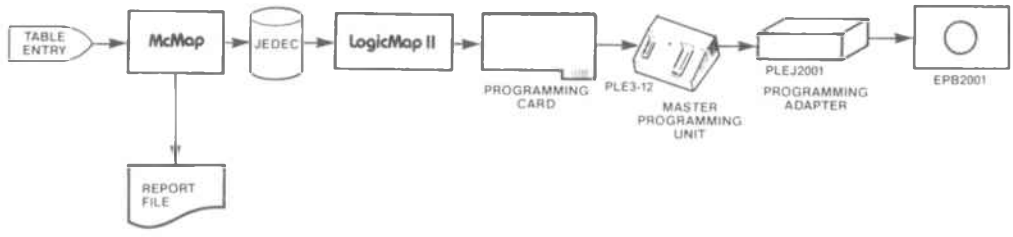
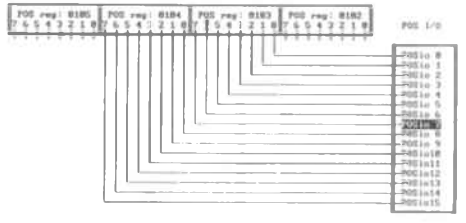


Figure 2. MC Map Main Menu

```
Micro Channel Map

Help
Edit Documentation...
Edit Chip Select...
Edit Chip Select controls
Edit POS I/O connections
Load Design
Save Design
Assemble JEDEC file
LogicMap
Quit                    <Esc>
```

Figure 3. POS I/O Connections



INTRODUCTION

Altera Applications provides several software utility programs which complement the A+PLUS development system. Each utility program has been written to address customer needs while designing EPLDs. For example, the AVEC utility adds functional test vectors to JEDEC files.

Each utility program listed below is available to current software warranty (PLAESW) holders via Altera's Electronic Bullentin Board under the file area EAU (Electronic Application Utilities). The Bullentin Board phone number is (408) 249-1100. Utility programs operate with an IBM-XT or compatible with MS-DOS 2.1 or later revisions.

ADDRESS DECODER

"DECODER" utility provides automatic Boolean equation generation for address decoding applications. The program accepts a user-specified address bus width with upper and lower address bounds. The equations generated can then be placed into an ADF file and compiled by A+PLUS.

ALTERANS][

Alterans][is a menu driven, interactive, text entry and editor program allowing EPLD designs to be represented via Boolean equations. Alterans provides step by step menus for design entry, logic minimization, and JEDEC file construction for the EP310 and EP320 EPLDs. In addition, the software automatically creates an Altera Design File (ADF) which can be compiled by A+PLUS to produce JEDEC files for the EP600, EP610, EP900, EP910, EP1210 EPLDs.

AVEC

AVEC is a utility program which adds functional test vectors to EPLD JEDEC files. AVEC translates the table output files generated from Altera's functional simulator (PLFSIM) into JEDEC Standard test vectors. Third party programmers such as DATA I/O 29B and Unisite 40 machines have built-in hardware drivers which can apply these vectors to the programmed EPLD. Note, since EPLDs have the benefit of generic testability, post programming functional testing is not required. Altera EPLDs are 100% tested at the factory before shipped to the customer. The use of post programming testing began with fuse programmble devices that cannot be fully tested at the factory.

BACKPIN

The BACKPIN utility extracts the pin assignments that are contained in an A+PLUS generated JEDEC file (which were assigned during design fitting) and places them into the LogiCaps schematic drawing. If A+PLUS performs automatic pin assignments, BACKPIN can then be used to place these software-determined pin assignments back into the LogiCaps schematic drawing. If design changes are then made, the new circuit will retain the same pin assignments.

JEDPACK

"JEDPACK" compacts the size of JEDEC files. This allows more disk space to available on your hard disk without the loss of EPLD programming information. This utility is handy for archieving JEDEC files.

JEDSUM

The JEDSUM utility calculates the EPROM data Check-Sum contained in the EPLD JEDEC file. Check-Sums are sometimes used to document programming files for each EPLD.

LOGICAPS HOUSTON INSTRUMENTS INTERFACE

An Altera customer has written an interface program between LogiCaps and Houston Instruments plotters. This EAU provides the information on how to obtain this interface.

PAL2EPLD

"PAL2EPLD" provides conversion utilities for translating 20 pin PAL designs into EP320's. Two conversion programs are provided. The first program allows PALASM source code to be converted to an Altera ADF file. The ADF file can then be compiled by A+PLUS to produce a JEDEC file. The second program directly converts PAL JEDEC files into EP320 JEDEC files.

310 to 320 JEDEC FILE CONVERTOR

EP310 to EP320 Convertor will automatically convert EP310 JEDEC files to EP320 JEDEC files.

AB73 Rev 1.0

INTRODUCTION

An Electronic Bulletin Board System (BBS) is in place to provide continuous access to Applications information. The Bulletin Board provides up-to-date device and development tool information, Electronic Applications Briefs, useful Utility Programs, and serves as a medium for transfering files to and from Applications. Owners of A+PLUS (Altera Programmable Logic User's System) may refer to Appendix E of the A+PLUS Reference Guide for more details. Access to this service can be gained by calling:

(408) 249-1100

Requirements for connection via modem are:
1) Baud rate is 300 or 1200.
2) Bell Standard 212A modem or compatible.
3) Data format is: 8 bit data, 1 stop bit, no parity.
4) File transfer protocols supported are: Crosstalk, Xmodem, Kermit, ASCII, Minitel, Modem7, and Telink

After connection, press space key twice for the first menu. A password is required. Non-registered Altera users may logon using the name GUEST, and the password EPLD. Once a customer logs onto the service, menus guide the way. The BBS provides many services:

TO ALTERA/FROM ALTERA

File Area 1 & 2: "To: Altera/From: Altera" section is used to upload and download customer files where a problem or question requires our analysis and/or correction of the customer's file.

APPLICATION NOTES and BRIEFS

File Area 3: Electronic Applications Notes and Briefs (EABs) are put on the BBS at the same time they are released for printing, providing a quick way to get the latest information.

SOFTWARE UTILITIES

File Area 4: Software Utilites (EAUs) are available online through the BBS. See Utility Software Program section of this Handbook for a complete listing.

DATA SHEET SPECIFICATIONS

File Area 5: Data Sheet Specifications lists updates and changes for each EPLD.

TECHNICAL ALERTS

File Area 6: Technical Alerts from Altera Marketing and Applications provide up-to-date information for Altera products.

NEW PRODUCTS

File Area 7: This area is provided for Altera New Product information. Current software versions is also listed here.

DEMO DISKS

File Area 8: Demo Disks for Altera's Development Tools are available here.

ADLIB EXCHANGE

File Area 10 & 11: An ADLIB directory is available to allow a customer "swap area" for customer and Altera additions to the TTL MacroFunction Library.

The goal of all these services is to provide unsurpassed Applications customer support. Please contact Applications with any questions, comments, or suggestions.

3

RANDOM LOGIC INTEGRATION APPLICATIONS **PAGE NO.**

FEATURES

- **Estimation Formula.**
- **Estimation Worksheet.**

INTRODUCTION

This Application Brief provides design guidelines and an estimating formula to help determine which EPLD best implements the logic contained in a TTL design. It is assumed the user is familiar with EPLD architecture and device characteristics. Refer to the Altera Databook for complete description of each EPLD. To get started on an EPLD design, the following sequence of steps is suggested. If you have any questions during the design cycle call Altera Applications at (408) 984-2805 x102.

PARTITIONING

First, partition the design into functional blocks. Major functional blocks may be expressed in standard MSI TTL form for integration within the desired EPLD. Should the design require a multiple EPLD solution, the I/O connections between the EPLDs should be minimized. The complete schematic should be structured as a set of sub-systems such as counters, shift registers, comparators, basic gates, flip-flops, etc., to allow easy design entry.

TIMING SPECIFICATIONS

Knowledge of basic clock frequency and critical timing paths are necessary to make the correct choice of EPLDs. Critical timing paths are determined based on total propagation delay (tpd), maximum clock frequency (fcnt), set-up time (tsu) and clock to output delay (tco1). Refer to the respective EPLD AC specifications. See the EPLD Timing Simulation for assistance.

ESTIMATING A FIT

To estimate the amount of logic which can fit into a specific EPLD, the number of dedicated input and output pins, and the total number of Macrocells used by the logic must be determined.

To estimate the number of Macrocells used by the design, determine:

1. The number of buried flip-flops (flip-flops that do not drive output pins).
2. The total number of Macrocells required by MacroFunctions.

Each member of the MacroFunction library has a maximum number of Macrocells used to build the function. This number is shown in the lower right hand corner of the symbol. Note that some MacroFunctions have no Macrocell specification. These functions use only a portion of the logic array, thus other logic may be added before the entire Macrocell is used.

Since basic combinatorial gates (NAND, OR, XOR, etc.) are implemented within the EPLD Logic Array, they do not require an entire Macrocell, and may be safely ignored in this estimate.

Estimating Formula:

Refer to the Device Selector Guide located in the front of this Handbook to determine the total number of Inputs, I/O, and Macrocells available for each EPLD. From your design:

(a) Determine the number of output pins = OP
(b) Determine the number of input pins = IP

where
IP = Number of inputs - Total EPLD Inputs
(Schematic) (Device Selector Guide)
(if less than zero, enter zero)

This subtraction is necessary since I/O pins may be used as inputs.

(c) Determine the number of Macrocells = BFF + MR

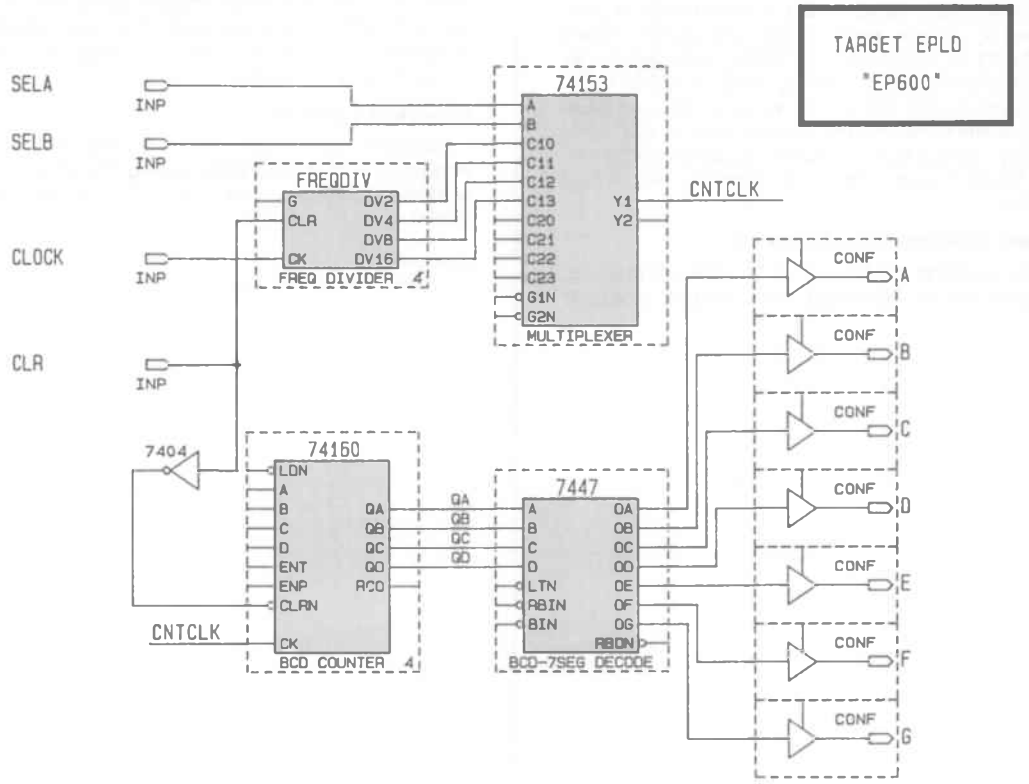
where BFF = Buried Flip-flops
(BFF = D, T, JK or SR flip-flops, which do not drive output pins.)

and MR = MacroFunction Requirements (located in right hand corner of symbol).

If OP + IP + BFF + MR is less than the Total Macrocell availability (listed below) then the design will most likely fit into the desired EPLD. Complete the design using LogiCaps and produce a Schematic Drawing (SD) file.

EPLD	MacroCells
EP1800	48
EP1810	48
EPB1400	20 (contains dedicated Octal Registers and Bus Transceiver Port)
EP1210	28
EP910	24
EP900	24
EP610	16
EP600	16
EP320	8
EP310	8

Figure 1. Example Circuit



DESIGN ENTRY

Design entry is quick and simple using the LogiCaps Schematic Capture software in conjunction with a three button mouse. LogiCaps symbol library contains symbols to support all possible EPLD architectures. These include basic gates, flip-flops, and I/O structures. In addition, TTL MacroFunctions are provided allowing high-level design entry. Boolean equations may also be directly entered into the schematic. LogiCaps supports such features as split windows, multiple zoom levels, orthogonal rubber-banding, automatic tag and drag of symbols, area editing, save and load functions. Schematics may be printed or plotted for final documentation.

HELPFUL HINTS

The following features should be noted when designing with MacroFunctions. Refer to section 6 (MacroFunction Tutorial) of the LogiCaps User Manual for a complete description.

MacroMunching

The A+PLUS software automatically removes any unused logic within a MacroFunction. This feature, called "MacroMunching", allows designers to freely employ MacroFunctions without the headaches of optimizing their use. For example, if a 74374 octal register is contained in the schematic, but only 6 of the 8 outputs are connected, A+PLUS automatically extracts the logic associated with the 2 unused outputs from the design. Thus, the total Macrocell count for this MacroFunction is 6 rather than 8.

I/O Architecture Compression

A+PLUS automatically performs architecture compression of I/O primitives. MacroFunction outputs which drive I/O pins via a CONF primitive are compressed inside the EPLD I/O structure. Therefore any MacroFunction output which drives an I/O pin (CONF) should not be added to the MacroFunction Requirements section (MF) of the estimation formula. See page 6-12 of the LogiCaps Users Manual.

Input Default Values

Each MacroFunction contains intelligent input default values (VCC or GND). If a MacroFunction input(s) is not used, each one automatically defaults to either VCC or GND. Users no longer have the burden of wiring VCC or GND to all unused inputs. For example, an active low Clear signal will have an input default value of VCC, thus always disabling the function. Refer to the MacroFunction Documentation for specific input default values.

Altera Provided MacroFunctions

In addition to familiar TTL MacroFunctions, Altera has also provided some custom functions optimized for EPLD architecture. These include counters such as 4COUNT which uses T flip-flops rather than D type. When implementing counters inside EPLDs, it is recommened to use MacroFunctions such as 4COUNT for most efficient EPLD resource utilization.

Estimation Example

Consider the circuit shown in Figure 1. The targeted EPLD is the EP600. Using the estimation worksheet, Figure 2, the design will fit into the EP600.

AB60 Rev 1.1

Figure 2. Estimation Worksheet

1. INPUT PINS IP = 4-4 = 0

SELA	SELB	CLOCK	CLR

2. OUTPUT PINS OP = 7

A	B	C	D
E	F	G	

3. BURIED LOGIC BFF = 0

DFF = 0 TFF = 0 JKFF = 0 SRFF = 0

4. TTL LOGIC MF = 8
(MACROFUNCTIONS)

FREQDIV = 4 74153 = 0
74160 = 4 7447 = 0

5. TOTAL MACROCELLS = IP + OP + BFF + MF = 0 + 7 + 0 + 8 = 15

6. TARGET EPLD (CIRCLE EPLD WITH PACKAGE TYPE AND SPEED GRADE)

EP1800JC	EP900DC	EP910DC-40	EP600DC	EP610DC-35	EP320DC
EP1800JC-3	EP900DC-3	EP910DC-35	EP600DC-3	EP610DC-30	EP320DC-2
EP1800JC-2	EP900DC-2	EP910DC-30	EP600JC	EP610DC-25	EP320DC-1
	EP900JC	EP910JC-40	EP600JC-3	EP610JC-35	EP310DC
	EP900JC-3	EP910JC-35		EP610JC-30	EP310DC-3
	EP900JC-2	EP910JC-30		EP610JC-25	EP310DC-2

ESTIMATION WORKSHEET

1. INPUT PINS IP = _____

_____ _____ _____ _____

_____ _____ _____ _____

_____ _____ _____ _____

_____ _____ _____ _____

2. OUTPUT PINS OP = _____

_____ _____ _____ _____

_____ _____ _____ _____

_____ _____ _____ _____

_____ _____ _____ _____

3. BURIED LOGIC BFF = _____

DFF = ____ TFF = ____

JKFF = ____ SRFF = ____

4. TTL LOGIC MF = _____
(MACROFUNCTIONS)

_____ _____

_____ _____

_____ _____

5. TOTAL MACROCELLS = IP + OP + BFF + MF = _____

6. TARGET EPLD (CIRCLE EPLD WITH PACKAGE TYPE AND SPEED GRADE)

EP1800JC	EP900DC	EP910DC-40	EP600DC	EP610DC-35	EP320DC
EP1800JC-3	EP900DC-3	EP910DC-35	EP600DC-3	EP610DC-30	EP320DC-2
EP1800JC-2	EP900DC-2	EP910DC-30	EP600JC	EP610DC-25	EP320DC-1
	EP900JC	EP910JC-40	EP600JC-3	EP610JC-35	EP310DC
	EP900JC-3	EP910JC-35		EP610JC-30	EP310DC-3
	EP900JC-2	EP910JC-30		EP610JC-25	EP310DC-2

FEATURES

- **Binary, Decade and Gray Code Counters.**
- **Load, Enable, Clear and Cascade Options.**

INTRODUCTION

Counters, often digital design's most useful building blocks, are easily constructed in EPLDs. This Application Brief discusses efficient EPLD counter design. A variety of counters are presented using both schematic capture and Boolean equation design entry methods.

CHOOSING A FLIP-FLOP

Toggle flip-flops are the answer to simple and efficient counter design. To see the advantages offered by toggle flip-flops, compare the two 8 bit binary counters shown in Figure 1 and 2. Figure 1 uses D flip-flops. An additional product term is required for each successive significant bit, (see Equations section). The most significant bit, Q7, requires 9 product terms. The counter in Figure 2 uses toggle or T-type flip-flops. Each counter bit requires only one product-term. Thus, D type flip-flops require significant more gated logic to construct the equivalent counter function.

Altera EPLDs have 8 product terms per macrocell. Using D flip-flops, one could only fit a 7 bit counter within the EPLD macrocells (the nth bit requires n+1 product terms). Altera EPLDs offer programmable flip-flops (D, T, JK, and SR). By choosing the T flip-flop, counters of any size are built without infringing on product term restrictions. Therefore, it is best to design counters with T flip-flops.

DESIGNING COUNTERS WITH LOGICAPS SCHEMATIC CAPTURE

TTL MACROFUNCTION COUNTERS

Altera's TTL MacroFunction library (see Altera Databook section PLSLIB-TTL) provides 22 predefined MSI counter functions. These counters range from Decade to Binary with such features as Load, Clear, and Enable, as well as cascade options. Table 1 lists counter options available from Altera's MacroFunction library. Note, some entries are TTL part numbers followed by a "T" (e.g. 74163T), indicating toggle (T-type) flip-flops were used to implement the counter. There are also some Altera designed counters such as 8COUNT (cascadable 8 bit up/down counter), FREQDIV is a 4 bit frequency divider and GRAY4 which is a 4 bit Gray code counter.

Table 1. Counter MacroFunctions

	TYPE				OPTIONS				
	Bits	Decade	Binary	Gray	Up	Down	Load	Clear	Enable
7493	4		x		x			x	
74160	4	x			x		x	x	x
74160T	4	x			x		x	x	x
74161	4		x		x		x	x	x
74161T	4		x		x		x	x	x
74162	4	x	x		x		x	x	x
74162T	4	x			x		x	x	x
74163	4		x		x		x	x	x
74163T	4		x		x		x	x	x
74190	4	x			x	x	x		x
74190T	4	x			x	x	x		x
74191	4		x		x	x	x		x
74191T	4		x		x	x	x		x
74192T	4	x			x	x	x	x	x
74193T	4		x		x	x	x	x	x
74393	4		x		x			x	
GRAY4	4		x	x	x				
4COUNT	4		x		x	x	x	x	x
8COUNT	8		x		x	x	x	x	x
FREQDIV	4		x		x				
16CUDSLR	16		x		x				

Figure 1. Counter with D-Type Flip-Flops— *Counters which use D-type flip-flops require one additional product term for each significant bit.*

```
Altera Corp
Feb. 29, 1988
1.00
A
EP1210
8-BIT BINARY UP COUNTER WITH D FLIP-FLOPS

PART: EP1210

INPUTS:  ENABLE, RESET, CLOCK

OUTPUTS: Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7

NETWORK:
     ENABLE = INP(ENABLE)
     CLR    = INP(RESET)
     CK     = INP(CLOCK)

     Q0,Q0  = RORP(Q0d,CK,CLR,,)
     Q1,Q1  = RORP(Q1d,CK,CLR,,)
     Q2,Q2  = RORP(Q2d,CK,CLR,,)
     Q3,Q3  = RORP(Q3d,CK,CLR,,)
     Q4,Q4  = RORP(Q4d,CK,CLR,,)
     Q5,Q5  = RORP(Q5d,CK,CLR,,)
     Q6,Q6  = RORP(Q6d,CK,CLR,,)
     Q7,Q7  = RORP(Q7d,CK,CLR,,)

EQUATIONS:
        Q7d = /Q0 * Q7
            + /Q1 * Q7
            + /Q2 * Q7
            + /Q3 * Q7
            + /Q4 * Q7
            + /Q5 * Q7
            + /Q6 * Q7
            + /ENABLE * Q7
            + ENABLE * Q0 * Q2 * Q3 * Q4 * Q5 * Q6 * /Q7;

        Q6d = /Q0 * Q6
            + /Q1 * Q6
            + /Q2 * Q6
            + /Q3 * Q6
            + /Q4 * Q6
            + /Q5 * Q6
            + /ENABLE * Q6
            + ENABLE * Q0 * Q1 * Q2 * Q3 * Q4 * Q5 * /Q6;

        Q5d = /Q0 * Q5
            + /Q1 * Q5
            + /Q2 * Q5
            + /Q3 * Q5
            + /Q4 * Q5
            + /ENABLE * Q5
            + ENABLE * Q0 * Q1 * Q2 * Q3 * Q4 * /Q5;

        Q4d = /Q0 * Q4
            + /Q1 * Q4
            + /Q2 * Q4
            + /Q3 * Q4
            + /ENABLE * Q4
            + ENABLE * Q0 * Q1 * Q2 * Q3* /Q4;

        Q3d = /Q0 * Q3
            + /Q1 * Q3
            + /Q2 * Q3
            + /ENABLE * Q3
            + ENABLE * Q0 * Q1 * Q2 * /Q3;

        Q2d = /Q0 * Q2
            + /Q1 * Q2
            + /ENABLE * Q2
            + ENABLE * Q0 * Q1 * /Q2;

        Q1d = /Q0 * Q1
            + /ENABLE * Q1
            + ENABLE * Q0 * /Q1;

        Q0d = /ENABLE * Q0
            + ENABLE * /Q0;

END$
```

MacroFunction documentation, shown in Figure 3a-3c, provides symbol nomenclature, a function table, a declaration statement, and gate level logic schematic. If MacroFunction inputs are left unconnected, a default value (shown in parenthesis in Figure 3a) is used. For example VCC is the default value for LDN. Worst case-macrocell requirements are shown in the lower right hand corner of the symbol (4 macrocells for 4COUNT). The

Figure 2. Counter with T-Type Flip-Flops— *Counters which use T-type flip-flops require only one product term for each significant bit.*

```
Altera Corp
Feb. 29, 1988
1.00
A
EP610
8-BIT BINARY UP COUNTER WITH T FLIP-FLOPS

PART: EP610

INPUTS:  ENABLE, RESET, CLOCK

OUTPUTS: Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7

NETWORK:
     ENABLE = INP(ENABLE)
     CLR    = INP(RESET)
     CK     = INP(CLOCK)

     Q0,Q0  = TOTF(Q0t,CK,CLR,,)
     Q1,Q1  = TOTF(Q1t,CK,CLR,,)
     Q2,Q2  = TOTF(Q2t,CK,CLR,,)
     Q3,Q3  = TOTF(Q3t,CK,CLR,,)
     Q4,Q4  = TOTF(Q4t,CK,CLR,,)
     Q5,Q5  = TOTF(Q5t,CK,CLR,,)
     Q6,Q6  = TOTF(Q6t,CK,CLR,,)
     Q7,Q7  = TOTF(Q7t,CK,CLR,,)

EQUATIONS:
        Q7t =  ENABLE * Q0 * Q2 * Q3 * Q4 * Q5 * Q6;

        Q6t = ENABLE * Q0 * Q1 * Q2 * Q3 * Q4 * Q5;

        Q5t = ENABLE * Q0 * Q1 * Q2 * Q3 * Q4;

        Q4t = ENABLE * Q0 * Q1 * Q2 * Q3;

        Q3t = ENABLE * Q0 * Q1 * Q2;

        Q2t = ENABLE * Q0 * Q1;

        Q1t = ENABLE * Q0;

        Q0t =  ENABLE;

END$
```

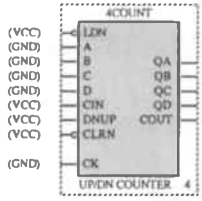
'Declaration' line shows the invocation for Boolean design entry.The function table in Figure 3b shows the device operation under all input conditions. The logic schematic in Figure 3c shows the gate level logic. When using MacroFunctions, check the ADLIB User Manual for any discrepancies with design requirements.

MACROMUNCHING

MacroMunching, contained in the A+PLUS software, provides automatic removal of unused portions of a MacroFunction. (See PLS2 Data Sheet and ADLIB USER Manual for complete description of macromunching). For example, if the design requires only a 3 bit counter, 4COUNT may still be used. By leaving the extra counter bit, QD, unconnected, its flip-flop and related gated logic will automatically be removed from the design, not consuming additional resources within the EPLD. Altera's MacroFunctions and macromunching capability allow counters of any size to be easily built in EPLDs. Figure 4 shows a 13 bit counter using MacroFunctions.

Figure 3. MacroFunction Documentation

a. 4COUNT symbol, Boolean declaration and input default values.



Name: **4COUNT** (4-Bit Up/Down Binary Counter With Synchronous Load and Asynchronous Clear)

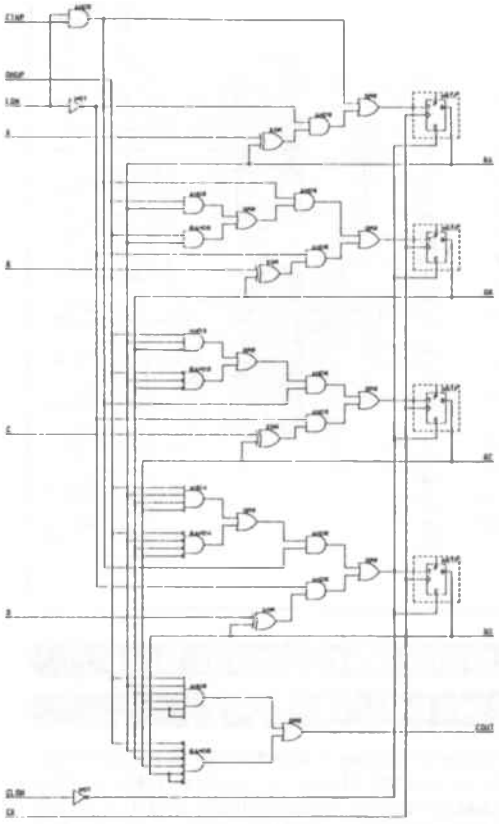
Declaration: 4COUNT(CLRN,LDN,DNUP,CIN,A,B,C,D, CK,QD,QC,QB,QA,COUT)

(LDN = Load, Active Low; CIN = Carry In; DNUP = Down/Up; CLRN = Clear, Active Low; CK = Clock; COUT = Carry Out)

EPLDs: EP600, EP610, EP900, EP910, EP1800, EPB1400

Default Signal Levels: GND — CK, A, B, C, D
VCC — CLRN, LDN, DNUP,CIN

c. Logic schematic using Altera primitives.



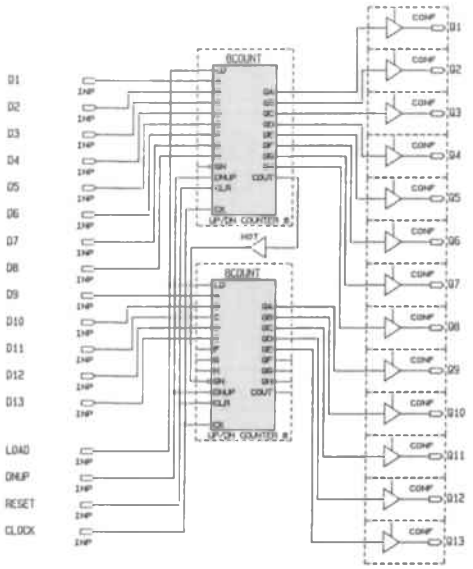
b. The function table describes the Macro-Function operation over all input conditions.

4COUNT Function Table

INPUTS									OUTPUTS				
CK	LDN	CLRN	DNUP	CIN	D	C	B	A	Q_D	Q_C	Q_B	Q_A	COUT
X	X	L	X	X					L	L	L	L	X
↑	L	H	X	X	d	c	b	a	d	c	b	a	X
↑	H	H	X	L					HOLD				X
↑	H	H	L	H					COUNT DOWN				L
↑	H	H	H	H					COUNT UP				L
↑	H	H	L	H					H	H	H	H	H
↑	H	H	H	H					L	L	L	L	H

H = high level (steady state)
L = low level (steady state)
X = don't care (any input including transitions)
↑ = transition from low to high level
a,b,c,d, = level of steady state input at inputs A,B,C,D

Figure 4. 13 Bit Up/Down Counter—*A 13-bit up/down counter is constructed cascading two 8-bit counters. Macromunching will automatically remove the unused bits.*



DESIGNING COUNTERS WITH BOOLEAN EQUATIONS

Figure 5-7 shows Altera Design Files (ADF) listings for a 4 bit binary up counter (74161), 4 bit decade up/down counter (74190T), and 8 bit binary up/down counter (8COUNT) with Load, Enable, and Clear options. Other counters may be designed by using the Boolean equations given in these Figures.

CONCLUSION

Counters are easily realized in EPLDs. TTL Macro-Functions provide ease of use and optimal utilization of EPLD resources. Whenever possible choose toggle flip-flop implementations of counters to save macrocell and product terms. Altera's A+PLUS software provides macromunching algorithms to optimize use of logic. Custom counter features can be created using cascading Macro-Functions or by using Boolean equations to represent the logic.

Figure 5. 4-Bit Binary Up Counter with Load and Clear

```
Altera Corporation
2/29/88
1.00
B
EP610
4-BIT BINARY UP COUNTER WITH
SYNCHRONOUS LOAD, ASYNCHRONOUS CLEAR

PART: EP610

INPUTS: /CLEAR, /LOAD, ENP, ENT, A, B, C, D, CLOCK

OUTPUTS: QD, QC, QB, QA, RCO

NETWORK:

        CLRn  = INP(/CLEAR)
        LDn   = INP(/LOAD)
        ENP   = INP(ENP)
        ENT   = INP(ENT)
        A     = INP(A)
        B     = INP(B)
        C     = INP(C)
        D     = INP(D)
        CK    = INP(CLOCK)

       QA,QA  = RORF(QAd,CK,CLR,GND,VCC)
       QB,QB  = RORF(QBd,CK,CLR,GND,VCC)
       QC,QC  = RORF(QCd,CK,CLR,GND,VCC)
       QD,QD  = RORF(QDd,CK,CLR,GND,VCC)
       RCO    = CONF(RCOc,VCC)

         CLR  = NOT(CLRn)                  % ACTIVE LOW CLEAR %

EQUATIONS:

RCOc = ENT*QD*QC*QB*QA;

QDd  = LDn'*D                  % LOAD %

     + ENT'*LDn*QD             % HOLD %
     + ENP'*LDn*QD

     + LDn*QD*QA'              % COUNT %
     + LDn*QD*QB'
     + LDn*QD*QC'
     + ENT*ENP*LDn*QD'*QA*QB*QC;

 QCd = LDn'*C
     + ENT'*LDn*QC
     + ENP'*LDn*QC
     + LDn*QC*QA'
     + LDn*QC*QB'
     + ENT*ENP*LDn*QC'*QA*QB;

 QBd = LDn'*B
     + ENT'*LDn*QB
     + ENP'*LDn*QB
     + LDn*QB*QA'
     + ENT*ENP*LDn*QB'*QA;

 QAd = LDn'*A
     + ENT'*LDn*QA
     + ENP'*LDn*QA
     + ENT*ENP*LDn*QA';

END$
```

Figure 6. 4-Bit Up/Down Decade Counter with Load

```
Altera Corporation
2/29/88
1.00
A
74190T
4-BIT UP/DOWN DECADE COUNTER

PART: EP610

INPUTS: /LOAD, DNUP, /ENABLE, CLOCK, A, B, C, D

OUTPUTS: QA, QB, QC, QD, /RCO, MNMX

NETWORK:

 Gn   = INP(/ENABLE)        A = INP(A)
 LDn  = INP(/LOAD)          B = INP(B)
DNUP  = INP(DNUP)           C = INP(C)
 CK   = INP(CLOCK)          D = INP(D)

QA,QA = TOTF(QAt, CK, GND, GND, VCC)
QB,QB = TOTF(QBt, CK, GND, GND, VCC)
QC,QC = TOTF(QCt, CK, GND, GND, VCC)
QD,QD = TOTF(QDt, CK, GND, GND, VCC)

/RCO  = CONF(RCOc, VCC)
MNMX  = CONF(MNMXc,VCC)


EQUATIONS:
TEN   = (LDn # QA # QD # /DNUP
      +  LDn # /QA # /QD # DNUP # /QB # /QC)';

MNMXc = QD # QA # /DNUP # /QC #/QB # /Gn # LDn
      + /QD # /QA # DNUP # /QC # /QB # /Gn # LDn;

RCOc  = (QD # QA # /DNUP # /QC # /QB # /Gn # LDn
      + /QD # /QA # DNUP # /QC # /QB # /Gn # LDn)';

 QDt  = /Gn # /QD # D # /LDn
      + /Gn # QD # /D # /LDn
      + /Gn # QD # LDn # QA # /DNUP
      + /Gn # LDn # /QA # DNUP # /QB # /QC
      + /Gn # LDn # QA # /DNUP # QB # QC;

 QBt  = TEN # /Gn # /QB # B # /LDn
      + TEN # /Gn # QB # /B # /LDn
      + TEN # /Gn # LDn # /QA # DNUP
      + TEN # /Gn # LDn # QA # /DNUP;

 QCt  = TEN # /Gn # /QC # C # /LDn
      + TEN # /Gn # QC # /C # /LDn
      + TEN # /Gn # LDn # /QA # /QB # DNUP
      + TEN # /Gn # LDn # QA # QB # /DNUP;

 QAt  = /Gn # LDn
      + /Gn # QA # /A
      + /Gn # /QA # A;

END$
```

Figure 7. 8-Bit Up/Down Counter with Load and Clear

```
Altera Corporation
2/29/88
1.00
A
EP910
8 BIT BINARY UP/DOWN COUNTER
WITH LOAD, ENABLE, AND CLEAR

PART: EP910

INPUTS: LOAD, ENABLE, DNUP, CLEAR, CLOCK,
        A, B, C, D, E, F, G, H

OUTPUTS: QA, QB, QC, QD, QE, QF, QG, QH, COUT

NETWORK:

          LD = INP(LOAD)        A = INP(A)
          CK = INP(CLOCK)       B = INP(B)
         CLR = INP(CLEAR)       C = INP(C)
        DNUP = INP(DNUP)        D = INP(D)
      ENABLE = INP(ENABLE)      E = INP(E)
                                F = INP(F)
                                G = INP(G)
                                H = INP(H)

       QA,QA = TOTF(QAt, CK, CLR, GND, VCC)
       QB,QB = TOTF(QBt, CK, CLR, GND, VCC)
       QC,QC = TOTF(QCt, CK, CLR, GND, VCC)
       QD,QD = TOTF(QDt, CK, CLR, GND, VCC)
       QE,QE = TOTF(QEt, CK, CLR, GND, VCC)
       QF,QF = TOTF(QFt, CK, CLR, GND, VCC)
       QG,QG = TOTF(QGt, CK, CLR, GND, VCC)
       QH,QH = TOTF(QHt, CK, CLR, GND, VCC)
       COUT  = CONF(COUTc,VCC)

EQUATIONS:
      COUTc = /QH # /DNUP # /QA # /QB # /QC #
              /QD # /QE # /QF # /QG # /LD$/ENABLE
            + QH # DNUP # QA # QB # QC #
              QD # QE # QF # QG # /LD$/ENABLE;

        QHt = LD # /QH # H                          % LOAD EQUATION %
            + LD # QH # /H

            + /QA # /QB # /QC # /QD # /QE #   % COUNT DOWN EQUATION %
              /QF # /QG # /DNUP # /LD$/ENABLE

            + QA # QB # QC # QD # QE #
              QF # QG # DNUP # /LD$/ENABLE;   % COUNT UP EQUATION %

        QGt = LD # /QG # G
            + LD # QG # /G
            + /QA # /QB # /QC # /QD # /QE #
              /QF # /DNUP # /LD$/ENABLE
            + QA # QB # QC # QD # QE #
              QF # DNUP # /LD$/ENABLE;

        QFt = LD # /QF # F
            + LD # QF # /F
            + /QA # /QB # /QC # /QD # /QE #
              /DNUP # /LD$/ENABLE
            + QA # QB # QC # QD # QE #
              DNUP # /LD$/ENABLE;

        QEt = LD # /QE # E
            + LD # QE # /E
            + /QA # /QB # /QC # /QD # /DNUP #
              /LD$/ENABLE
            + QA # QB # QC # QD # DNUP #
              /LD$/ENABLE;

        QDt = LD # /QD # D
            + LD # QD # /D
            + /QA # /QB # /QC # /DNUP # /LD$/ENABLE
            + QA # QB # QC # DNUP # /LD$/ENABLE;

        QCt = LD # /QC # C
            + LD # QC # /C
            + /QA # /QB # /DNUP # /LD$/ENABLE
            + QA # QB # DNUP # /LD$/ENABLE;

        QBt = /QA # /DNUP # /LD$/ENABLE
            + LD # /QB # B
            + LD # QB # /B
            + QA # DNUP # /LD$/ENABLE;

        QAt = /ENABLE # /LD
            + LD # /A # QA
            + LD # A # /QA;

END$
```

FEATURES

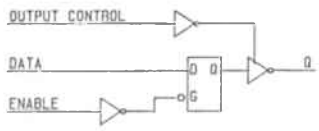
- **Transparent D-Type latch (i.e. 74LS373)**
- **R-S Latch (i.e. 74LS279)**
- **Three State Output Capability**
- **Programmable Input/Output Polarity**

D-TYPE LATCH

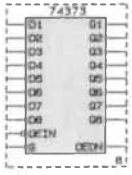
Asynchronous D-type latches commonly implemented with 74LS373 TTL devices hold data at the latch input until a control signal is applied. Once enabled, the output will follow the data input. Figure 1 shows the symbol representation, MacroFunction, function table, and gate level logic.

Figure 1. Transparent D-Type Latch

a. Symbolic Representation



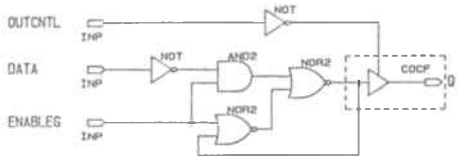
b. MacroFunction



c. Function Table

Output Control	Enable (G)	Data	Output (Q)
L	H	H	H
L	H	L	L
L	L	X	Q
H	X	X	Z

d. Logic Schematic



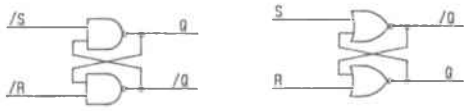
S-R LATCH

Asynchronous R-S latches are constructed with either cross-coupled NOR or NAND gates. For the NOR-NOR implementation, the output will go to a logical one (HIGH) if the input SET is HIGH. The output will produce a logical zero (LOW) if the input RESET is HIGH. The symbolic representation, Altera provided MacroFunctions, function table, and gate level logic is given in Figure 2.

To design either latch into an EPLD requires only one Macrocell using combinatorial feedback. The ADF files for the Transparent D-type latch and SR latch are included in Figure 3 and 4.

Figure 2. S-R Latch

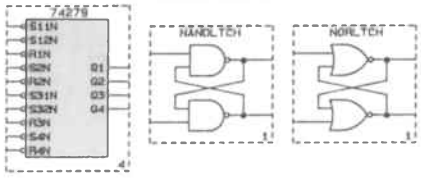
a. Symbolic Representation



Active low inputs

Active high inputs

b. MacroFunctions



c. Function Table

Nandlatch

/S	/R	Q
L	L	Q
L	H	H
H	L	L

Norlatch

S	R	Q
L	L	Q
L	H	L
H	L	H

d. Logic Schematic

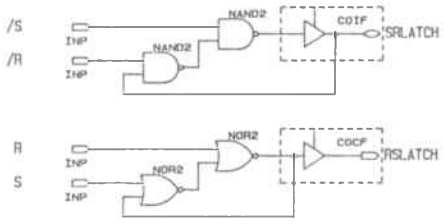


Figure 3. Altera Design File—Transparent Latch

```
Altera
7/29/85
1.0
EPLD
TRANSPARENT D-TYPE LATCH

OPTIONS:TURBO=OFF

PART:AUTO                          % Automatic part selection %

INPUTS:DATA,ENABLEG,OUTCNTL

OUTPUTS:Q

NETWORK:

DATA    = INP(DATA)
ENABLEG = INP(ENABLEG)
OUTCNTL = INP(OUTCNTL)

Q,Q = COCF(Qc,OE)

EQUATIONS:

Qc = /((/DATA*ENABLEG) + /(ENABLEG + Q));  % Active high output %

OE = /OUTCNTL;

END$
```

Figure 4. Altera Design File—S-R Latch

```
Altera
7/29/85
1.0
EPLD
SR-LATCH

OPTIONS:TURBO=OFF

PART:AUTO

INPUTS: S,R

OUTPUTS: RSLATCH
NETWORK:

S = INP(S)
R = INP(R)

RSLATCH,Q = COCF(RSLATCHc,VCC)

EQUATIONS:

RSLATCHc = /(/(S+Q)+R);        % NOR-NOR implementation %

END$
```

PROGRAMMABLE OPTIONS

Altera EPLDs offer increased flexibility for specifying the latch operation. For the transparent latch, the output polarity is active high (Q=DATA when ENABLEG=HIGH), when the DATA input pin drives the inverter (NOT) gate as shown in Figure 1.d. For an active low output (Q=/DATA when ENABLEG=1) remove the inverter gate and connect the DATA input pin directly to the AND gate. In addition, the OUTPUT CONTROL and ENABLEG inputs can also be defined active high or low. For example, connecting an inverter gate after the ENABLEG input pin will cause the circuit to pass DATA when ENABLEG=LOW and latch DATA when ENABLEG=HIGH.

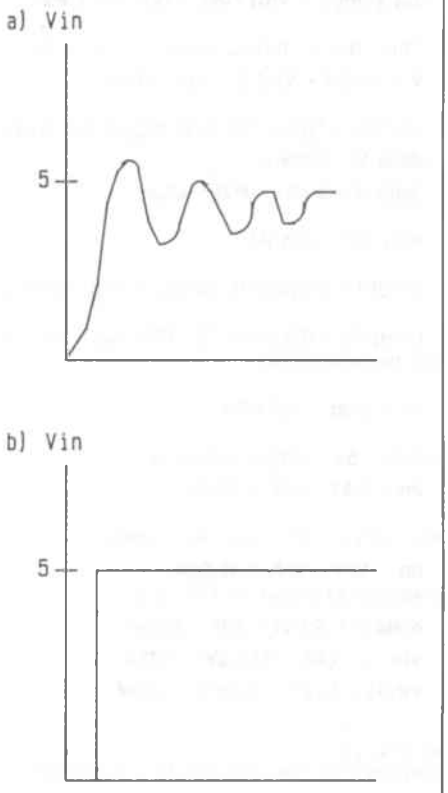
For Altera EPLDs whose I/Os do not support direct combinatorial feedback (EP320, EP600, EP610, EP900, and EP910), input feedback may be substituted to route the signal back into the AND array. For example, the SR-Latch implemented with NAND gates uses COIF I/O architecture rather than COCF, see Figure 2.d. When using the COIF, the output must always be enabled to ensure the feedback path is connected to the logic.

AB9 Rev 2.0

INTRODUCTION

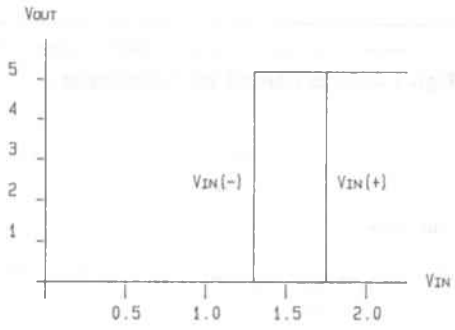
Digital signals with slow and noisy rising and falling edges degrade a digital logic element performance and reliability. In receiver applications, a digital transmitter may send "clean" signals with fast rising and falling edges, however transmission line parasitic inductance and parallel capacitance will distort the signals as trace length increases. The voltages received may resemble Figure 1a, rather than the initial signal shown in Figure 1b.

Figure 1. Transmission Line Capacitance and Inductance Distorted Signals.



A Schmitt trigger circuit will tend to filter noisy inputs, providing signals with clean edges and high noise immunity. A Schmitt trigger output remains logically low until Vin reaches 1.73 V, the positive-going input threshold voltage. At this point, the output switches to a stable 5V. For a negative-going edge, a high Schmitt trigger output will not drop to 0V until Vin decreases to 1.32V. The voltage difference between the positive and negative input thresholds acts as a noise buffer. For input voltage fluctuations between 1.73V and 1.32V, the output remains stable. Figure 2 illustrates a Schmitt trigger hysteresis feature.

Figure 2. Voltage Transfer Characteristics for EPLD Implemented Schmitt Trigger Circuit.



EPLD SCHMITT TRIGGER CIRCUITS

EPLD based receivers or similar noise sensitive input applications may emulate Schmitt triggers. Figure 3 illustrates an EPLD macrocell configured as a Schmitt trigger. The circuit has two external resistors tied to the EPLD input and output pin. Each circuit consumes one macrocell. R1 and R2, the external resistors, have values derived from the equations shown below.

The calculations for this type of circuit yield worst case values for HIGH and LOW level input current of approximately 40 μA and 20 μA respectively. These values fall well within the range of standard TTL Schmitt trigger devices such as the 74LS14. By using this "built-in" Schmitt circuit, the EPLD user may take advantage of the benefits of Schmitt trigger circuitry without the overhead of an additional 7400 series package.

Figure 3. Schmitt Trigger Logic Within EPLD.

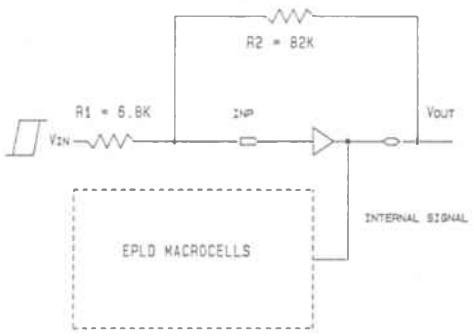
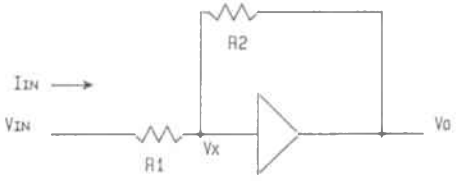


Figure 4. Circuit Model for Calculations.



CALCULATIONS

Using the circuit model shown in Figure 4:

Assume R1, R2 >> 1K
i.e., no output loading

Assume R1, R2 << Zin
i.e., no input loading

In general,

1) (Vin - Vx) / R1 = (Vx - Vo) / R2

and for the positive-going input threshold,

2a) (Vin(+) - Vth) / R1 = (Vx - 0) / R2

Simplifying,

2b) Vin = ((R1 / R2) + 1) Vth

Similarly for the negative-going input threshold,

3a) (Vin(-) - Vth) / R1 = (Vx - 5) / R2

Thus, the hysteresis voltage is given by:

V = Vin(+) - Vin(-) = 5 (R1 /R2)

and for a typical Schmitt trigger hysteresis,

delta V = 400mV

delta V = 5 (R1 / R2) = 400mV

thus R2 = 12.5 R1

An EPLD's typical threshold voltage (Vth) is 1.6V.

Using R1 = 6.8K and R2 = 82K, the input currents can be determined.

Iin = (Vth - Vo) / R2

for Vo = 5v, a high level input:

Iin = 3.4V / 82K = 41μA

and for V0 = 0V, a low level input:

Iin = 1.6V / 82K = 19.5μA

therefore as shown in Figure 2,

delta V = 5 (R1 / R2) = 410mV

Vin(+) = (.08 + 1) 1.6V = 1.73V

Vin(-) = 1.73V - 410mV = 1.32V

AB19 Rev 2.0

FEATURES

- **Bistable Circuits.**
- **One-Shots.**
- **RC Oscillators.**
- **Crystal Oscillators.**

INTRODUCTION

Multivibrator circuits are often the key to many sequential logic designs. Such circuits can be broken into three groups:

1. Bistable circuit
2. Monostable or "one shot" circuit
3. Astable or oscillator circuit

This Application Brief will show how to implement these types of multivibrator circuits with Altera EPLDs.

BISTABLE CIRCUITS

The most basic bistable circuit is the latch. A wide variety of latches may be implemented in EPLDs by providing a positive feedback signal to the "trigger" of the latch. Figure 1a shows how to implement an asynchronous, level triggered D-type latch. In addition to using EPLD macrocells to design latches, users always have the option of using the built-in edge triggered flip-flops within each macrocell. Figure 1b shows the different flip-flop options available within Altera EPLDs.

Figure 1a. D-Type Latch

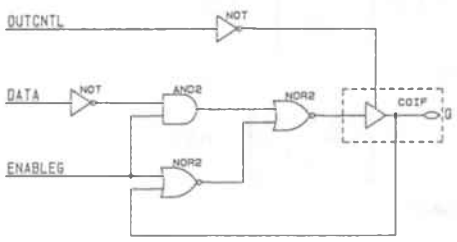
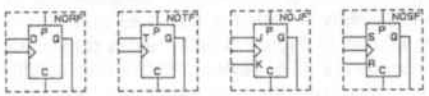


Figure 1b. Programmable Flip-Flops



MONOSTABLE CIRCUITS...

"ONE SHOTS"

A simple implementation of a monostable multivibrator with an EPLD (in this case the EP900) is shown in Figure 2. By using one macrocell, it is possible to construct the 74121-type one shot. One input pin is used for the triggering pulse, another input pin is used to provide the "regenerative path." A combinatorial output is taken and tied to the external RC circuit. Diodes should be used to "clamp" the output pulse voltage to control the recovery time for the one shot output pulse.

In some cases, the user may choose to substitute an OR gate in place of the NOR gate in Figure 2. This OR configuration enables the one shot on a falling edge of the triggering input pulse.

The output waveform, Vx, is a pulse whose duration is determined primarily by R and C and the positive-going switching threshold voltage, Vth. The calculation which follows may be used as a guideline in determining the period of your monostable output pulse:

Following the initial transition, the voltage Vx is of the form:

$$Vx(t) = A + B\, e^{(-t/T)} \text{ where } T = RC$$

but

$$Vx(0) = 0 = A + B$$

and

$$Vx(t) = VCC = A \quad \text{as t approaches infinity.}$$

Therefore,

$$B = -VCC$$

which yields

$$Vx(t) = VCC[1 - e^{(-t/T)}].$$

We want
$Vx(t1) = Vth$ where t1 = duration of one shot pulse.

Solving,

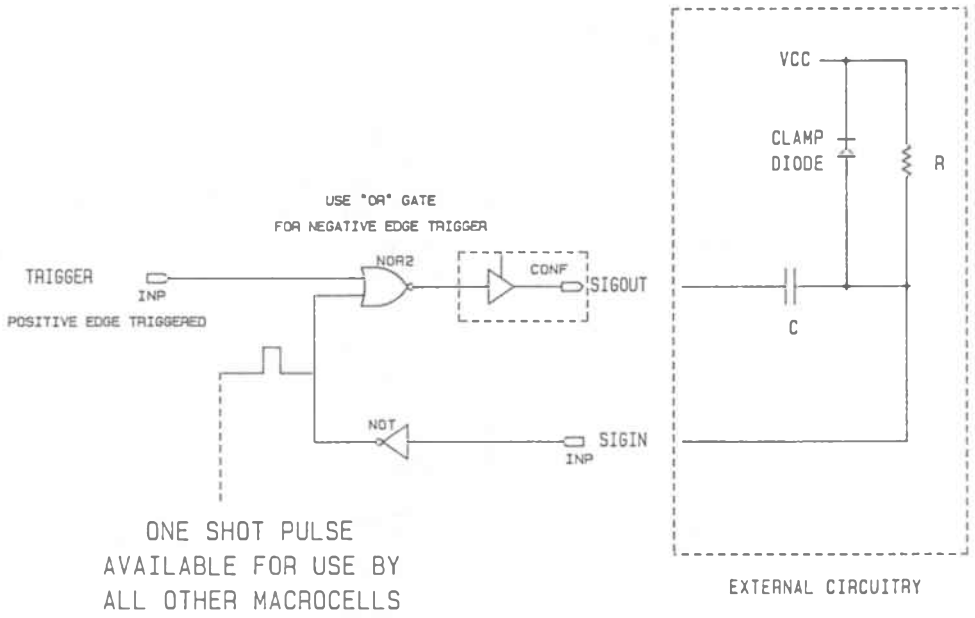
$$t1 = Tln\left[\frac{VCC}{VCC - Vth}\right]$$

Hence, with Vth = 1.60v

$$t1 = 0.39T = 0.39RC$$

Thus, for a one shot pulse of t1 = 1ms, the user should choose values of R = 27 Kohms and C = 1 μF.

Figure 2. The One Shot



ASTABLE CIRCUITS... "OSCILLATORS"

Astable circuits are not recommended design practice with EPLDs. Nevertheless, Figure 3 shows a basic implementation of an astable multivibrator using an Altera EP900. By utilizing two macrocells, an input pin and an external RC circuit, it is possible to construct an oscillating waveform with a predetermined frequency. A resistor and a capacitor are tied externally to each of the outputs, then brought to a common node ultimately feeding back to the original input pin.

The following calculations can be made to determine the approximate frequency of the output waveform:

$Vx(t) = A + Be^{(-t/T)}$ where $T = RC$.

For (t0 → t1):

$Vx(t1) = Vth = (Vth + VCC)\, e^{(-t1/T)}$

Thus,

$$t1 = T \ln \left[\frac{Vth + VCC}{Vth} \right]$$

Similarly, for (t1 → t2):

$Vx(t2) = Vth = VCC + (Vth - 2VCC)\, e^{(-t2/T)}$

Thus,

$$t2 = T \ln \left[\frac{(Vth - 2VCC)}{Vth - VCC)} \right]$$

Using Vth = 1.60v,

$$t1 = T \ln \left[\frac{1.6 + 5}{1.6} \right] = 1.3T$$

and

$$t2 = T \ln \left[\frac{1.6 + 2(5)}{1.6 - 5} \right] = 0.9T$$

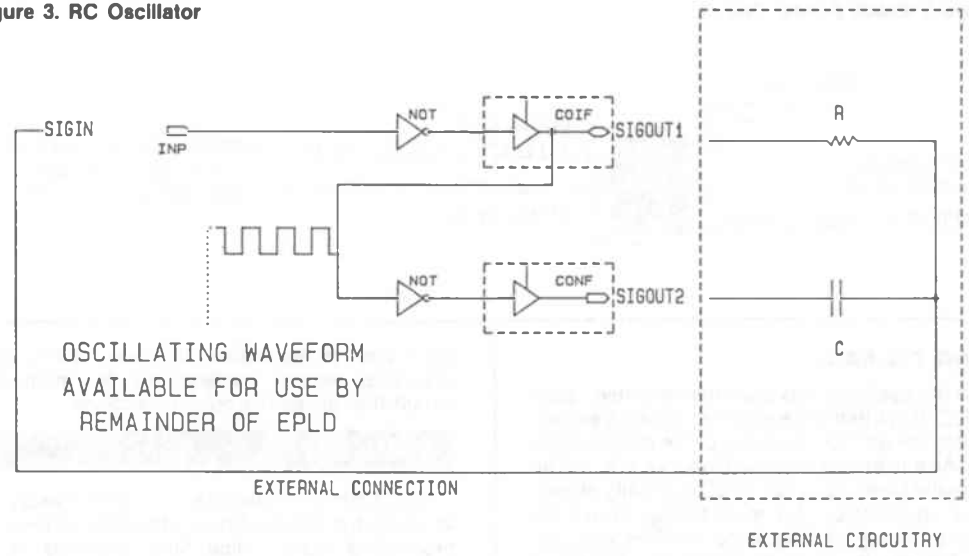
Finally,

$$f = \frac{1}{t1 + t2} = \frac{1}{2.2\ RC}$$

where f is the frequency of the oscillator.

The oscillator duty cycle is dependent upon the input threshold of the device. Because the Vth is approximately = 1.60v for Altera EPLDs, the oscillator shown in Figure 3 will yield a duty cycle of approximately 25%. Note, constructing oscillators is very differcult since the exact thershold of each transistor may vary.

Figure 3. RC Oscillator



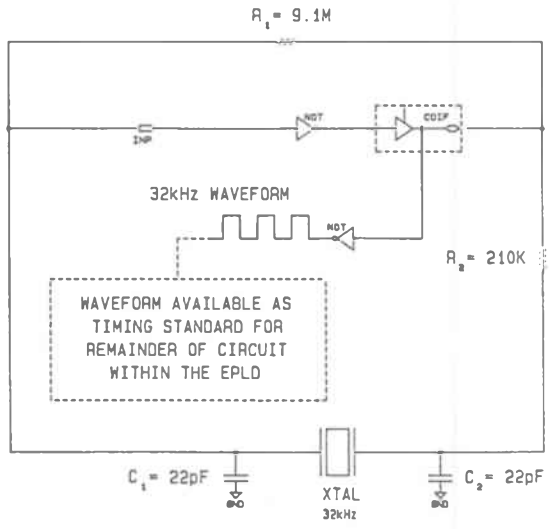
USING CRYSTALS

For more precise timing circuitry, the designer has the option of using a quartz crystal in conjunction with the EPLD. The circuit shown in Figure 4 illustrates the required external components for the EP900. Quartz crystals of varying frequencies may be used, limited only by the speed grade of the EPLD. The resistor and capacitor values shown in Figure 4 have been chosen for operation of a 32kHz quartz crystal. For crystals that have frequencies in the MHz range, the following values may be used:

R1 = 82 Kohms R2 = 24 Kohms C1 = C2 = 22 pF

"Best" values for these components are strongly dependent on the application and its requirements (for example, temperature, frequency tolerance, etc). It is strongly suggested all design recommendations for the EPLD be followed, such as grounding unused pins and using decoupling capacitors.

Figure 4. Crystal Oscillator



TIMING COMPONENT VALUES
FOR HIGHER FREQUENCY OPERATION

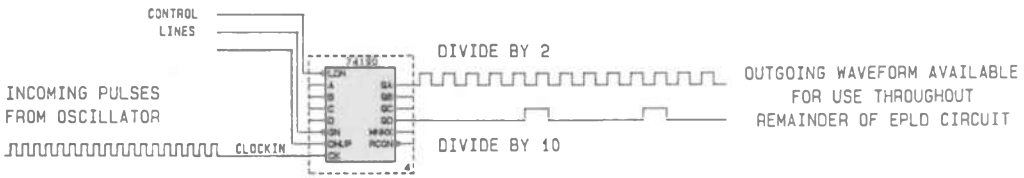
Use the following values for crystal frequencies in the MHz range (e.g., 3.58MHz):

R_1 = 82K

R_2 = 24K

$C_1 = C_2$ = 22pF

Figure 5. Clock Divider Circuit



VARYING THE RATE

Once the oscillator has been implemented upon the EPLD, it is often necessary to choose a subset frequency for different sections of the overall logic circuit. As a result, frequency dividers are found in most digital timing circuits. EPLD flexibility allows the user to construct his main timing circuit, as well as a variety of frequency dividing circuits. Figure 5 shows the implementation of a divide by 2 and a divide by 10 circuit within the EP900. Such circuits can be implemented using a small number of macrocells to create several different clock lines. With the addition of one more macrocell, an N to 1 multiplexer could be used to select any one of the clock frequencies. Even after these functions have been implemented, the bulk of the EPLD resources remain available for the main logic circuit that drives the output functions.

SUMMARY

Most timing circuits consist of three basic parts: an oscillator, or main timing standard; some digital processing logic, which often amounts to frequency dividing circuitry; and main logic circuitry which drives the output functions. In the past it took several different IC packages, each with specified functions, to implement such a timing circuit. With the availability of high density EPLDs, logic designers may now integrate "the timer, the divider and the function" within a single chip.

AB15 Rev 3.0

FEATURES

- **What is Dual-Feedback.**
- **How to use Dual-Feedback both automatically and manually.**

INTRODUCTION

The EP1800/EP1810, EPB1400, and EP512, as well as all MAX Family devices, offer dual-feedback macrocells. Dual feedback means there are two feedback paths for a single macrocell. One comes from the register or combinatorial network (internal), and the other comes directly from the pin (external). Thus, a buried register can use one feedback path while the I/O pin can simultaneously use the second path.

Figure 1 compares a dual-feedback macrocell with a standard one. Notice that the standard macrocell provides a single feedback into the logic array from either internal logic or directly from the pin, but not both. If the register in a standard macrocell is not used as an output (i.e. buried), then the internal feedback path will be selected and the pin will be unused. Likewise, if the pin is used as an input, the external feedback will be selected and the register will be unused.

A dual-feedback macrocell, on the other hand, provides two feedback paths: one from either side of the tri-state buffer. When the tri-state buffer is enabled, the two feedback paths are logically connected and the macrocell behaves much like a standard macrocell. By disabling the tri-state buffer, however, the internal feedback becomes isolated from the external feedback. Thus, a buried register and an input can be placed in the same macrocell. The register uses the internal feedback, and the input uses the external feedback.

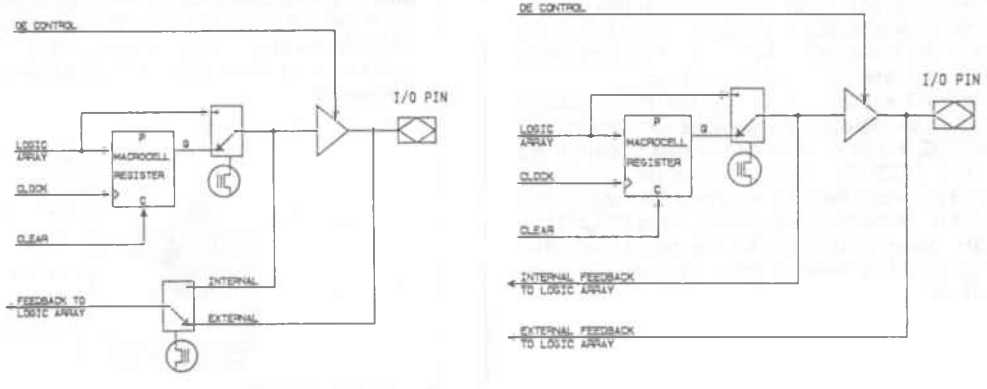
ACCESSING DUAL-FEEDBACK

The A+PLUS software automatically implements dual-feedback by placing buried logic in one of the dual-feedback macrocells whenever it sees the need. If the design uses buried registers, and needs more dedicated inputs than available on the EPLD, the software will place the buried register in a dual-feedback macrocell and use the I/O pin as an input.

If the need arises, however, manual placement of the buried registers provides complete control over the fitting process. The convention for manual placement is to assign the Q node of a buried register to a pin number. Pin assignment is accomplished by appending the node name with the "@" symbol and entering the pin number exactly as done with input and output pins.

Figure 2 shows how manual placement is accomplished with the LogiCaps schematic capture package. The Q node of the No Output Register Feedback (NORF) primitive is given the nodename CAT and assigned to pin 10. The input signal DOG is placed in an INP primitive and also assigned to pin 10.

Figure 1. Standard and Dual Feedback Macrocells—*The standard macrocell provides a single feedback into the logic array. The dual-feedback macrocell provides two feedback paths into the logic array. One comes from the internal path, and the other comes from the external path.*

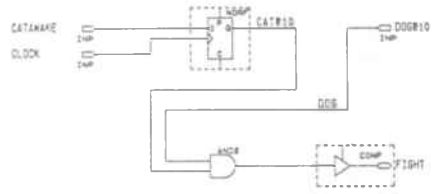


A+PLUS accepts the duplicate pin assignments because it recognizes that pin 10 on the EP1810 corresponds to a dual-feedback macrocell. It then isolates the internal feedback from the external feedback by disabling the tri-state buffer between them so that the buried register uses the internal feedback and the input uses the external feedback.

With Boolean equation entry, the same pin assignments are accomplished by placing the following lines in the .ADF file.

```
INPUTS: DOG@10 ...
OUTPUTS: CAT@10 ...
NETWORK:
DOG = INP(DOG)
CAT = NORF(CATAWAKE,CLOCK,GND,GND)
.
.
```

Figure 2. Manual Placement of Buried Registers—*Buried registers can be assigned to a macrocell by using the "@" sign and the corresponding pin number. In this case the register (CAT) and the input symbol (DOG) will be placed in the same dual-feedback macrocell.*



BI-DIRECTIONAL DUAL-FEEDBACK

There are a few applications where dual-feedback is required on a bi-directional pin. These cases require dynamic control over the tri-state buffer. The MacroFunctions called RO2F, CO2F, TO2F, JO2F, and SO2F, provide complete control of the dual-feedback macrocells. The "2F" at the end of the name refers to the fact that both feedback paths are shown on the symbol.

Figure 3 shows how to use the RO2F MacroFunction to approach the previous application. The pin DOG is a bi-directional pin that is controlled by the OECNTL input. In normal operation, OECNTL would be LOW and the 2 feedback paths would be isolated as in Figure 2. If the OECNTL is HIGH, however, then the bi-directional pin called DOG, would be driven with the value on the register (CAT).

When using the Boolean equation design entry technique, the following lines emulate the circuit shown in Figure 3.

```
INPUTS: DOG@10 ...
OUTPUTS: DOG@10 ...
NETWORK:
DOG = INP(DOG)
DOG, CAT = RORF(CATAWAKE,
CLOCK,GND,GND,OECNTL)
.
.
```

Figure 3. Bi-Directional Dual-Feedback—*When dual feedback is required on a bidirectional pin, the xO2F macrofunctions (RO2F, TO2F, JO2F, SO2F, and CO2F) provide access to both feedback paths and the output enable.*

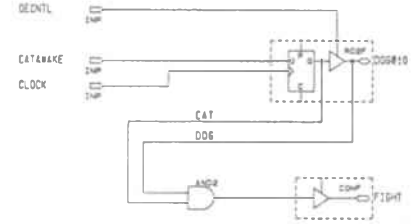
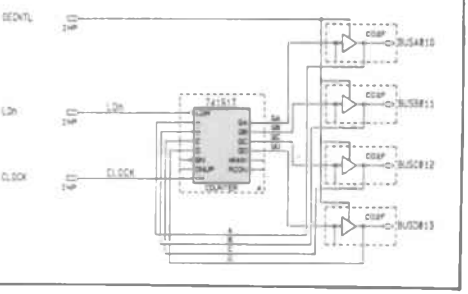


Figure 4 shows a more advanced example of bi-direction dual-feedback using MacroFunctions. The design required that a 4-bit counter be connected to a bi-directional bus. The bus must be able to read the current counter value as well as to specify a new value to load into the counter.

The design was completed by connecting CO2F MacroFunctions to the outputs of a 74191 MacroFunction. The OECNT input controls the direction of the bus. If OECNTL is high, the counter value is driven onto the bus. If OECNTL is low, the value on the bus is applied to the parallel load inputs of the counter.

Figure 4. Bi-Directional Dual-Feedback (with MacroFunction)—*Bi-directional dual-feedback can also be accessed by attaching a CO2F to the output of a MacroFunction. This design consumes just 4 dual-feedback macrocells.*



AB55 Rev 1.0

FEATURES

- **Direct replacement for all commonly used 20 pin PALs.**
- **User-Configurable I/O architecture.**
- **Zero-Power option (150 μA).**

INTRODUCTION

The Altera EP320 can directly replace all functions that may be implemented with the 20 pin PAL family. The EP320 architecture is user-configurable, allowing it to be functional as well as pin-to-pin compatible with PALs. Manufactured with a CMOS EPROM technology, the EP320 eliminates the high power requirements demanded by conventional fuse-programmable bipolar PALs. At standby, the device only consumes 150μA. In addition the EP320 offers erasability, making it reprogrammable. By offering this greater flexibility, a single EP320 can replace many logic device types that would otherwise need to be purchased and inventoried.

EP320 FUNCTIONAL OVERVIEW

The EP320, like PALs, implements sum of products logic using a programmable AND, fixed OR logic array. The device provides 10 dedicated inputs and eight I/O pins. Each I/O can be independently configured for input, output, or bi-directional operation, see EP320 Data Sheet for a complete description.

Internally, the EP320 is divided into eight macrocells. Each macrocell can sum (logically OR) eight product terms. The result is directed to an I/O Architecture Control block which produces either combinatorial or registered outputs, (active high or low), and a feedback path to the AND array. Each macrocell contains a ninth product term connected to the I/O pin tri-state buffer. The AND array contains 36 input lines which are generated from the true and complement signals of the ten input pins and eight feedback paths. Thus, each product term is equilvalent to a 36 input AND gate.

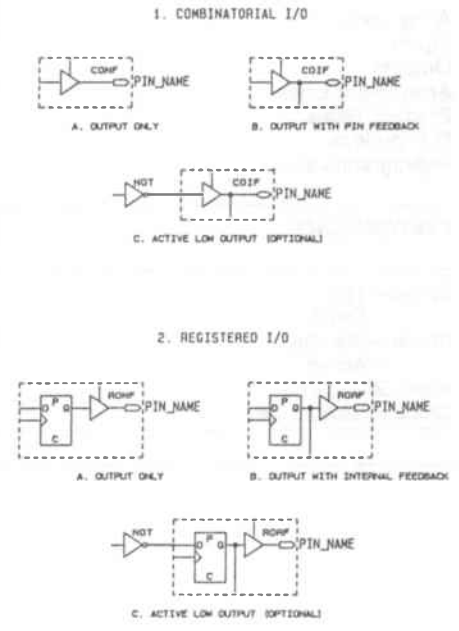
Clocking internal registers is accomplished through pin number one. The true signal is connected to all internal registers. The EP320 flip-flops are positive edge triggered, meaning data transitions occur on the rising edge of the clock signal. If no clocking is required, pin one may be used as an additional input to the AND array. All registers perform automatic reset (outputs go to logical zero) on chip power up.

PAL COMPATIBILITY

Unlike PALs which have fixed I/O architectures, every macrocell of the EP320 contains a user-configurable I/O selection. Each output can be configured for combinatorial (directly from OR gate) or registered (output through D-flipflop) operation, and programmed either active high or low. Figure 1 shows the output modes which can be configured.

Table 2 gives detailed listing for proper I/O configurations to replace many of the commonly used 20 pin PALs. Table 1 compares device specifications.

Figure 1. EP320 I/O Options



OUTPUT ENABLE

Every output of the EP320 has an available three-state buffer, controlled by a dedicated product term. When the product term is asserted HIGH, the output will be enabled. Output enable logic is implemented directly from the AND array. Therefore it can be programmed active high or low or be conditionally asserted from any of the selected inputs and feedback paths.

For combinatorial outputs, the EP320 and PAL have similar 3-state output implementation. The only difference is the PAL uses one of its eight product terms to control the buffer. The result is only seven product terms for logic. The EP320 provides an additional product term, thus leaving its eight input OR gate intact.

Registered PALs have their Output Enable function hard wired to pin 11 allowing active low operation. If desired, the EP320 can remain exactly compatable by connecting the complement of pin 11 to the Output Enable product term.

DESIGN TOOLS

A+PLUS as well as Altera Utility Programs called ALTERANS][and PAL2EPLD may be used to construct both Boolean design files (ADF file) and JEDEC files for EP320's. When using A+PLUS, a text editor (in non-document mode) is used to create the ADF file (see Boolean Equation section). The Utility Program section contains information on ALTERANS][and PAL2EPLD. LogiCaps may also be used to allow equation and TTL level schematic capture logic input.

AN2 Rev 2.0

TABLE 1. FEATURES AND PERFORMANCE COMPARISION

	EPLD	PAL	
FEATURES	**EP320**	**16L8**	**16R8**
Array Logic	AND-OR	AND-OR	AND-OR
Inputs	17	16	10
Outputs	8	8	8
Array Input Lines	36	32	32
Product Terms	72	64	64
D Flip-Flops	8	NA	8
Reprogrammable	YES	NO	NO
PERFORMANCE	**EP320-1**	**16L8B-4 (Quarter Power)**	**16R8A (High Speed)**
Speed—Tpd	30 ns	35 ns	NA
—Fmax	28 MHz	NA	28 MHz
Power—Standby	150 μA	55 mA	180 mA
—Active	5 mA	55 mA	180 mA
Input Set-up time	20 ns	NA	25 ns
Clock to Output Delay	18 ns	NA	15 ns

TABLE 2. EP320 CONFIGURATIONS FOR 20 PIN PAL REPLACEMENT

Pal Part Number	EP320 Pin Number	EP320 Macrocell Number	Output/ Polarity	Feedback
10H8	12-19	1-8	Comb/High	None
10L8	12-19	1-8	Comb/Low	None
12H6	12 13-18 19	8 2-7 1	None Comb/High None	Pin None Pin
12L6	12 13-18 19	8 2-7 1	None Comb/Low None	Pin None Pin
14H4	12-13 14-17 18-19	7-8 3-6 1-2	None Comb/High None	Pin None Pin
14L4	12-13 14-17 18-19	7-8 3-6 1-2	None Comb/Low None	Pin None Pin
16C1	12-14 15 16 17-19	6-8 5 4 1-3	None Comb/Low Comb/High None	Pin None None Pin
16H2	12-14 15-16 17-19	6-8 4-5 1-3	None Comb/High None	Pin None Pin
16L2	12-14 15-16 17-19	6-8 4-5 1-3	None Comb/Low None	Pin None Pin
16H8 & 16HD8	12 13-18 19	8 2-7 1	Comb/High/Z Comb/High/Z Comb/High/Z	None Comb None
16L8 & 16LD8	12 13-18 19	8 2-7 1	Comb/Low/Z Comb/Low/Z Comb/Low/Z	None Comb None
16R4	12-13 14-17 18-19	7-8 3-6 1-2	Comb/Low/Z Reg/Low/Z Comb/Low/Z	Comb Reg Comb
16R6	12 13-18 19	8 2-7 1	Comb/Low/Z Reg/Low/Z Comb/Low/Z	Comb Reg Comb
16R8	12-19	1-8	Reg/Low/Z	Reg
16P8	12 13-18 19	8 2-7 1	Comb/Option/Z Comb/Option/Z Comb/Option/Z	None Comb None
16RP4	12-13 14-17 18-19	7-8 3-6 1-2	Comb/Option/Z Reg/Option/Z Comb/Option/Z	Comb Reg Comb
16RP6	12 13-18 19	8 2-7 1	Comb/Option/Z Reg/Option/Z Comb/Option/Z	Comb Reg Comb
16RP8	12-19	1-8	Reg/Option/Z	Reg

FEATURES

- **Types of Macrocells.**
- **Local and Global Bus Structures.**
- **Programmable Clock Options.**

INTRODUCTION

Each EP1800/EP1810 macrocell can be individually configured for general purpose logic, clocking options, and I/O architectures. This Application Brief discusses the type of macrocells contained in the EP1800/EP1810, inter-macrocell communication via Local and Global Bus structures, and available clock options for each internal flip-flop. Use the EP1800/EP1810 Data Sheet for additional detail during the following discussion.

MACROCELLS

Each EP1800/EP1810 contains a total of 48 macrocells. The device is partitioned into 4 quadrants. Each quadrant contains 12 macrocells. Within each quadrant there are 3 kinds of macrocells: General, Enhanced, and Global, (see Figure 1). Each shares common architectural features:

- 8 product terms for logic
- Active high or low polarity control
- Registered or combinatorial operation
- Programmable flip-flops (D, T, JK, SR)
- 1 product term for flip-flip Asynchronous Clear
- 1 product term for Programmable Clock/OE control

Slight differences exist between macrocells in terms of (1) feedback options and (2) array delays, (tlad):

General Macrocells provide a single feedback path to the logic array. The feedback is multiplexed, (Figure 1), coming from macrocell internal logic or from the I/O pin. Independent of the selection, the feedback path only drives the Local Bus, (refer to Macrocell-Bus Interface).

Enhanced Macrocells are similar to General Macrocells, except they have faster array delays. This was made possible during layout of chip topology. The array delay for Enhanced macrocells is listed under the "tlade" AC specification in the EP1800/EP1810 data sheets.

Global Macrocells have the same array delays as General macrocells (tlad), but provide dual feedback. The I/O architecture contains two independent feedback paths. The first feedback path is internal, derived from macrocell combinatorial or registered logic. The second feedback path comes directly from the I/O pin. By providing two independent paths, Global macrocells may be used for buried logic and at the same time serve as dedicated input pin. Note, the internal feedback path drives the Local Bus, while the I/O feedback path drives the Global bus.

During fitting, if the macrocell requires global feedback (i.e. must feed macrocells located in other quadrants), A+PLUS automatically places and promotes feedback signals to the I/O feedback path, hence, to the Global Bus. If the I/O feedback line is used by macrocell internal logic for global routing, the I/O pin must be reserved for this function and can not be used as an addition input.

MACROCELL-BUS INTERFACE

Input and feedback signals are connected to each EP1800/EP1810 macrocell via a Local and Global Bus. Figure 2 shows the Macrocell-Bus interface for Quadrant D.

Within each macrocell, product terms share the entire bus structure. Therefore, each product term may produce a logical AND of any of the array inputs (or their complements) present on either bus.

All quadrants share the same Global Bus. The Global Bus contains 64 input signals. Input signals come from the true and complement of the 12 dedicated data input pins, 4 clock input pins, and the 16 Global macrocell pin feedback paths.

Each quadrant has its own Local Bus which contains 24 input signals. Inputs to the Local Bus come from the feedback (true and complement signals) of the 12 quadrant macrocells. For the 4 General Macrocells and 4 Enhanced Macrocells, feedback signals come from macrocell internal logic or from the I/O pin. For the 4 Global Macrocells, feedback signals only come from macrocell internal logic (Local Bus feedback path). Table 1 summarizes Macrocell-Bus interconnect.

PROGRAMMABLE CLOCK

Each EP1800/EP1810 macrocell may be clocked synchronously or asynchronosly. Synchronous clocks (also referred to as system clocks) come directly from the dedicated clock pins. The EP1800/EP1810 has 4 dedicate clock pins, one for each quadrant. When using synchronous clocks, the flip-flops are positive edge triggered. Asynchronous clocks are generated from the OE/CLK product-term, see Figure 1 . Thus asynchronous clocks can be a dedicated input pin, I/O pin, or internal logic (AND gates, NOR gates, or feedback signals). Since the product-term contains both true and complement signals, asynchronous clocks

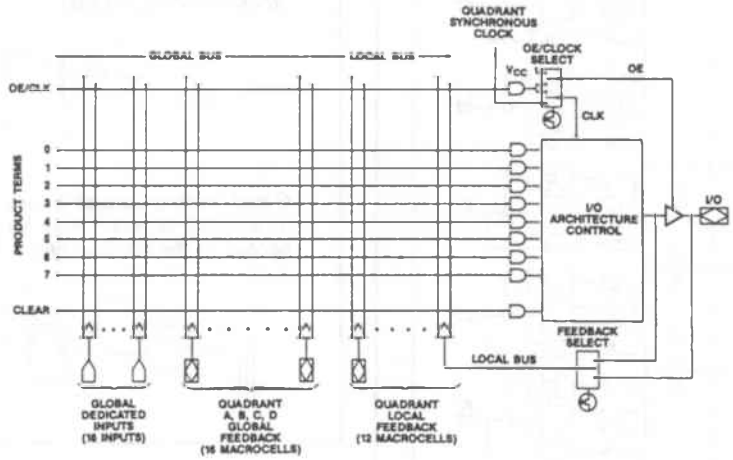
can be programmed for positive or negative edge triggered operation.

Each macrocell clock may be individually configured for either synchronous or asynchronous clocking. When designing with A+PLUS, synchronous clocking is configured by connecting an Input Pin primitive (INP) directly to the clock input of the flip-flop as shown in Figure 3a. Synchronous clocks provide the advantage of fast clock to output delay times.

Asynchronous clocks are configured by the use of a CLKB primitive or by driving the flip-flop clock input with logic. Examples of asynchronous clocking is shown in Figures 3b and 3c. When using gated clock structures whose minimized logic requires more than a single product term (AND gate), an NOCF primitive must be inserted between the clock logic and the flip-flop.

For example, the logic in Figure 4a requires more than one product term after minimization. The A+PLUS design processor gives a fitter error "breaks one p-term limit". By placing an NOCF primitive in the circuit as shown in Figure 4b the design can be implemented into the EPLD architecture.

Figure 1. General and Enhanced Macrocell



Global Macrocell

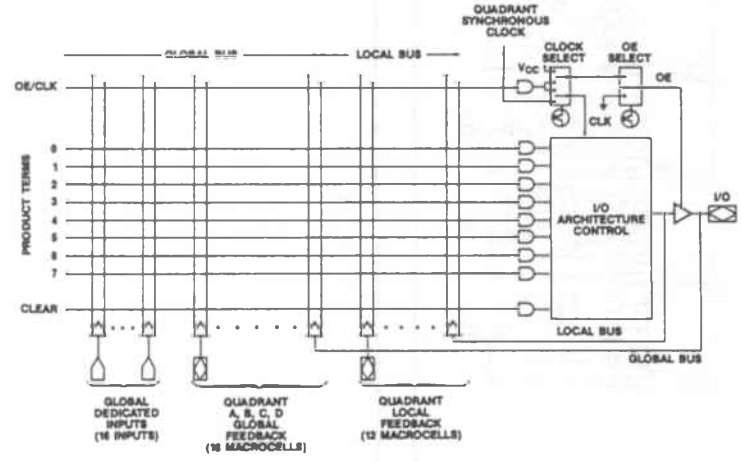


Figure 2. Macrocell-Bus' Structure

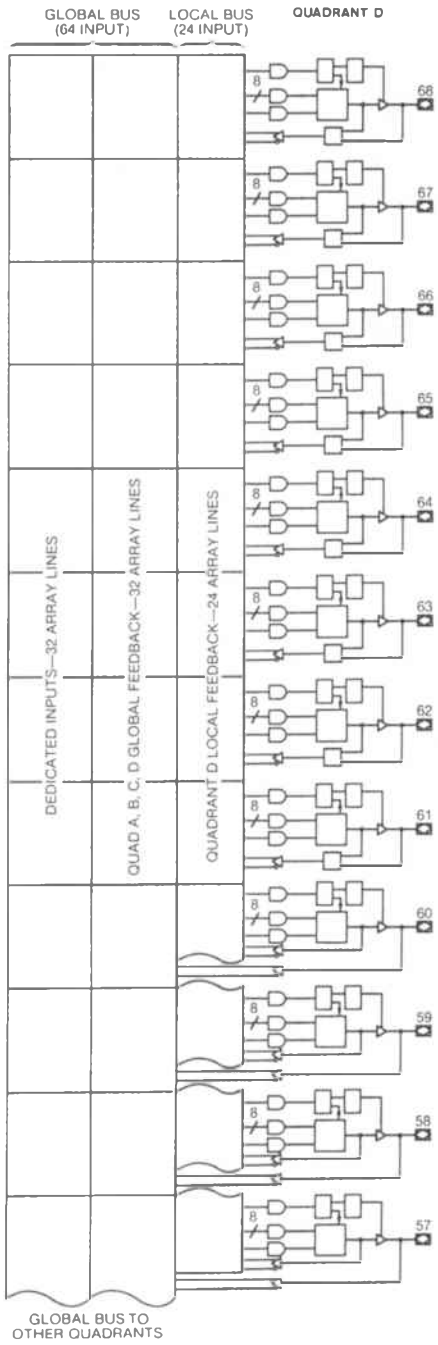
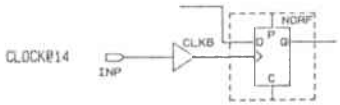


Figure 3. Configuring Clock Options

a. Synchrononus clocks are configured by connecting any of the dedicated clock pins directly to the flip-flop clock input.

b. Asynchronous clocks from the input or I/O pins are implemented with the use of a Clock Primitive (CLKB).



c. Gated clock structures whose logic reduces to an AND function may be directly connected to the flip-flop clock input.

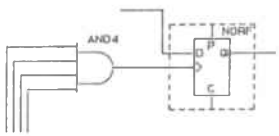
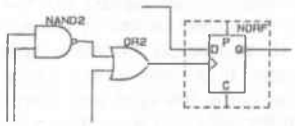
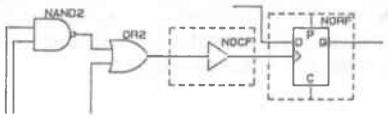


Figure 4. Gated Clock Implementation

a. The clock logic requires more than a single product term. A+PLUS will issue an error message "breaks 1 p-term limit".



b. Inserting an NOCF primitive between the clock logic and flip-flop will place the logic into a different EP1800/EP1810 macrocell allowing A+PLUS to successfully fit the design.



Note, the NOCF primitive requires an additional macrocell be used for the clock logic. The result is then fed-back to the macrocell containing the flip-flop.

The OE/CLK product term may be connected to either the flipflop clock input or the macrocell tri-state buffer, but not both. Therefore, when asynchronously clocking flip-flops, the macrocell tri-state buffer can not be controlled by logic. It must be directly tied to either VCC or GND.

The OE/CLK product term is a 88 input AND gate. It does not have programmable inversion capability. Thus, if the clock logic requires inversion (such as a NAND or OR function), A+PLUS will issue the error message "illegal inverion on clk". If this results, the NOCF must be inserted as shown in Figure 4b.

CONCLUSION

By keeping the considerations outlined within the Application Brief in mind, EP1800/EP1810 users gain a more thorough understanding of the device. As a result, potential applications can be evaluated more efficiently and the EPLD can be utilized to its maximum capability.

AB61 Rev 1.0

TABLE 1. MACROCELL-BUS INTERCONNECT

	Pin #	Macrocell #	Macrocell Type	Feedback Interconnect
Quad A	2-5	1-4	Enhanced	Quad A
	6-9	5-8	General	Quad A
	10-13	9-12	Global	Quad A,B,C,D
Quad B	23-26	13-16	Global	Quad A,B,C,D
	27-30	17-20	General	Quad B
	31-34	21-24	Enhanced	Quad B
Quad C	36-39	25-28	Enhanced	Quad C
	40-43	29-32	General	Quad C
	44-47	33-36	Global	Quad A,B,C,D
Quad D	57-60	37-40	Global	Quad A,B,C,D
	61-64	41-44	General	Quad D
	65-68	45-48	Enhanced	Quad D

FEATURES

- **Description of a generic Barcode.**
- **Description of a Bar Code Decoder EPLD.**
- **State machine implements controller functions.**
- **Functional simulation verifies design before commitment to silicon.**
- **TTL MacroFunctions simplify and speed up the design task.**

INTRODUCTION

The following Applications Brief describes a bar code decoder implemented in an EP1810. The EP1810 decodes a generic bar code, stores the decoded data byte, and alerts a microprocessor that data is ready. This Application Brief describes a generic bar code decoder, the various design methodologies used to implement the design, and the functional simulation used to verify the design before programming devices. The final design is specified by a mix of design entry formats: state machine design entry is used to specify an internal controller, while schematic capture and TTL macrofunctions are used to define additional functions.

What is Bar code?

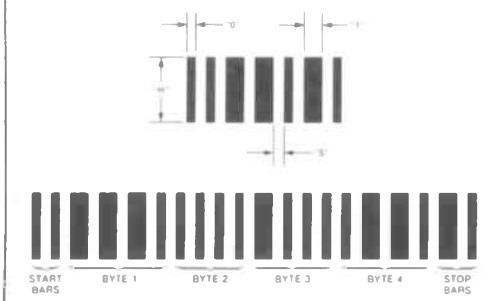
Bar code, a means of representing binary data or program information, has become popular due to its great flexibility and cost effectiveness. For many applications bar code yields superior results when compared to optical character recognition, particularly for success on a first time read. Bar code is suitable in many applications where magnetic stripe or other media would be impractical. Bar code has several variations; this Application Brief covers a common version with some advanced features.

Physical Specifications of Bar Code

Although many versions of bar code exist to support the variety of applications served, there is enough in common to treat a meaningful generic case (Figure 1). All bar codes have "zero", "one", and space characters; the "zero" and space character are the same width, while the "one" is twice that width. All bar codes have a header and a tail. The generic bar code has a header consisting of a zero-zero sequence, followed by a checksum byte. The tail is a one-zero sequence. Data follows the header and terminates with the tail. All bar codes have maximum limits on the number of data bytes.

One requirement for accurate bar code detection is that the reader, usually a light pen, scan the bar code at a relatively constant speed. For this reason bar codes are of modest width; its easier to move a light pen at reasonably constant speed over shorter distances than over longer ones. Before reading the bar code, the light pen output is low, indicating that the light pen is inactive or in a white region. As the light pen reaches the dark region of the header, the light pen output goes high. If the light pen's speed varies by too much, then the mis-read should be detected by verification against the checksum.

Figure 1. Sample Barcode—*A generic bar-code has "0", "1", and space characters. Bar code sequences consist of START and STOP bars, data and checksum bytes.*



Bar Code Decoder Overview

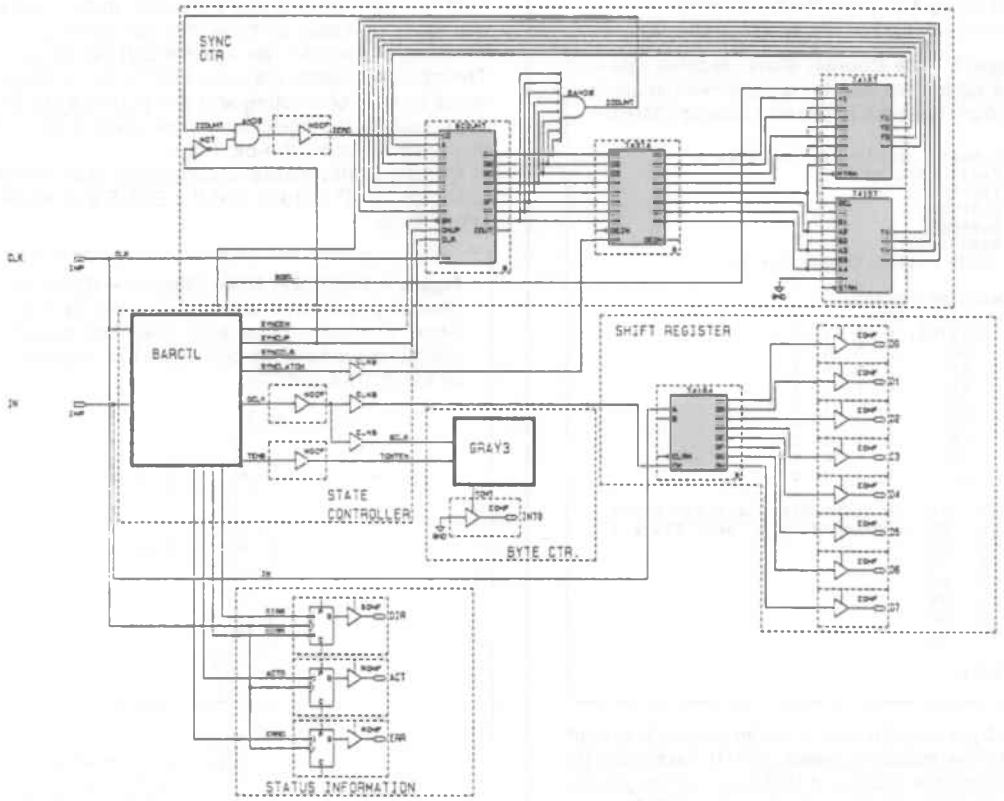
Figure 2 shows a bar code decoder implemented in an EP1810. The bar code data is fed into the EP1810 through the IN input. The data is used by the sync counter to determine the input data read rate, and by the state machine to decode incoming data, which is stored in a shift register. Once 8 bits of data have been shifted into the shift register, an open collector interrupt back to the microprocessor (INT0) goes low. Various status outputs (DIR, ACT, and ERR) indicate the current state of the bar code decoder.

The Bar Code Decoder consists of 5 different modules: a sync counter, a controller state machine, a byte counter, a shift register, and a microprocessor interface.

Sync Counter

The bar code decoder uses the bar code header's leading single width dark region to measure the fundamental width of a "zero". When the light pen passes over the first dark region, the sync

Figure 2. Bar Code Decoder Schematic—*A barcode decoder implemented in an EP1810 consists of Sync Counter, State Controller, Byte Counter, Status Outputs, and Shift Register sections.*



counter begins counting. The sync counter stops counting only when the light pen has completed reading the dark region (ie. when it reads the beginning of light region). The count value thus reflects the amount of time required to read the width of a space or "zero" bar. It takes twice the count value to read a "one" bar.

The count value is used to sample the bar code data. Since the light pen scanning speed will vary somewhat over the bar code label, the count value is only approximately correct at measuring width. For this reason it is best not to sample data on the edges of the light and dark regions, but rather is the center of these regions. The first sampling occurs in the middle of the space width following the leading dark region when the sync counter reaches half of its sync value. The counter is reset, and subsequent samples occur when the sync counter reaches the full sync value.

The values of the data samples correspond to the bar encoded data. If "dark-light" is read, then the light pen has passed over a single width dark region followed by a single width light region, indicating a "zero". If a "dark-dark-light" is read, then the light pen has passed over two adjacent dark regions followed by a light region, indicating a "one". If any other combinations starting with "dark" are read, or if two adjacent "light"s are read, then the code is invalid.

The sync counter is implemented by four Macro-Functions (8COUNT, 2 of 74157s, and the 74374) and terminal count circuitry comprised of logic and an NOCF primitive.

Byte counter specification

The byte counter counts the number of bits shifted in, and if they are a multiple of 8, alerts the microprocessor that a full byte is ready to be read. The byte counter is implemented using a Gray

code sequence, a sequence which only has one bit change between any two count transitions, so that its outputs will be glitch-free; preventing spurious outputs from inadvertantly interrupting the microprocessor. Figure 3 shows the byte counter implemented using the state machine format.

Figure 3. Byte Counter State Machine File— *An 8 stage gray counter is implemented using the high level state machine format (SMF).*

```
% GRAY3  3 BIT GRAY COUNTER %
PART: EP1810J
INPUTS:
OUTPUTS:
NETWORK:
EQUATIONS:
 TCNT = TCNTEN*/G2*/G1*/G0;

MACHINE: GRAY3
  CLOCK: GCLK
  STATES: [ G2 G1 G0 ]
    S0    [  0  0  0 ]
    S1    [  0  0  1 ]
    S2    [  0  1  1 ]
    S3    [  0  1  0 ]
    S4    [  1  1  0 ]
    S5    [  1  1  1 ]
    S6    [  1  0  1 ]
    S7    [  1  0  0 ]

S0:  S1   %STATES MAKE UNCONDITIONAL%
S1:  S2   %TRANSITION TO NEXT STATE %
S2:  S3
S3:  S4
S4:  S5
S5:  S6
S6:  S7
S7:  S0

END$
```

The byte counter has an open collector output back to the microprocessor (INTO), fashioned by connecting the input of a tristate driver to ground and selectively enabling the tristate (TCNT).

Shift register specification

Input data is stored in a 74164 shift register. The shift register clock is fed from the state controller state machine to assure that data is latched at the appropriate time. The 74164 outputs should be buffered and connected to the microprocessor bus. The shift register is implemented by a 74164 macrofunction and a CONF output primitive.

Bar Code Decoder Status Information

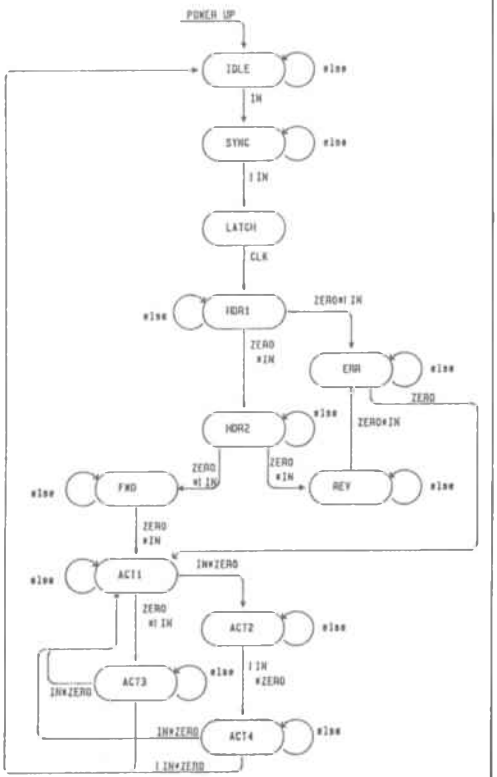
Special outputs allow external devices to determine the status of the bar code decoder. If active, the DIR signal indicates that the tail was read before the header. In this instance all data and checksum words would be reversed, and the microprocessor would have to make the compensating transformations. The ability to read bar codes backwards as well as forwards is a practical requirement, permitting bar codes to be read upside down as well as scanned from right to left.

The ACT signal indicates that a bar code is being scanned in. This is useful for a variety of error handling or initialization tasks: for example a microprocessor may poll on this signal and enter special routines dedicated to bar code reading.

The ERR signal indicates an illegal sequence of light and dark regions was encountered during the decoding process; perhaps the bar code is unreadable, or the scan rate was not uniform enough. The microprocessor may use ERR to dump illegal reads without computing and comparing a checksum against the checksum byte passed to the microprocessor by the bar code.

The status information is comprised of the open collector CONF output, and the SONF and RORF status flags.

Figure 4. Controller State Diagram— *Barcode decoding activities are coordinated by the State Controller. The state diagram below describes the behavior of the BARCTL portion of the State Controller.*



State controller specification

The bar code decoding process requires intelligence to determine if a header is valid, and to convert the light and dark bar code regions to

machine readable data. Beyond data conversion, the bar code decoder must coordinate activities of sync counter, shift register, byte counter, and status generation circuitry. The simplest means of achieving these aims is to create a centralized state

Figure 5. Controller State Machine Described Using the State Machine Format.

```
% Bar Code Controller %
PART: EP1800J
INPUTS:
OUTPUTS:
NETWORK:
EQUATIONS:
   DIRS = FWD;                    DIRN = IDLE;
   ACTD = !IDLE;                  ERRD = ERR # !IDLE;
   SYNCCLR = IDLE # /IN;          SYNCUP = SYNC;
   SYNCRN = IDLE;                 SYNCLATCH = LATCH;
   DCLK = ACT2 + ACT3;            TKND = ACT2 + ACT3 + ACT4;
   BSRL = /LATCH;

MACHINE: BARCTL
  CLOCK: CLK
  STATES: [Q3 Q2 Q1 Q0]
     IDLE  [ 0  0  0  0]
     SYNC  [ 1  0  0  1]
     LATCH [ 0  1  1  1]
     HDR1  [ 1  0  1  1]
     HDR2  [ 1  1  1  1]
     FWD   [ 0  1  0  1]
     REV   [ 0  1  1  0]
     ACT1  [ 0  1  0  0]
     ACT2  [ 1  1  0  0]
     ACT3  [ 0  0  1  0]
     ACT4  [ 0  0  1  1]
     ERR   [ 1  1  1  0]

IDLE:
        IF IN THEN SYNC
SYNC:
        IF /IN THEN LATCH
LATCH:
        HDR1
HDR1:
    IF IN#ZERO THEN HDR2
    IF /IN# ZERO THEN ERR
HDR2:
        IF IN # ZERO THEN REV
        IF /IN # ZERO THEN FWD
FWD:
        IF ZERO THEN ACT1
REV:
        IF IN # ZERO THEN ERR
        IF /IN # ZERO THEN ACT1
ACT1:
        IF IN # ZERO THEN ACT2
        IF /IN # ZERO THEN ACT3
ACT2:
        IF IN # ZERO THEN ERR
        IF /IN # ZERO THEN ACT4
ACT3:
        IF IN # ZERO THEN ACT1
        IF /IN # ZERO THEN IDLE
ACT4:
        IF IN # ZERO THEN ACT1
        IF /IN # ZERO THEN IDLE
ERR:
        IF ZERO THEN ACT1

END$
```

controller state machine. Figure 4 shows the state diagram for the state controller, implemented using the high level state machine syntax shown in Figure 5. The algorithm for Figure 4 is as follows:

IDLE—the machine idles until a dark region is read by the input device (eg. light pen) at which time the sync counter is started.

SYNC—The sync counter counts up while in this state. The machine stays in this state until reading the first light region. The sync count corresponds to the width of the first dark bar.

LATCH—The machine latches the count value into a holding register. Hereafter the sync count will expire on every modulus of the latched count, and cause the reading of the input stream.

HDR1, HDR2, FWD, REV, ERR—The machine begins reading the rest of the header bits. Bear in mind that we get two different sequences depending if we read a 0-0 sequence or a 0-1 sequence. The 0-0 sequence is a forward read, while a 0-1 sequence is a backward read. If there are any improper reads we will go to ERR. Direction status is latched dependent on either state FWD or REV. At the end of this we begin the reading of data and checksum bytes into the shift register.

ACT1, ACT2, ACT3, ACT4—These are the active reading states. The ACT1-ACT3 loop indicates a "0" was read. The ACT1-ACT2-ACT4 loop corresponds to the reading of a "1". Reading is stopped when a long white space is read indicating the end of the bar code.

Figure 6. Multiple Design Processing Prompts.

```
APLUS  ADP

FORMAT   : Adf
FILE NAME: bar barctl.smf gray3.smf
MIN      : yes
INV CTL  : no
LEF ANAL : yes
```

Implementation of the Bar Code Decoder

Figure 2 shows a LogiCaps schematic of the Bar Code Decoder. The sync counter is implemented by four MacroFunctions (8COUNT, 2 of 74157s, and the 74374) and terminal count circuitry comprised of logic and an NOCF primitive. The byte counter is implemented in state machine format in a file named GRAY3.SMF (Fig 4). The shift register is implemented with MacroFunction 74164 and CONF output primitives. The microprocessor interface consists of a CONF configured as an open collector output and status flags implemented by SONF and RORF primitives. The State controller was implemented in a state machine file named BARCTL.SMF (Fig 5). The state machines are outlined on the schematic for documentation purposes only. The borders are not required for design processing.

Design Processing

Design input is contained in three seperate files, of two different formats. LogiCaps generates the bar.adf file, whereas barctl.smf and gray3.smf are

state machine files. Linking all the design information is done in the Altera Design Processor (ADP) section of the A+PLUS development software. This is done by answering all the prompts as in Figure 6.

ADP automatically links the names between the various input files and create an output file bar.jed for device programming. Device utilization, given after processing, indicated that 38 of the 48 macrocells were used, 2 of the 7 inputs, and 36 percent of the available logic.

Simulation Verifies Operation Before Programming a Device

Simulation was run on the device to assure proper operation. In this instance a serial input stream corresponding to a valid bit stream is read by the design. It properly sequences through the states, latches the data, and interrupts a processor. The anticipated simulation data is shown in Figure 7. The controller state machine is verified by comparing the Q0-Q3 outputs, and the state table values in Figure 5. The actual simulation run is shown in Figure 8.

AB27 Rev 2.0

Figure 7. Anticipated Simulation Data—*The Simulator will be driven with a data input emulating the above sequence. Design behavior is verified by monitoring for the correct response from the Functional Simulator.*

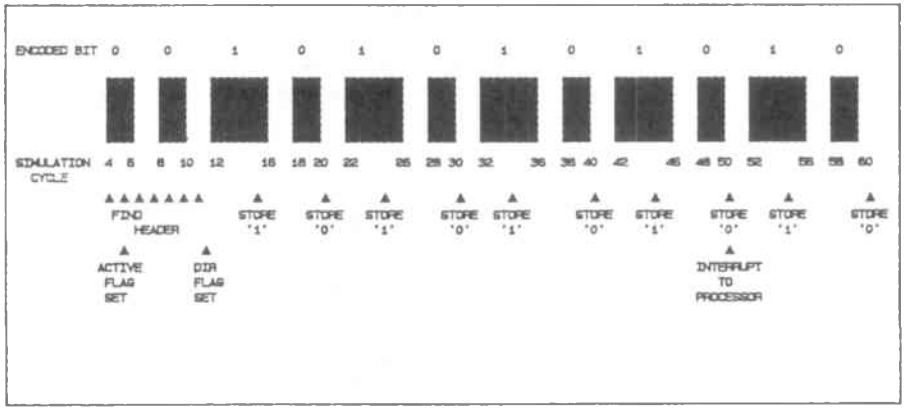
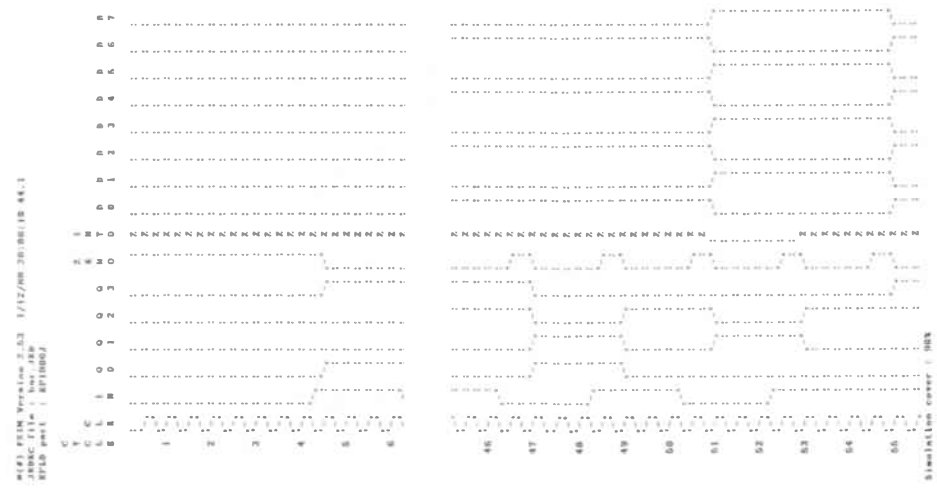


Figure 8. Simulation Run—*Simulation verifies correct operation.*



FEATURES

- **EPLD timing model.**
- **Relationship between data sheet parameters and model.**
- **Applying the model to simulate EPLD timing delays.**
- **Describes EPB1400 (BUSTER) microprocessor interface timing considerations.**

INTRODUCTION

EPLDs integrate complex logic functions into single chip solutions. After a design is functionally compatible with design requirements, timing analysis should be completed to assure A.C. parameter compatibility. This Application Brief discusses timing delays which exist in Altera EPLDs. The major focus is to present the internal delay paths inherent in every EPLD, show their relation to the data sheet AC specifications, and present simulation values for these parameters. Designers will be able to model and simulate their own logic designs once they understand how logic designs are actually implemented into EPLDs via the A+PLUS software.

EPLD ARCHITECTURE BASICS

GATE DELAYS DON'T PROVIDE EPLD TIMING CHARACTERISTICS

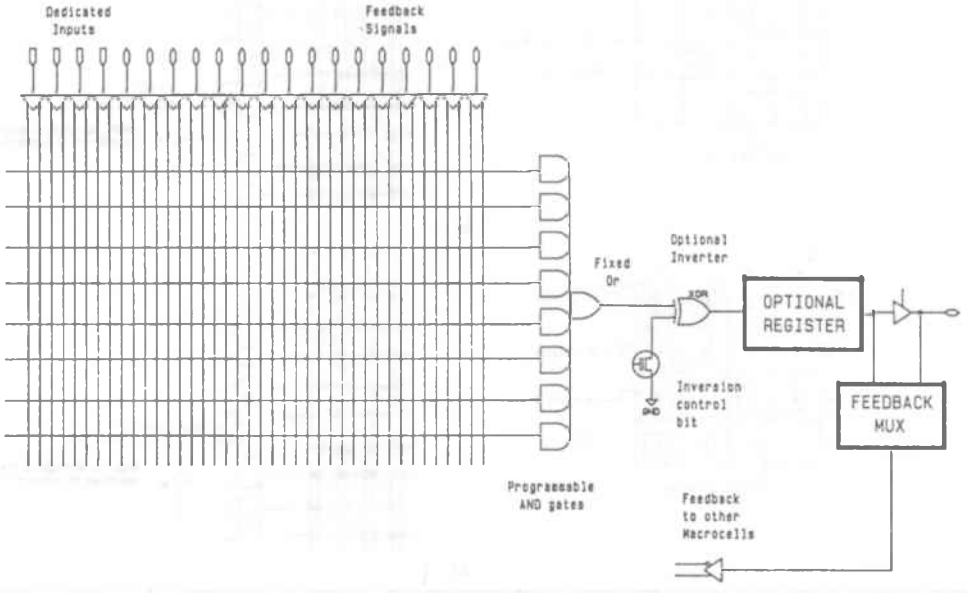
Accurately modeling the timing characteristics requires an understanding of how a given application is implemented within the EPLD. Most designs targeted for EPLDs contain basic gates, and TTL MacroFunctions, which are emulated by the EPLD general macrocell structure. The macrocell structure is an array of logic in an AND/OR configuration with a programmable inversion followed by an optional flip-flop and feedback (Figure 1).

When designing with EPLDs, the term "gate delay" is not a useful measure. Within the EPLD AND array are product terms. A product term is simply an n-input AND gate, where n is the number of connections. Depending on the logic implemented, a single product term may represent one to several gate equivalents.

THE AND/OR/INV STRUCTURE

The AND portion consists of a column of AND gates; each of which has a very large number of possible inputs selected by EPROM bits. The EPROM bits, serve as electrical switches. An erased bit passes the input into the AND gate

Figure 1. EPLD Macrocell



(switch on), while a programmed bit cuts it off (switch off). All bits are initially erased.

The number of possible inputs to an AND gate varies from 36 (EP320) to 88 (EP1800). In the EP310, EP320, EP600/EP610, and EP900/EP910 every dedicated input and its inversion and every macrocell feedback and its inversion are possible inputs to the AND gate. In the EP1210 and EP1800/EP1810, which have local and global bussing, not all of the macrocell feedback is available at every AND gate. The reason larger devices do not have all feedbacks feeding the AND gates is to preserve the speed characteristics of the device. As will be shown later, a significant portion of the component delay is in the migration through the array.

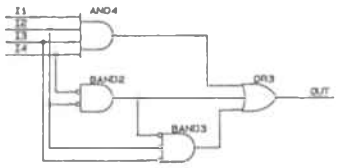
Following the AND gates is a fixed 8 input OR function (except the EP1210 which has OR gates that vary from 4 to 12 inputs). The structure is called a fixed OR because the AND functions are hardwired into the OR gates, and cannot be redistributed if unused.

The OR gate feeds a programmable inverter (XOR). A dedicated EPROM bit either programs the inversion function on or off.

DESIGNING FOR AND/OR EXPLICITLY

Though not obvious, the AND/OR/INV structure can implement general logic structures in either sum of products or product of sums forms. Because A+PLUS development software

Figure 2.



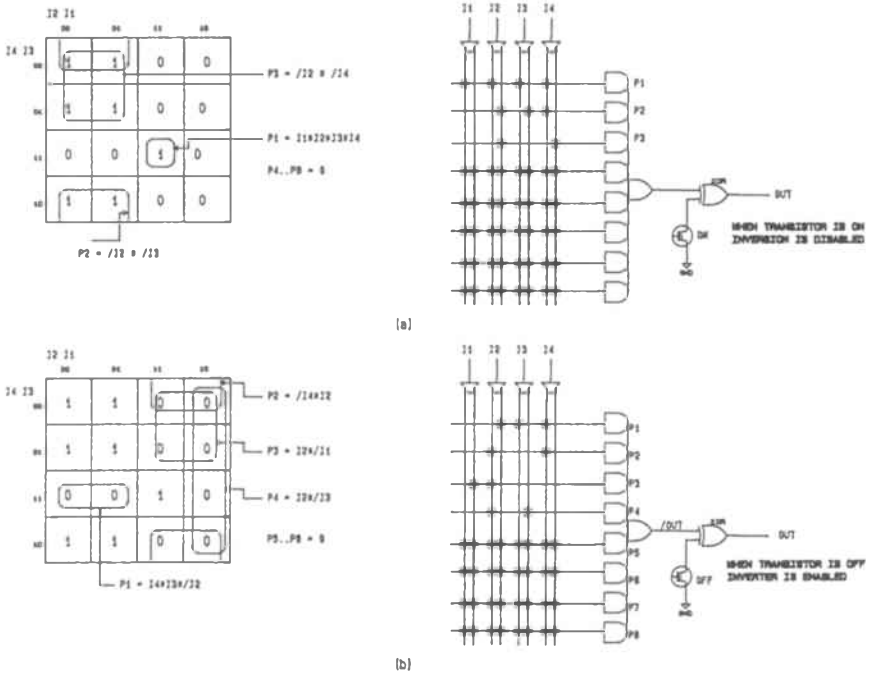
OUT = (I1*I2*I3*I4) + (/I2*/I4) + (/(/I2*/I4)*(/I2*/I3)

(a)

I4 I3 \ I2 I1	00	01	11	10
00	1	1	0	0
01	1	1	0	0
11	0	0	1	0
10	1	1	0	0

(b)

Figure 3.



automatically performs the translation between the designers input file and the minimized boolean form, designers will not be aware of the intermediate boolean form. Although Altera's A+PLUS design software performs the translation automatically, it is still important to understand how Boolean logic is implemented in the array structure.

Figure 2a,b shows a single network of SSI functions, its corresponding Boolean equation, and Karnaugh map. A sum of products equation is formed by blocking adjacent product groups, called product terms (PTERMS), and ORing them together. By programming appropriate input connections, each AND function can implement a single pterm, labeled p1, p2, and p3 in Figure 3a, by programming appropriate connections. Note that if there were more than 8 pterms it would not be possible to implement the logic in a single AND/OR array of the type shown. P4, p5, p6, p7, and p8 don't interfere as they are programmed out (LOW logic level since both the true and compliment of every signal are ANDed).

The architecture also supports product of sums, the ANDing of OR functions, through De Morgan's inversion; possible because both the true and complement forms are available to the AND array. Figure 3b shows the formation of product terms by grouping the zero's and setting the invert bit. In this instance there are more product terms, but sometimes this number can be dramatically smaller.

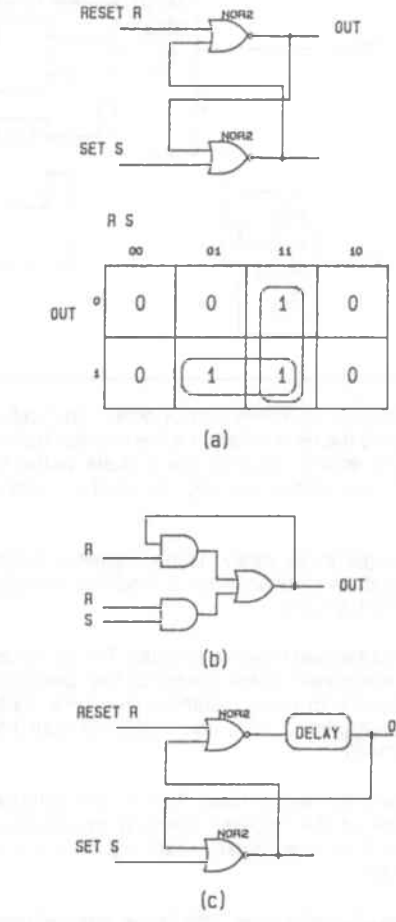
LOGIC MINIMIZATION IS NOT ENOUGH

The implementation of a latch points out a subtle difference between discrete and programmable logic structures. If the cross coupled latch structure in Figure 4a is implemented in an EPLD according to the logic extracted from the KMAP in Figure 4b, it will fail; a set pulse (S) will not result in a set output. This is not an error in minimization, but an oversight in implementation. The implicit understanding that portions of a design have delay must be made explicit, as shown in Figure 4d. This approach recognizes that OUT occurs some period after the inputs S or R.

EPLD DELAY ELEMENTS

The simplest solution to the architectural requirements is to model time through the logic array as a constant. This parameter is called tlad. The rest of the elements in the timing model are akin to those found in conventional logic. There are input and output delay parameters (tin, tio, tod); register parameters (tsu, th, tclr, tics, tic); and internal connection parameters (tfd). A detailed diagram is shown in Figure 5, with a description of the signals.

Figure 4. R-S Latch



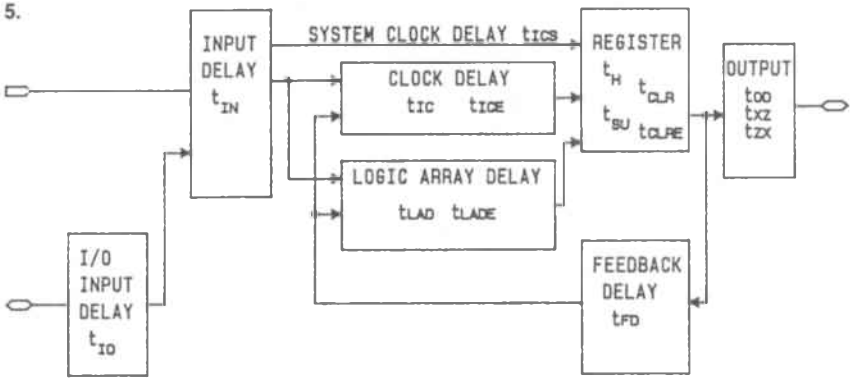
tin—Input pad and buffer delay which direct the true and complement data input signals into the AND array.

tio—I/O input pad delay. This delay applies to I/O pins committed as inputs.

tod—Output buffer and pad delay. For registered applications this incorporates the clock to output delay of the flip flop. In combinatorial applications it incorporates delay from the output of the array to the output of the device.

txz—Time to tri-state output delay. This delay incorporates the time between a high-to-low transition on the enable input of the tri-state buffer to assertion of a high impedence value at an output pin.

Figure 5.



tzx—Tri-state to active output delay. This delay incorporates the time between a low-to-high transition on the enable input of the tri-state buffer to assertion of a high or low logic level at an output pin.

tlad—Logic array delay. This parameter incorporates all delay from an input or feedback through the AND/OR structure.

tlade—Enhanced logic array delay. The structure of some macrocells is enhanced for fast propagation of signals. In these instances the faster tlade parameter applies. This is currently only available in the EP1800.

tsu—Register setup time. This is the internal setup time of the register inside a macrocell—measured from the register data input until the register clock.

th—Register hold time. This is the internal hold time of the register inside a macrocell: measured from the register clock to the register data input.

tclr—Asynchronous register clear time. This is the amount time it takes for a low signal to appear at the output of a register after the transition at the logic array.

tclre—Enhanced asynchronous register clear time. Similar to the tclr delay, with the exception that this faster parameter applies to enhanced macrocells. This parameter is currently only valid for the EP1800.

tics—System clock delay. This delay incorporates all delays incurred between the output of the input pad and the clock input of the registers for dedicated clock pins.

tic—Clock delay. This delay incorporates all the delay incurred between the output of an input pad or I/O pad and the clock input of a register, including the time required to go through the logic array. This delay is differentiated from the system clock delay tics, by the need to pass through a CLKB primitive, which specifies individual register clocking.

tice—Enhanced Clock delay. Similar to the tic delay, with the exception that this faster parameter applies to enhanced macrocells.

tfd—Feedback delay. In registered applications, this is the delay from the output of the register to the input of the logic array. In combinatorial applications it is the delay from the combinatorial feedback to the input of the logic array.

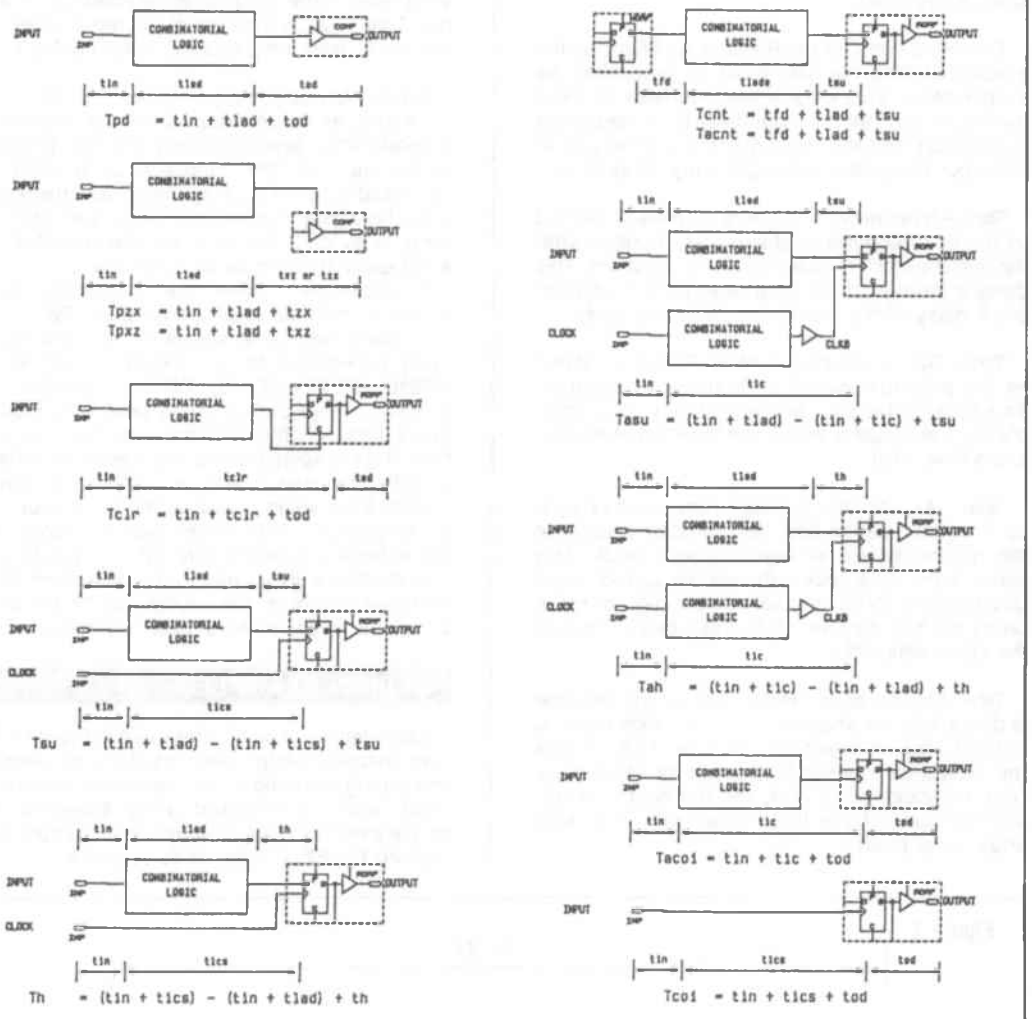
Data Sheet Specifications

The data sheet for each Altera EPLD references timing parameters which characterize the AC operating specifications. These parameters are measured values, derived from extensive device characterization and guaranteed by 100 percent testing. The data sheet shows worst-case values. Among the AC characteristics are the following: Tpd1, Tpd2, Tpzx, Tpxz, Tclr, Tsu, Th, Tco1, Tcnt, Tasu, Tah, Taco1, Tacnt. These parameters, described below in detail, may be represented by the EPLD internal delay elements.

Tpd1—Propagation Delay is defined as dedicated input to non-registered output. The propagation delay is the time required for any dedicated input to propagate through any combinatorial logic and appear at the EPLD external output pin. This delay is the sum of input delay(tin), array delay (tlad) and output delay (tod).

Tpd2—Propagation Delay is defined as I/O pin to non-registered output. The propagation delay is the time required for any external I/O input to

Figure 6. Timing Equations



propagate through any combinatorial logic and appear at the EPLD external output pin. This delay is the sum of the I/O delay (tio), input delay (tin), array delay (tlad) and output delay (tod).

Tpzx—High Impedence to Active output is the time required to change an external output from tri-state to a valid high or low logic level measured from an input transition. This delay is the sum of input delay (tin), array delay (tlad), and the time to de-activate the tri-state buffer (tzx).

Tpxz—Time to Tristate Delay is the time required to change an external output from a valid high or low logic level to a tristate from an input transition. This delay is the sum of input delay (tin), array delay (tlad), and the time to activate the tristate buffer (txz).

Tclr—Time to Clear Register Delay is the time required to change the output from high to low through a register clear measured from an input transition. This delay is the sum of input delay (tin), register clear delay (tclr), and the output delay (tod).

Tsu—Setup Time on the Register is defined as the time data must be present at the register before the system clock. This value is the difference between the sum of input delay (tin), array delay

(tlad), and internal register setup time (tsu) and the sum of the input delay (tin) and the system clock delay (tics).

Th—Hold Time for the Register is defined as the amount of time the data must be valid after the system clock. This value is the difference between the sum of the internal input delay (tin), the system clock (tics), and the hold time (th) and the sum of the input delay (tin) and logic array delay (tlad).

Tco1—System Clock to Output Delay is defined as the time required to obtain a valid output after the system clock is asserted on an input pin. This delay is the sum of the input delay (tin), the system clock delay (tics), and the output delay (tod).

Tcnt—System Clocked Counter Period is defined as the minimum period a counter can maintain. This delay is the sum of the feedback delay (tfd), the logic array delay (tlad), and the internal register setup time (tsu).

Tasu—Asynchronous Setup Time on the Register is defined as the time data must be present at the register before an asynchronous clock. This value is the difference between the sum of input delay (tin), array delay (tlad) and the register setup time (tsu) and the sum of the input delay (tin) and the clock delay (tic).

Tah—Asynchronous Hold Time for the Register is defined as the amount of time the data must be present after an asychronous clock. This value is the difference between the sum of the input delay (tin), the clock delay (tic), and the hold time (th) and the sum of the input delay (tin) and logic array delay (tlad).

Taco1—The Asynchronous Clock to Output Delay is defined as the time required to obtain a valid output after a clock is asserted on an input pin. This delay is the sum of the input delay (tin), the clock delay (tic), and the output delay (tod).

Tacnt—Asynchronous Clocked Counter Period is defined as the minimum period a counter can maintain when asynchronously clocked. This delay is the sum of the feedback delay (tfd), the enhanced logic array delay (tlade), and the register setup time (tsu). In some components an enhanced array delay does not exist and the standard logic array delay (tlad) must be substituted.

It is possible to solve for the model parameters by using the data sheet parameters. Table 1, the simulation data table, shows each of the internal delay parameters for the EP310, EP320, EP600/EP610, EP900/EP910, EP1210, EP1800 and EPB1400. Note that in some cases the modeled result differs slightly from the data sheet specification. This discrepancy is the result of different guardbands used to ensure compliance with the specification when tested during the manufacture of the product. The model does not account for the extended guardbanding of some data sheet specifications. Even though the data sheet shows max/min numbers, the model results should be considered "typical worst case" numbers.

DESIGN EXAMPLES

Consider the circuit represented in Figure 7. This combinatorial design uses no clock or feedback. The timing restrictions are dependent only on the input, array, and output delay elements. This simple example will help illustrate circuit partitioning into the EPLD internal delay paths.

Figure 7.

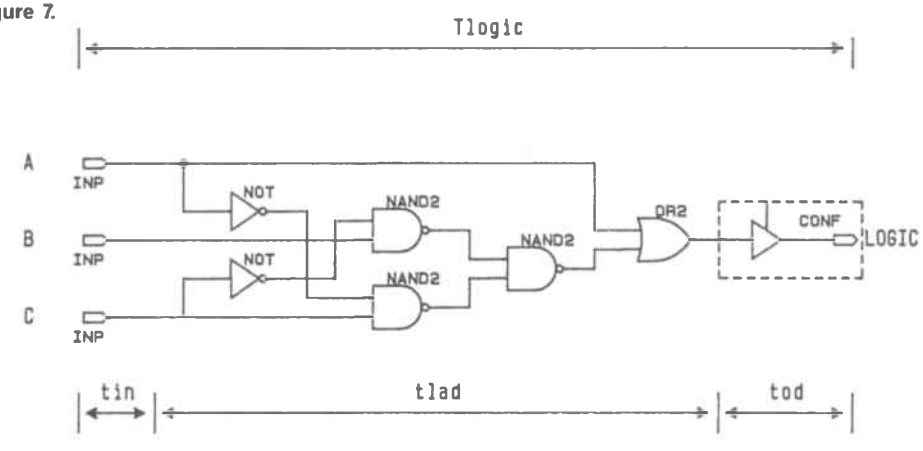
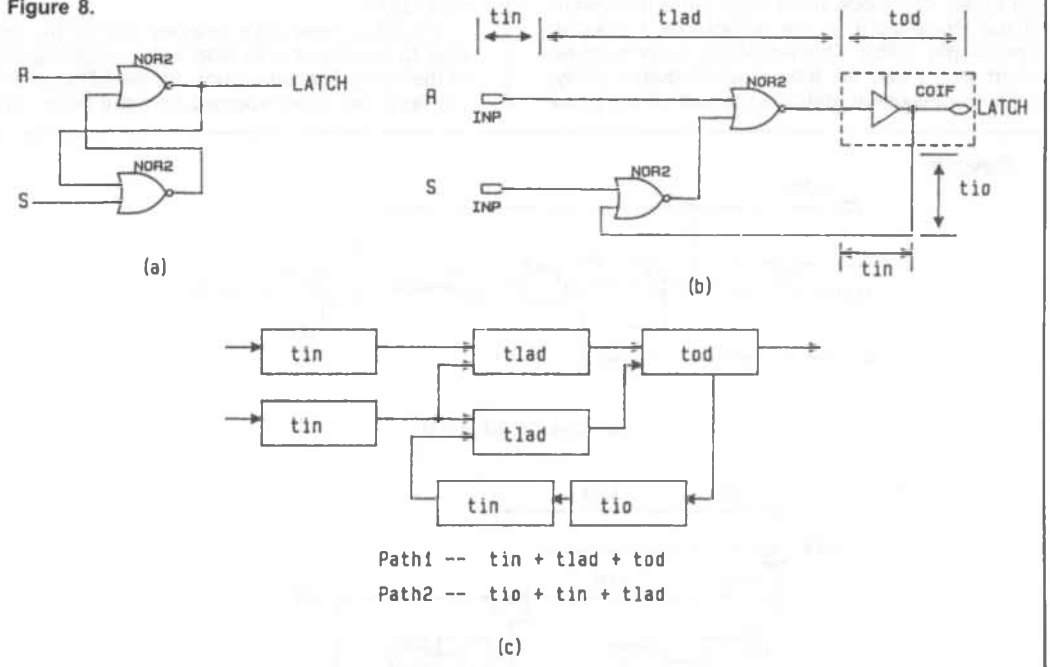


Figure 8.



As Figure 7 shows, the circuit consists of three inputs, gating logic, and one output. The circuit is therefore partitioned into three delay paths: input delay, array delay, and output delay. Notice that all the gating logic is incorporated into the AND/OR array. The maximal time allowed for any input to propagate through the entire circuit (from input pin to output pin) is also known as the propagation delay, Tpd1. The worst case Tpd1 is given in the respective EPLD data sheet. Propagation delay always applies when combinatorial output logic is used. Thus

tlogic = tin + tlad + tod = Tpd1

Return to the instance of cross coupled NOR gates to examine a design using combinatorial feedback. Figure 8b shows the implementation in EPLD primitives, and Figure 8c shows the design partitioned into appropriate delay paths; input, output, array and feedback delays. The propogation delay is the maximum combinatorial delay from input to output; shown as path 1 in Figure 8c. If the output of the latch is a logical zero (Q=L), and a logical one is applied to the set input (S=H), the output of the latch will go to a logic one (Q=H) within one propagation delay. The same principle is true if the latch is reset (R= H, S =L).

Path 2 is the hold time of the latch. To avoid any unwanted output glitches, the asynchronous set-up time of the latch must be observed. The pulse width of either Reset or Set inputs must be long enough to ensure that the feedback signal can return from the output primitive and reinsure the latched condition. The input pulse width must be greater than the time required for the feedback signal to return to the AND array and propagate through the gating logic. Otherwise, an H-L or L-H glitch may occur at the output. Thus, input pulse width is greater than the sum of tio, tin, and tlad. Note that (tio + tin) is the feedback delay in this case because the COIF feedback comes from the I/O pin.

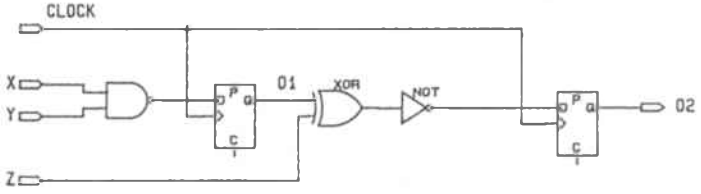
The next example evaluates a synchronous design presented in Figure 9. This circuit uses three inputs, one clock (common to both D-flip-flops), gating logic, and one output. Note that the output is active low. The design is represented with the Altera primitive symbols and partitioned to highlight the delay paths in Figure 9b. There are five inherent delays in this design: clock delay, input delay, array delay, feedback delay, and output delay. There are also setup and hold time requirements on each of the registers. All gating logic, including the inverter, is incorporated into the AND/OR array.

Consider the clock to output delay detailed by path 1. Path 1 refers to the output register of Figure 9c (lower D flip-flop). Assuming the data satisfies the setup time for the register, a rising edge clock at the clock of the flip-flop causes data residing on the D input to appear at the output pin,

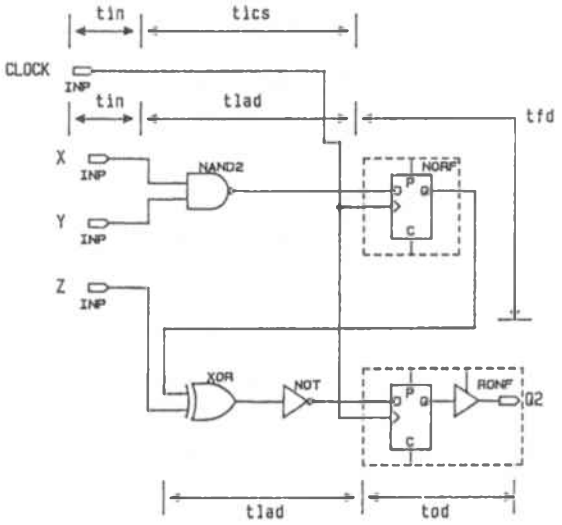
after passing through the register and tristate output buffer. The clock at the input pin is delayed in to the clock input of the register by a specific amount (tin + tics). This additional delay must be incorporated into the total clock-to-output delay. Thus, the maximum delay is the sum of the input delay (tin), the clock delay (tics), and the output delay (tod).

The set-up time is the time required for the input data to become stable before the triggering edge of the clock. The set-up time for the EPLD internal flip-flops has been modeled from the input, array,

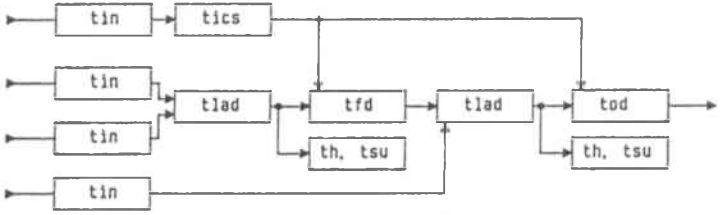
Figure 9.



(a) Circuit Schematic



(b) EPLD primitive implementation

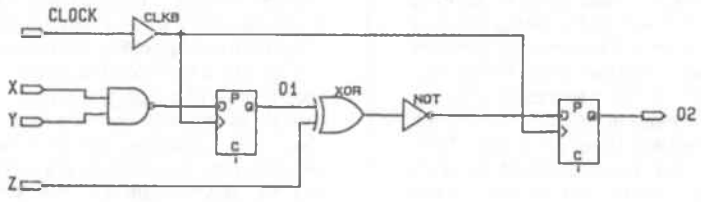


Path1 = tin + tics + tod
Path2 = (tin + tlad + tsu) - (tin + tics)
Path3 = tfd + tlad + tsu

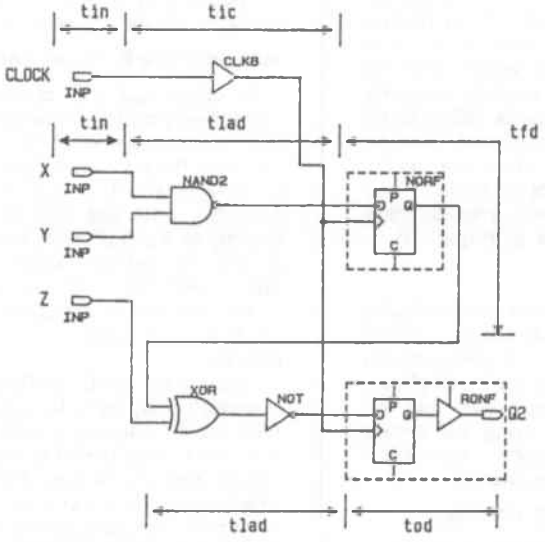
(c)

and clock delay paths as well as actual register setup time. Path 2 specifies the setup time requirement. The first term is requirement for stable data at the flip-flop and satisfy the register setup. It is the sum of the input, logic array, and setup time. The clock is delayed from the input pin to the clock input of the flip-flop, by the sum of the input delay and the clock delay. The setup time for the EPLD is the difference between these quantities—this is the equation of path 2. As long as the set-up time is obeyed at the external inputs, the circuit will function properly.

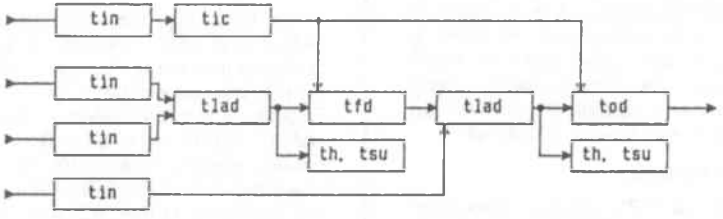
Figure 10.



(a) Circuit Schematic



(b) EPLD primitive implementation



Path1 = tin + tic + tod
Path2 = (tin + tlad + tsu) - (tin + tic)
Path3 = tfd + tlad + tsu

(c)

The maximum internal counter frequency, data sheet specification Tcnt, is governed by the internal feedback path. This is indicated in path 3 of Figure 9 as the time required for a signal to pass from the upper register to the lower register. This path determines the minimum internal clock period limit for the circuit. Once data is triggered into the top register, the signal passes through a feedback delay (tfd) and array delay (tlad) before reaching the bottom register to meeting the setup time of the register (tsu). The sum of these delays specifies the maximum practical internal clock frequency. Therefore, circuits which are dependent only on internal signals can operate at the minimum clock period specified by the data sheet parameter Tcnt.

Some circuit functions are dependent on both external and internal signals. When this is the case, the maximum clock frequency is dependent on the input delay, array delay, output delay, feedback delay, and clock delay.

The last example illustrates use of the programmable clock feature in applicable EPLDs (Figure 10). The circuit schematic is similar to the first stage of Figure 9a, with the exception that the programmable clock option is invoked by using the CLKB primitive. This change is reflected by the substitution of the tics parameter with tic (Figure 10b). Figure 10c shows the timing paths. Note that the clock to output delay (path 1) and the setup time parameters are both affected (path 2), but not the internal feedback path (path 3).

MACROFUNCTION TIMING

A+PLUS development software automatically decomposes MacroFunctions into low level EPLD architecture elements, which can be analyzed with the timing model and techniques described above. Worst case MacroFunction timing can be obtained prior to decomposition by applying the timing model to the primitive MacroFunction representations found in the library documentation.

TIMING SIMULATION FOR THE EPB1400

Timing simulation for the EPB1400 is similar to the general purpose EPLDs with the exception of timing paths involving the EPB1400 dedicated microprocessor (MPU) interface. A description of these MPU interface functions and their timing models, how to calculate delay paths to these functions, and some examples showing timing calculations follows. This section should be read in conjunction with the EPB1400 data sheet.

The MPU Interface Functions

The symbols for the MPU interface functions are shown in Figure 5 of the EPB1400 data sheet. The functions consist of octal input registers, input latches and a bus port transceiver. Since functions are not modeled out of the programmable logic array, their timing is different from that of general purpose macrocells: Each function has a unique timing model.

For simplicity, each MPU interface function's timing parameters are similar to those describing the TTL function which best approximates its behavior. For example, the RBUSI function in Figure 5 of the EPB1400 data sheet is similar to a 74377 function except a signal called /WS (Write Strobe) feeds the clock input, and a signal called WE (Write Enable) is inverted and feeds CE (Chip Enable). Relevant timing should be specified as it is for the 74377 in this configuration.

Such timing specification can be found in Figure 15B of the EPB1400 data sheet. All setup (Tsu), hold (Th), and propagation (Tp) delays are shown as well as the required widths for control signals (Tw). For example, setup times are specified between data inputs and /WS, between data inputs and WE, and between WE and /WS. Waveforms are provided to clarify any ambiguities between the parameters and the events they represent. EPB1400 data sheet Figures 15A through 15F provide similar timing information for the remaining MPU interface functions.

MPU Interface Function Control Input Timing

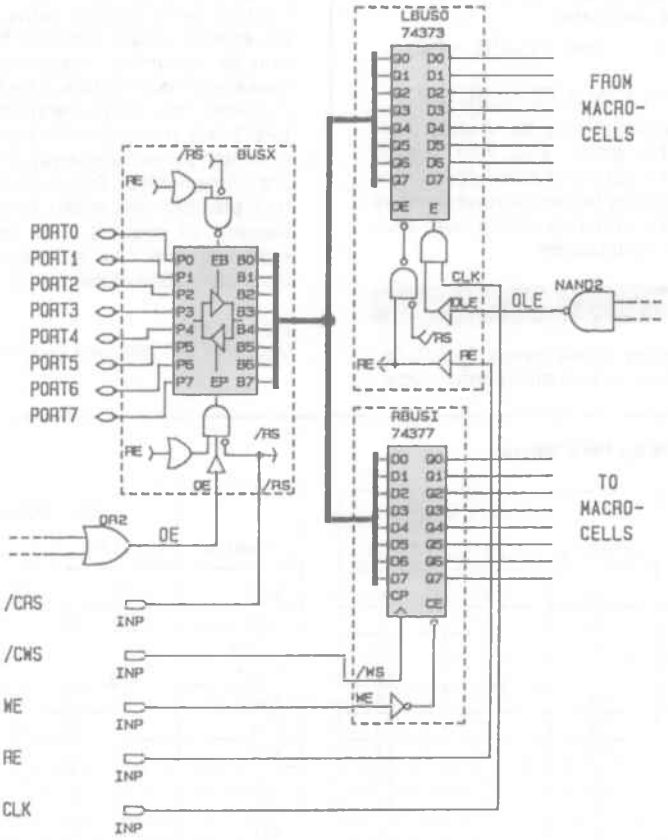
To obtain full A.C. characterization of timing paths involving MPU interface functions, the relevant microparameters are added to interface function timing parameters. Control inputs are either connected to dedicated strobes or logic. The /RS, /WS and CLK strobes are connected directly to input pins, as are synchronous clocks in general purpose macrocells. Other control inputs (RE, WE, OE, OLE) are fed by control macrocells, which are buried combinatorial functions with 2 product terms and an optional inversion.

Figure 7 on the EPB1400 data sheet shows the macrocell delay paths for the EPB1400 device. /RS, /WS, and CLK signals pass through the input delay (tin) and then to the MPU interface function; other control inputs pass through the control array delay (tcad). Input pins, I/O pins, input latch feedback, or general purpose macrocell feedbacks may all serve as control array and logic array inputs.

MPU Interface Timing Example

The following demonstrates using microparameters and MPU interface timing parameters to calculate internal timing paths. Figure 11 shows a simple microprocessor interface containing a BUSX bus port transceiver, a RBUSI input register and an LBUSO output latch. The A.C. compatibility issues generally concern the time required to read or write data into the EPB1400 from an MPU databus. In Figure 11 the /WS and /RS control signals are connected to device inputs /CWS and /CRS respectively. WE and RE are also connected to input pins, but still pass through the control array, although the control array has no logical effect on the signal values in this instance.

Figure 11.



Consider timing relevant for writing data to the input register. One delay path is from a /CWS to valid data at the output of the RBUSI. The delay is:

T(/CWS to valid RBUSI output) =
tin + tp(/WS -Q)[RBUSI].

Another delay is setup of WE to /WS. This is the time for WE to reach RBUSI less the time for /CWS to arrive it:

Tsu(WE to /CWS) = (tin + tcad +tsu(WE-/WS)[RBUSI]) -(tin)
= (tcad + tsu(WE-/WS)[RBUSI])

Data inputs pass through the BUSX transceiver to the input of the RBUSI, and incur delay specified in the propagation delay (tp) parameters of the BUSX function (Figure 11). Data on the bus port must satisfy a setup time before being latched into the RBUSI input register. Assume that /RS is disabled sufficiently long before data is at the bus port so that the BUSX primitive is already driving the internal bus. The setup time of the RBUSI is then the time to pass through BUSX, satisfy the RBUSI data setup requirement, less the time for /WS to reach RBUSI.

Tsu(DATA to /CWS) = tp(p-ib)[BUSX] +
tsu(ib -/WS)[RBUSI] -tin.

Analyzing the time required to read data from the EPB1400 involves the LBUSO interface function instead of RBUSI. Assume data is already stored in the LBUSO output latch. To read this data on the MPU databus, the LBUSO data must be enabled from the LBUSO onto an internal bus, and pass through BUSX to the bus port pins. There are two timing paths, and both must be satisfied. One path considers the /CRS enabling the LBUSO contents onto the internal bus and passing that data out to the port as the worst case. The other considers the /CRS enabling the BUSX contents out to the port as the worst case. These are:

(1) Tzx(/CRS to Data) = tin +
txz(/RS -ib)[LBUSO] + tp(ib -p)[BUSX]

(2) Tzx(/CRS to Data) = tin + txz(/RS -p)[BUSX]

The read enable setup time, and the time to disable data are also calculated:

Tsu(RE -/CRS) = (tin + tcad + tsu(RE -/RS) [LBUSO]) -tin.
Txz(/RS -Data) = tin + tpxz(/RS -p)[BUSX].

Many other timing paths may be studied. Permutations of possible paths, and their related timing are beyond the scope of this Applications Brief. Using combinations of the microparameters and the MPU interface function timing, any arbitrary delay path can be modeled.

CONCLUSION

To understand timing relationships in EPLDs, break up the internal paths into meaningful micro-parameters that model portions of the EPLD architecture. Once internal paths are decomposed, its possible to obtain accurate timing delay information by summing appropriate combinations of these microparameters. The simulation data table provides the micro-parameter values. Relevant EPLD data sheets provide architectural information on which parameters apply, and how the primitives are implemented. The A+PLUS development system provides minimized files that aid in decomposition of designs. The combination of these elements and the techniques described allow characterization of any timing path within an EPLD.

Table 1. EPLD Timing Parameters

parameter	PART NUMBER EP1800-2	EP1800-3	EP1800
tin	10	12	14
tio	5	5	5
tlade	35	39	43
tlad	40	44	48
tod	15	19	23
tzx	15	19	23
txz	15	19	23
tsu	12	14	18
th	30	30	30
tic	40	44	48
tice	35	39	43
tics	4	4	4
tfd	10	14	16
tclr	40	44	48

parameter	PART NUMBER EP1210-1	EP1210-2	EP1210
tin	10	12	16
tio	3	3	3
tlade	*	*	*
tlad	36	42	62
tod	9	11	12
tzx	9	11	12
txz	9	11	12
tsu	18	17	11
th	10	10	10
tic	*	*	*
tice	*	*	*
tics	16	16	16
tfd	4	6	8
tclr	71	87	122

parameter	PART NUMBER EP900-2	EP900-3	EP900
tin	6	8	8
tio	5	5	5
tlade	*	*	*
tlad	29	31	35
tod	10	11	12
tzx	15	16	17
txz	15	16	17
tsu	13	14	15
th	13	13	16
tic	29	30	34
tice	*	*	*
tics	4	4	4
tfd	8	10	10
tclr	34	36	40

parameter	PART NUMBER EP320-1	EP320-2	EP320
tin	4	5	7
tio	1	1	1
tlade	*	*	*
tlad	20	22	26
tod	5	7	11
tzx	6	8	12
txz	6	8	12
tsu	8	10	14
th	10	10	10
tic	*	*	*
tice	*	*	*
tics	8	7	7
tfd	7	8	10
tclr	*	*	*

Table 1. (continued)

parameter	PART NUMBER EP600-3	EP600
tin	9	10
tio	2	2
tlade	*	*
tlad	23	29
tod	11	14
tzx	13	16
txz	13	16
tsu	14	14
th	10	15
tic	23	29
tice	*	*
tics	4	8
tfd	8	12
tclr	25	31

parameter	PART NUMBER EP310-2	EP310
tin	7	10
tio	2	2
tlade	*	*
tlad	20	27
tod	8	12
tzx	8	12
txz	8	12
tsu	10	10
th	10	10
tic	*	*
tice	*	*
tics	4	4
tfd	3	5
tclr	30	33

parameter	PART NUMBER EP910-30	EP910-35	EP910-40
tin	7	8	8
tio	3	3	3
tlade	*	*	*
tlad	16	19	22
tod	7	9	10
tzx	7	9	10
txz	7	9	10
tsu	10	10	10
th	15	15	15
tic	16	19	22
tice	*	*	*
tics	4	5	6
tfd	4	6	8
tclr	19	22	25

parameter	PART NUMBER EP610-25	EP610-30	EP610-35
tin	5	6	7
tio	2	2	2
tlade	*	*	*
tlad	14	17	19
tod	6	7	9
tzx	6	7	9
txz	6	7	9
tsu	8	8	8
th	12	12	12
tic	14	17	19
tice	*	*	*
tics	4	4	4
tfd	3	5	8
tclr	16	17	21

parameter	PART NUMBER EPB1400-2	EPB1400
tin	4	5
tio	5	7
tlad	20	25
tcad	16	18
tod	11	15
tzx	15	20
txz	15	20
tsu	10	12
th	10	12
tic	20	25
tice	*	*
tics	5	5
tfd	3	4
tclr	25	30

* indicates the specification does not apply

NOTES

4

MICROPROCESSOR PERIPHERAL AND SUPPORT LOGIC APPLICATIONS

FEATURES

- **Address Decode and Chip Select Logic.**
- **Wait State Generation.**
- **Dynamic RAM Control.**

INTRODUCTION

Microcomputer address decoding for memory and peripheral devices, traditionally done with discrete logic or ROM's, is now done more effectively with EPLDs. The advantages include faster decode times, decreased PC board space, and significant power savings.

This Application Note presents address decoding, wait state generation, and memory control solutions using Altera EPLDs. The Intel 8086 and the Motorola 68000, examples of microprocessors with synchronous and asynchronous data buses respectively, are interfaced to several memory and peripherial devices via the EP610 and EP310.

8086 SOLUTIONS

ADDRESS DECODING

The circuit in Figure 1 shows an EP610 providing chip select signals in a high speed serial data line multiplexer. Control comes from a 8086 microprocessor whose program is stored in a 2764 EPROM. A 6164 static RAM stores data packets as they are assembled and stores the 8086's stack. Four SCC (serial communication control) peripheral chips provide an interface between the 8086 CPU and the serial data lines.

The EP610 in Figure 1 decodes the address lines of the 8086 into chip select signals for the CPU's peripherals (RAM, ROM, and SCC). The EP610's outputs follow the memory map in Figure 2a, where the accompanying Boolean equations describe the memory map for a 16-bit address bus. A more complicated memory map can be implemented by changing the equations. An Altera utility program called 'DECODER' helps here by automatically deriving equations for complicated memory maps. This program is available from Altera Bulletin Board Service (see Utility Program section).

WAIT STATE GENERATION

In addition to address decode, the EP610 also performs wait state generation. It is inefficient to run a microprocessor at the speed of its slowest peripheral chip. Instead, the microprocessor is usually clocked at its top speed and a wait state generator slows the microprocessor's bus cycles when the slower peripheral is selected. Wait state generators are implemented with a counter and decode logic. Figure 2b shows the state table for an 8086 wait state generator. To implement a wait state the READY signal must be driven LOW before the falling edge of T2. When accessing the ROM, the EP610 asserts the READY signal LOW for one 8086 clock cycle (one wait state). The READY signal is asserted LOW for two clock cycles (two wait states) if the RAM is selected. Note, for either wait state, a memory read or write (MRDC, MWTC) and address latch enable (ALE) must be valid. Figure 3 shows the EP610 logic required for both the address decode and wait state circuits.

68000 SOLUTIONS

Figure 4 shows a 68000-based system with an EP310 programmed as a wait state generator and address decoder.

Wait states in a 68000 circuit are easy to generate by delaying the DTACK (Data Transfer Acknowledge) signal. Two or three flip-flops can count the number of clock pulses after the assertion of either of the 68000's data select lines, LDS and UDS, and assert DTACK once a programmed count is reached. The programmed terminal count can depend on the speed of the selected peripheral device, allowing different numbers of wait states for different peripherals.

Figure 6 shows the logic schematic for the EP310. The design provides two chip-select signals, one for a 256K bank of D-RAM and the other for a 64K bank of ROM. The EP310 asserts DTACK one clock cycle after ROM is selected, providing the ROM with two wait states (WAIT2), two clock cycles after RAM read (WAIT4) and three clock cycles after RAM write (WAIT6).

As an added element of security, the 68000's bus error line, BERR, is asserted if neither RAM nor ROM is selected and no device asserts the DTACKIN signal by wait state 8. This feature is useful in industrial control applications to signal a controller fault or be used to generate system reset once a fault has been detected.

DYNAMIC RAM CONTROL

Dynamic RAM circuits must generate several control signals: RAS, CAS, and WR for the DRAMs themselves, MAS (memory address select) for the address decoder, and a DTACK or READY line to acknowledge the data transfer to the CPU. DRAM controllers connected to some 16 bit micropro-

Figure 1. 8086-EP610 Interface—*The EP610 controls the ROM, RAM and SCC chip select logic as well as memory wait state generation.*

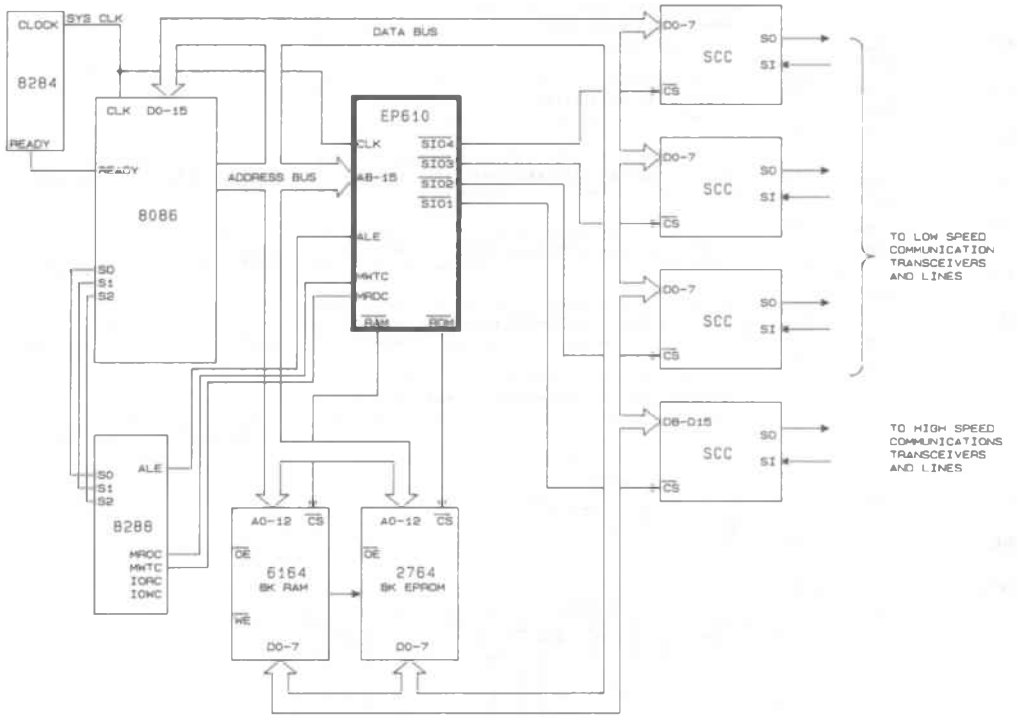
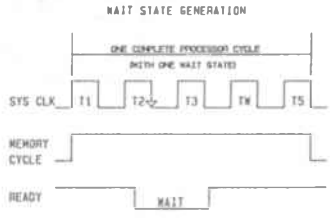


Figure 2. Address Decode Memory Map

a. Memory map for EP610 address decode in Figure 1.

SIGNAL NAME	LOW ADDRESS	HIGH ADDRESS	EQUATION
ROM	0000	1FFF	= A15' * A14' * A13'
RAM	2000	3FFF	= A15' * A14' * A13
SIO0	8000	80FF	= A15 * A14' * A13' * A12' * A11' * A10' * A9' * A8'
SIO1	8100	81FF	= A15 * A14' * A13' * A12' * A11' * A10' * A9' * A8
SIO2	8200	82FF	= A15 * A14' * A13' * A12' * A11' * A10' * A9 * A8'
SIO3	8300	83FF	= A15 * A14' * A13' * A12' * A11' * A10' * A9 * A0
SIO4	8400	84FF	= A15 * A14' * A13' * A12' * A11' * A10 * A9' * A8'

b. To implement a 8086 wait state, the READY signal must be LOW before the falling edge of T2.



EP610 WAIT STATE GENERATOR

STATE NAME	W0	W1	DEVICE
STANDBY	0	0	RESET
WAIT 1	1	0	ROM
WAIT 2	1	1	RAM

Figure 3. EP610 Address Decode and Wait State Generation—*Logic schematic for EP610. LogiCaps supports both gate level and equation design entry.*

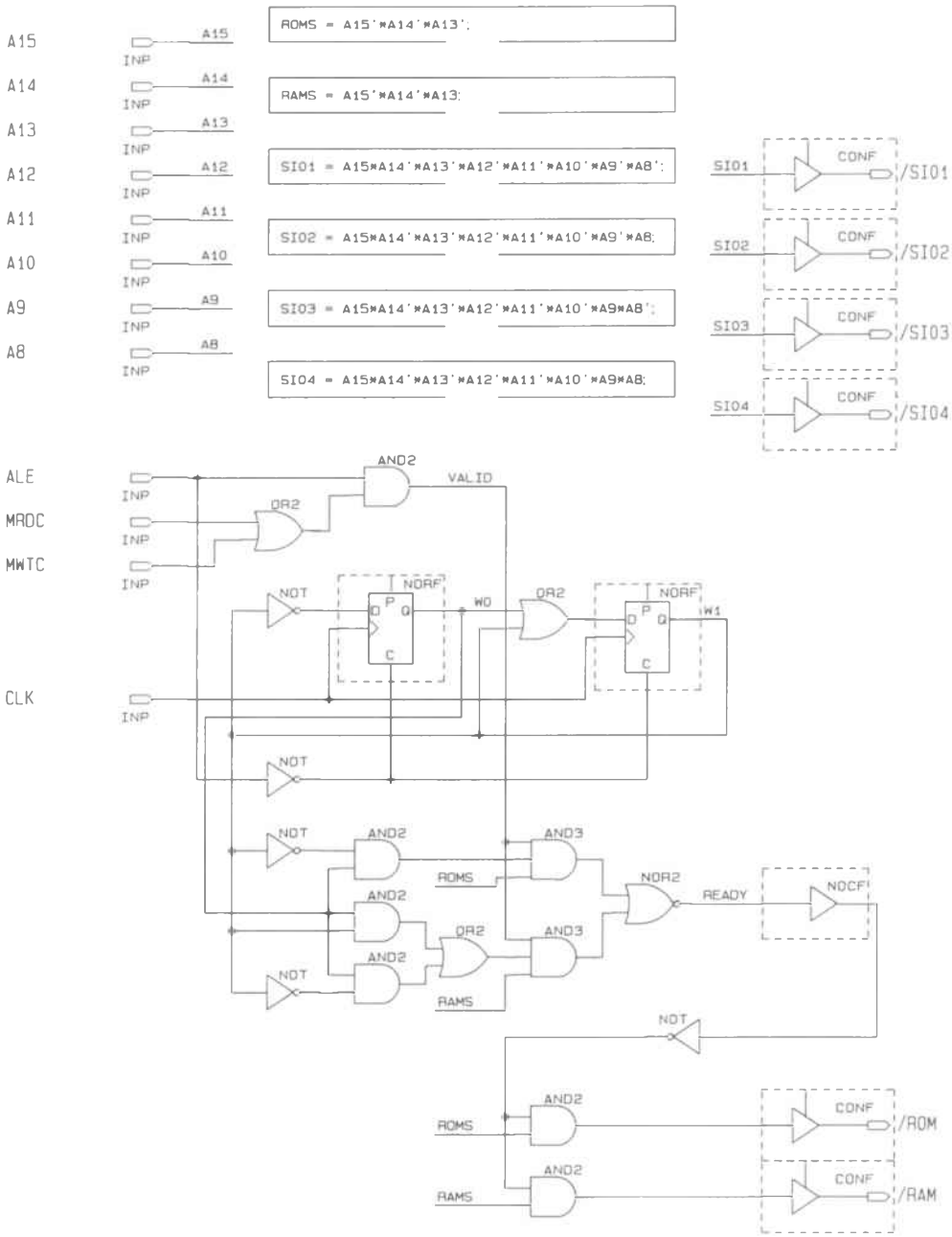
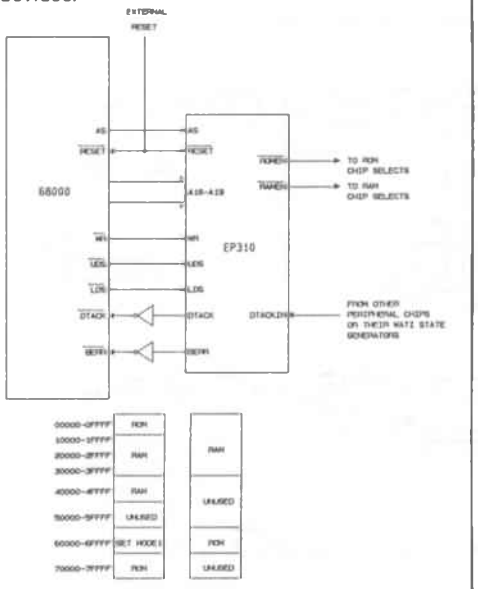


Figure 4. 68000-EP310 Interface—*The EP310 implements both address decode and wait state generation for RAM and ROM memory devices.*



cessors must allow 8-bit transfers over either half of a 16-bit bus (8086 or 68000). In addition, DRAMs must be refreshed—either once each bus cycle or in burst mode—a function that can be provided by either the DRAM controller, a DMA channel or an interrupt-driven CPU software loop.

A DRAM controller that handles all of the above is implemented in the EP310 as shown in Figure 6. The EPLD performs the following actions:

1. Place one half of the CPU address on the DRAM's address lines (negate MAS).
2. Assert the ROW Address Strobe.
3. (a) Place the second half of the CPU address on the DRAMs address lines (assert MAS).
   (b) Assert the DRAM's WR line if the bus transfer is a write operation.
4. Assert the Column Address Strobe.
5. Wait for the bus cycle to terminate, then negate all signals.

An EP310 programmed as Figure 5 remains in a state with all signals negated (number 1 above) until a bus cycle is started with RAMEN and either UDS or LDS. The EP310 then cycles through the actions above at a rate set by the clock, stopping at number 5 until the bus cycle end (RAMEN, UDS and LDS negate), then returns to number 1 to await the next bus access.

AN3 Rev 3.0

Figure 5. EP310 Dynamic RAM Control—*DRAM controller implemented in the EP310. The controller's clock frequency must be exactly that of the 68000 system clock.*

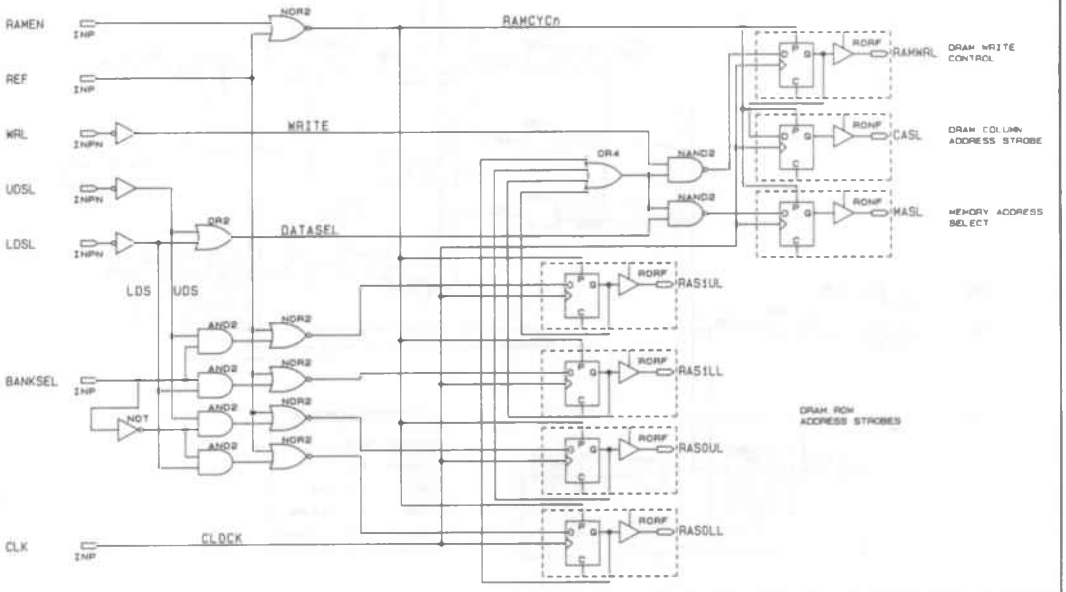
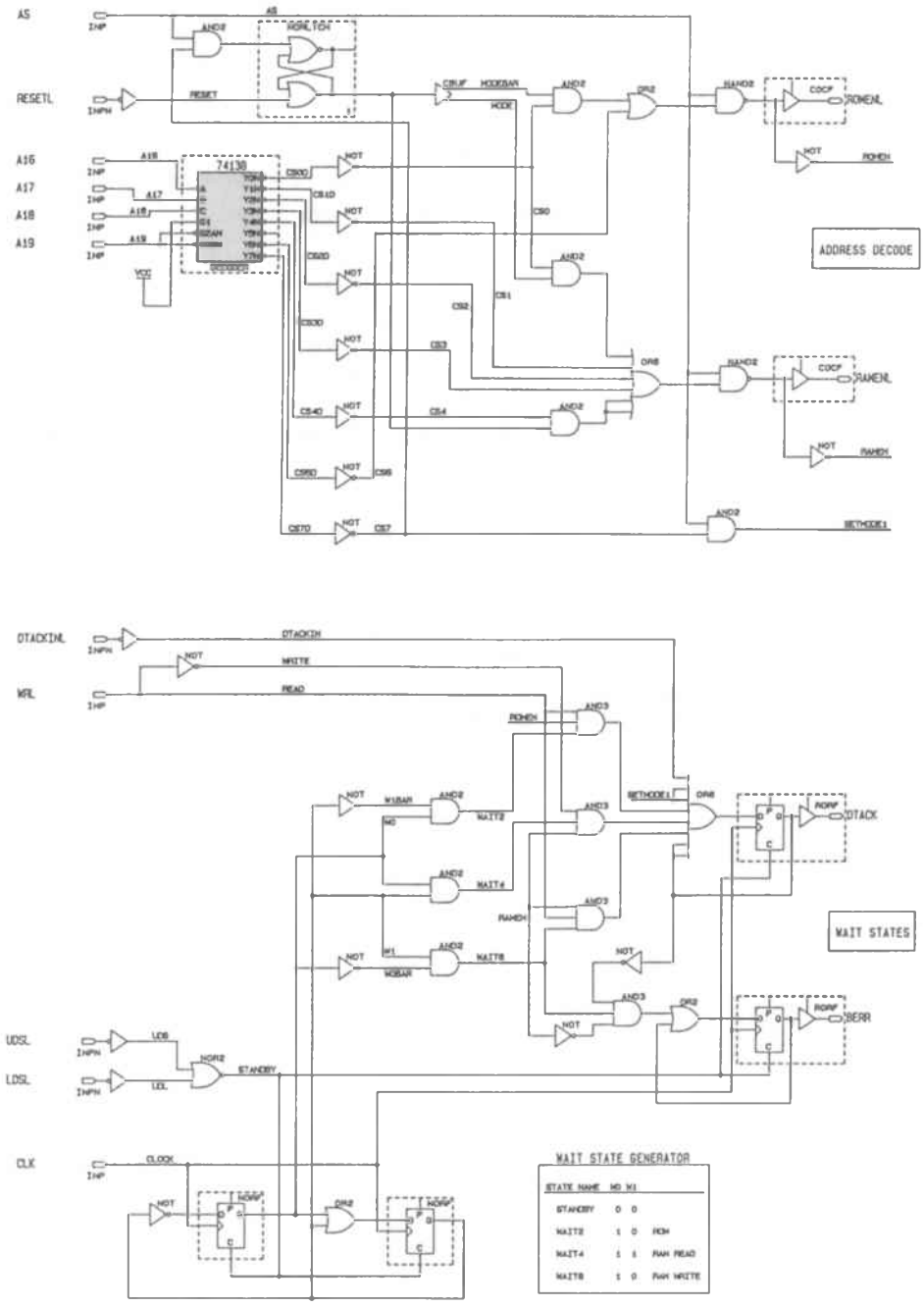


Figure 6. EP310 Address Decode and Wait State Logic—*Logic Schematic for the EP310.*



FEATURES

- **Serial Transmitters.**
- **Serial Receivers.**
- **Error Detection Logic.**

INTRODUCTION

A large class of digital logic designs involve serial data transmission and reception (UARTs). Altera EPLDs offer a user-configurable approach to UART design. Each EPLD can be customized to fit the end system requirements thus eliminating many SSI/MSI logic devices which would otherwise be required.

This Application Note focuses on serial transmitters and receivers which convert parallel data words into a serial bit stream and reconstruct the original word from the serial bits. This note presents modular designs that can be used to create larger and more complicated communication systems.

Such complex systems include communication interfaces between a CPU and a LAN (Local Area Network). The EPLD handles the handshake and protocol requirements of the LAN as well as processing incoming data from both the CPU and LAN. Other uses include stepper motor controllers and single chip remote data acquisition systems where the EPLD controls the analog to digital conversion of the inputs as well as the serial communication.

EPLDs can implement 'custom' serial protocols for telecom and government secure programs which can not be obtained by using standard off the shelf UARTs. EPLDs are easily reconfigured, allowing the transmission sequence to be altered the same day.

SERIAL TRANSMITTERS

A serial transmitter converts a parallel input word into a serial bit stream, and appends synchronization and error detection information. A simple transmission synchronization scheme may require only a START bit before each serial word and one or more STOP bits after each word. A more common error detection method consists of a parity bit appended to each word.

MULTIPLEXER TRANSMITTER

Serial transmission may be performed with a digital 'switch' or multiplexer. Figure 1 illustrates this concept. While idle, the switch stays connected to the STOP bit. Transmission starts where the multiplexer generates a START bit by "switching" to GND, then scans the parallel inputs one at a time at a predetermined baud rate, finally ending up at the STOP bit. Figure 2 shows a block diagram of a multiplexer digital switch transmitter implemented in the EP610. Figures 3, 4, and 5 show the internal logic for each block.

Figure 1. Digital 'Switch' Transmission—*Serial transmission by sequentially routing the parallel to the serial line.*

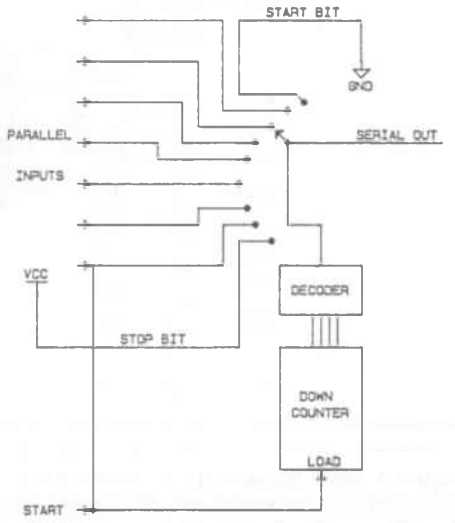


Figure 2. Multiplexer Transmitter Block Diagram—*Block diagram of the 8 bit multiplexing transmitter shown in Figures 3, 4, and 5.*

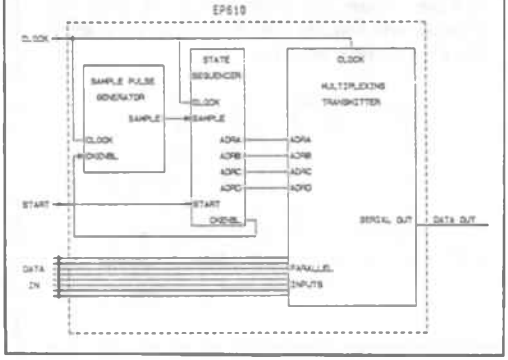


Figure 3. Multiplexing Transmitter—*The multiplexing transmitter has the advantage of only one output register, minimizing resource consumption but requiring stable inputs for the duration of the transmission.*

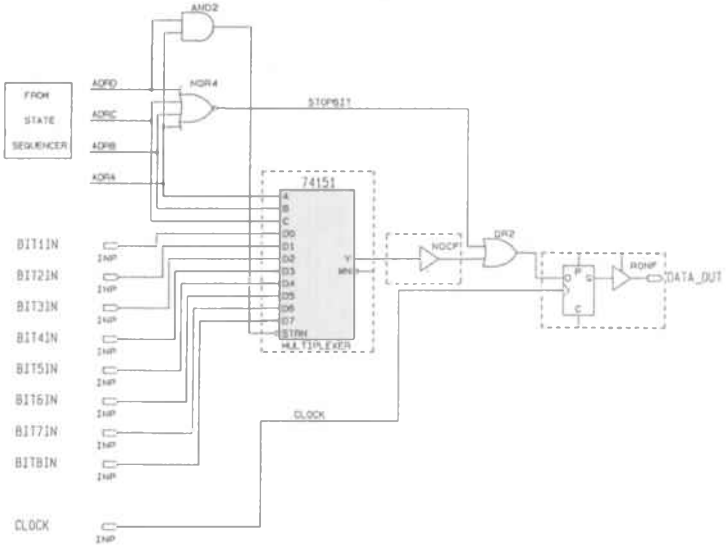


Figure 4. State Sequencer—*A down counter tracks the bit that is transmitted or received. When the transmission or reception sequence starts, the counter loads a binary nine (for eight data bits and a stop bit); the counter decrements to zero, the idle state, and stops.*

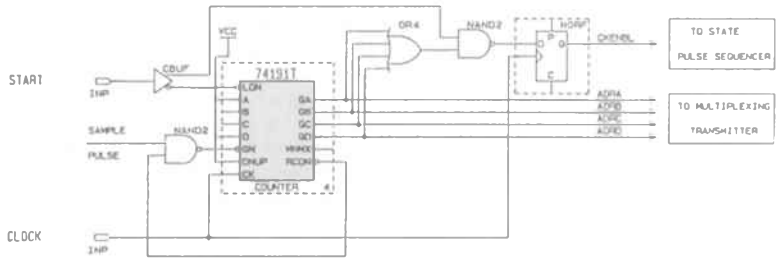
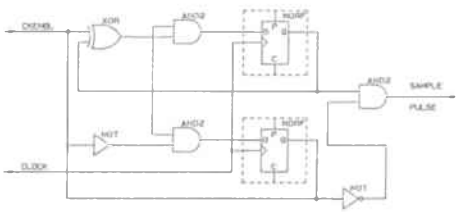


Figure 5. Sample Pulse Generator—*The sample pulse generator ensures that the receiver will sample the serial input only when the input has stabilized.*

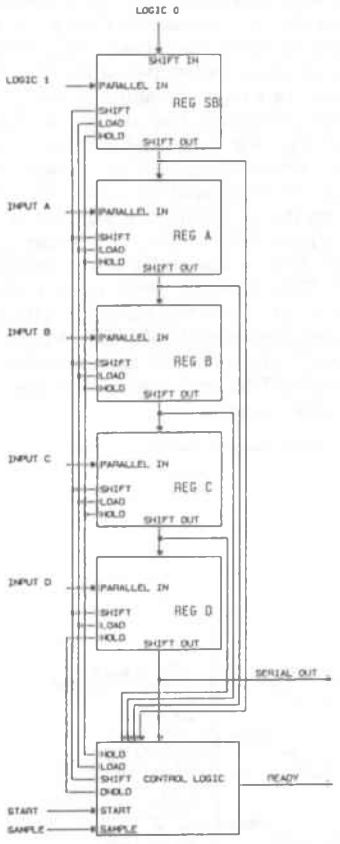


SHIFT REGISTER TRANSMITTER

For wide data inputs, a single EPLD may not contain the product term resources necessary to implement the multiplexing logic. The logic may double for each additional input bit. In this case, a shift register transmitter design becomes viable, since each additional bit requires only one extra register with the accompanying data select logic. Figure 6 shows a block diagram of a shift register transmitter. In this design, the shift register operates as a state machine with the current state being a function of the shift register's outputs. The logic schematic is shown in Figure 7.

When idle, all registers contain logic "0", with the exception of the serial output register (SEROUT) which is a logic "1". An asserted START bit resets

Figure 6. Shift Register Transmitter Block Diagram



the serial output register. The next SAMPLE pulse transmits a logic "0" start bit. SAMPLE controls the serial data output rate, shifting each bit out at one-fourth the state machine's clock rate. The next SAMPLE pulse loads the shift register with the parallel input data, and sets the stop bit register to logic "1". The stop bit will shift with the input data at every SAMPLE pulse. While the databits shift out, "0's" are shifted in; eventually, the "0's" propagate through the shift register until the control logic detects that SEROUT contains the stop bit and all other registers contain "0's". At this point transmission ends.

SERIAL RECEIVERS

Once a transmitter sends data, a serial receiver must detect the START bit, then decode and convert the serial input into a parallel word. For each input bit, transmission line parasitic inductance and capacitance, as well as the differing transmitter and receiver clock rates may adversely affect input rise and fall edges. Thus the receiver should sample data well after the rising edge and well before the falling edge to insure the data sample is valid. Typically mid-bit sampling is the accepted approach.

DEMULTIPLEXER RECEIVER

Similar to the multiplexing transmitter, a demultiplexing receiver, shown in Figure 8 detects the start bit, loads the input data counter with a predetermined value, decodes, then stores the input data. The counter decrements every four clock cycles to insure that the data sampled is not part of the rising or falling edge and is indeed stable data. Figure 9 shows the logic representation of a demultiplexing receiver.

Figure 7. Shift Register Transmitter Logic Schematic

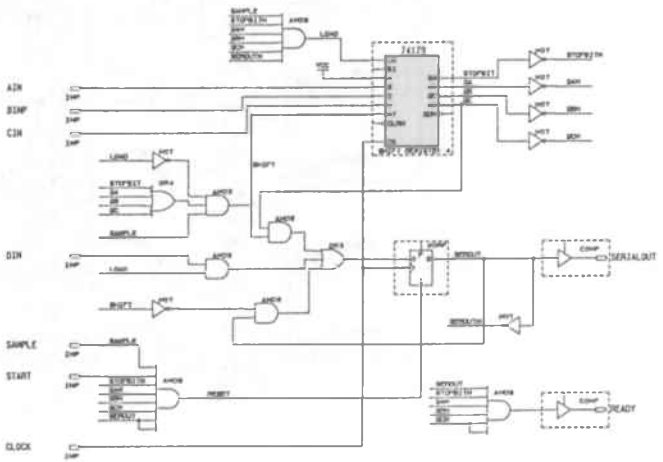
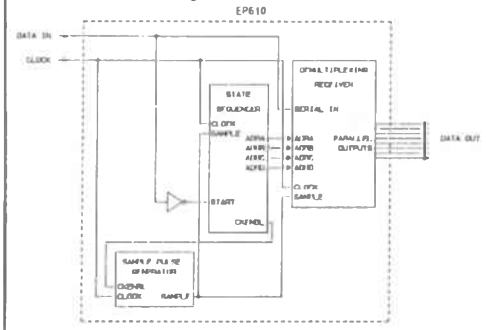


Figure 8. Demultiplexing Receiver Block Diagram—*Block diagram of an 8 bit Demultiplexing receiver. The schematics for the state sequencer and the sample pulse generator are shown in Figures 4 and 5.*



SHIFT REGISTER RECEIVER

A shift register receiver also decodes the serial input into a parallel word. The block diagram and logic schematic are shown in Figure 10 and 11. When in the idle state, the receiver output registers retain the previous transmitted word. The READY register remains a logic "1", and the timing counter (PHASE1 with PHASE2) stay at logic "0". Detection of a "0" on the serial input line is assumed to be a START bit. When the START bit is received, the timing counter is allowed to begin counting, the READY register is brought LOW, and all other registers are set HIGH. Two clock pulses later the START bit, still on the serial line, is shifted into register A. Data from the serial input is shifted in every four clock cycles thereafter, until the START bit propagates down to the READY register. When this occurs, the circuit again becomes idle, awaiting a new START bit, with the newly received data on the parallel outputs.

Figure 9. Demultiplexing Receiver Logic Schematic

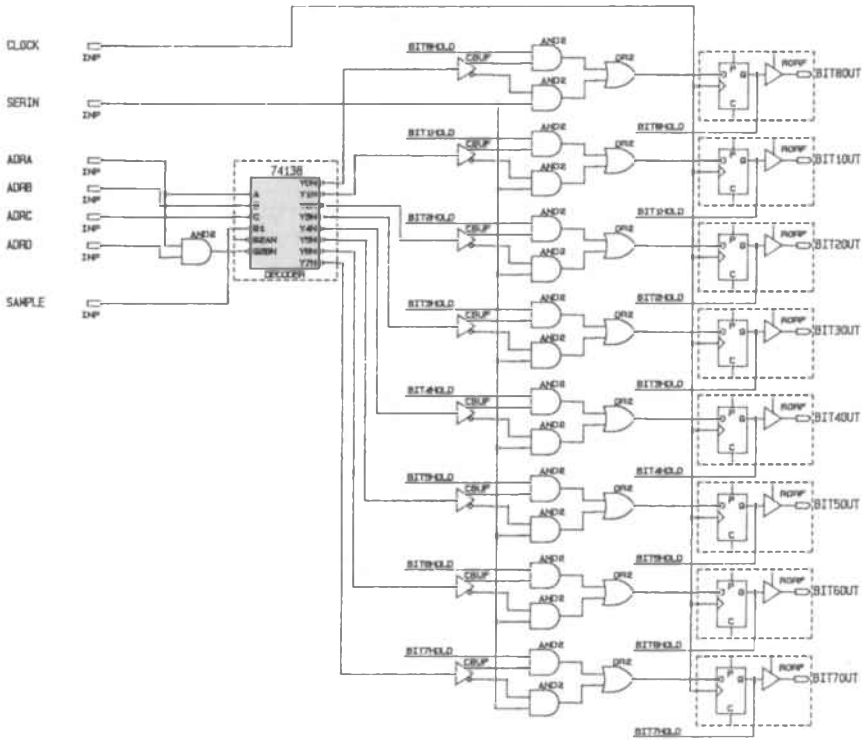
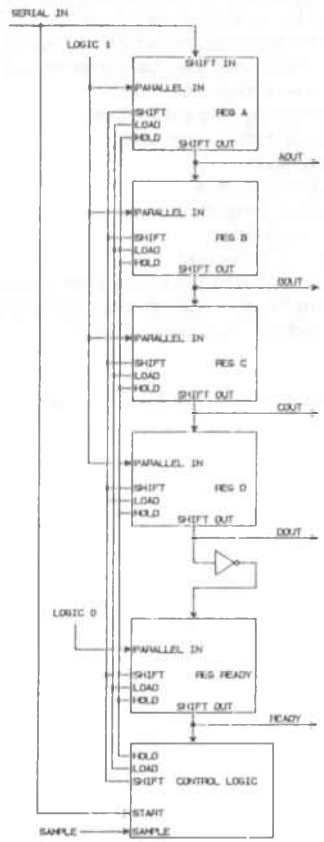


Figure 10. Shift Register Receiver Block Diagram



ADDING PARITY

A transmitted data word with an appended parity bit represents a simple transmission error detection scheme. By definition, a parity bit reveals whether the data word has an even or odd number of ones (or zeros). A simple way to implement parity is with a modulo two counter which toggles each time a logic "1" or "0" is transmitted or received.

Figure 12 shows a 12 bit serial transmitter implemented in the EP910. The ODD/EVEN input controls the parity checking sense. A logic "1" ODD/EVEN signifies 1's checking while a logic "0" ODD/EVEN enables 0's parity check. For example, if ODD/EVEN is HIGH, each time a "1" is transmitted the counter will toggle. After all data bits have been sent, the parity bit is sent, indicating whether an even or odd number of 1's have been transmitted.

For a serial receiver, a parity counter is also used for the incoming data. Once the STOP bit arrives, results of the parity counter and the transmitted parity bit are compared. Figure 13 shows a 12 bit receiver with parity check implemented in the EP910. Figure 14 show the parity check logic schematic.

Figure 11. Shift Register Receiver Logic Schematic

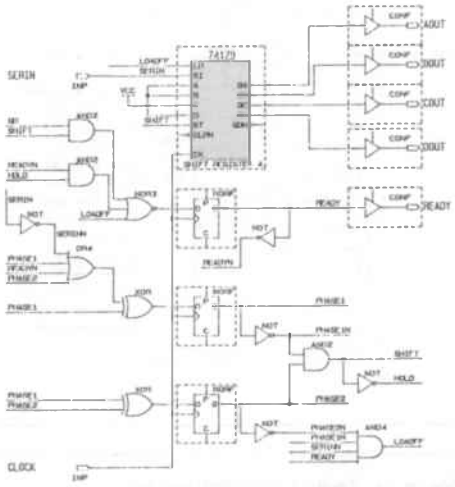


Figure 12. Serial Transmitter with Parity

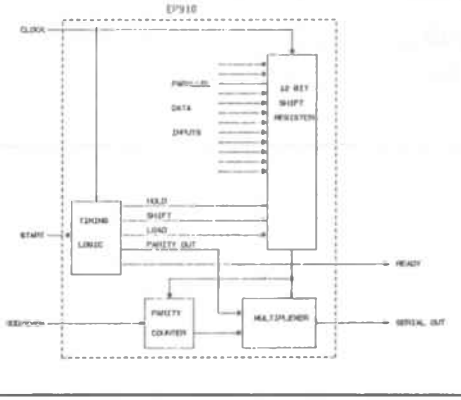


Figure 13. Serial Receiver with Parity

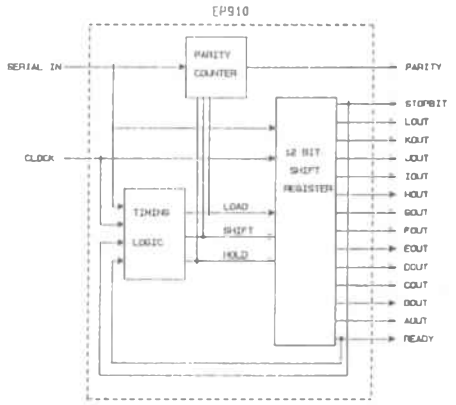
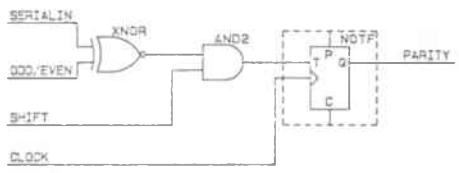


Figure 14. Parity Logic Schematic



CONCLUSION

These designs are basic building blocks which may be used for custom UART design with EPLDs. In addition to the transmitter and receiver logic, addition functions may also be implemented within the EPLD. These include:

Double buffering of parallel data
Cyclic Redundancy Checking
Error Correction
Latching Parity errors
Framing error detection
Overrun detection
False start bit detection

By combining these functions with the receiver and transmitter logic, it is possible to integrate an entire sub-system on a single EPLD.

AN6 Rev 2.0

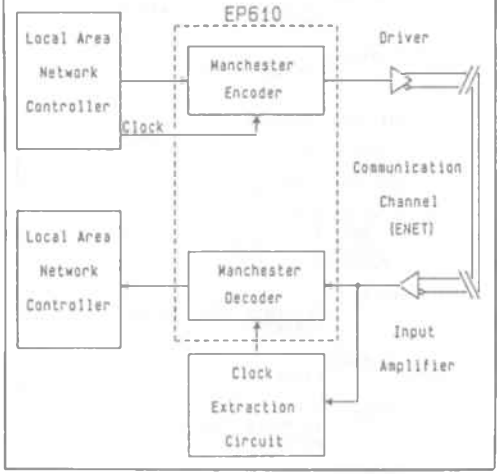
FEATURES

- **Local Area Network (LAN) component.**
- **Multiple State Machines for a single EP610.**

INTRODUCTION

Manchester Encoders and Decoders encode and decode serial binary data transmitted between synchronous communication systems with point-to-point or common bus architectures. For example, in a Local Area Network (LAN) system, a Manchester Encoder sends serial Non Return to Zero (NRZ) data packets to the Manchester Decoder via an Ethernet cable (see Figure 1).

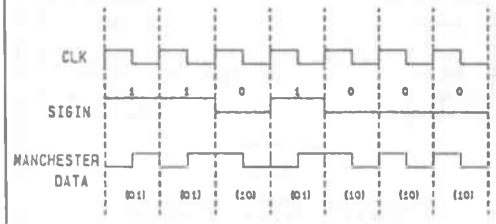
Figure 1. Local Area Network with Manchester Data Circuits—*Local Area Network communication systems employ Manchester encoded serial data in the communication channel. This permits the data to carry its own synchronization clock, and allows continuity over an AC coupled system.*



ENCODER

The Encoder converts the serial data and clock inputs into a Manchester pattern. The standard Manchester II bit stream format is "10" and "01", representing logic levels, "0" and "1", respectively, as shown in Figure 2. For each data bit, the Encoder generates two resultant bits; consequently, the Encoder operates at twice the data input rate. Before the Manchester circuit sends a data packet, it transmits a valid pulse marking the Manchester encoded data as good.

Figure 2. Functional Simulation for Manchester Encoder—*Example of a Manchester Encoder's serial input and encrypted output. The Encoder's internal clock must operate at twice the frequency of the incoming data's clock.*



DECODER

Once the Manchester Decoder obtains valid data, an external phase locked loop circuit extracts a clock signal from the Manchester encoded data, synchronizing the decoder; the Decoder then deciphers the transmitted data, restoring the original NRZ serial pattern.

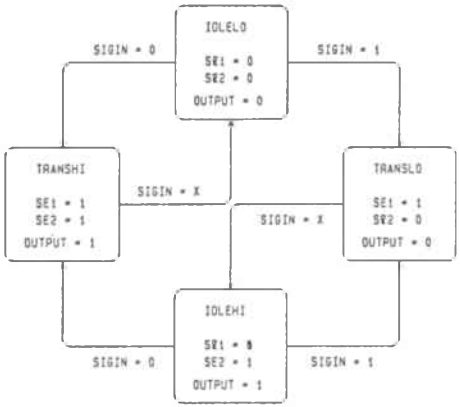
LOGIC DEFINITION

The state diagrams, shown in Figure 3a and 3b, represent the Encoder and Decoder's encryption and decryption routines. On power-up, the Encoder and Decoder reside in their respective states, IDLELO and PRELO, and the output data is a logic "0". When the Encoder input, SIGIN, assumes logic "1", the Encoder enters state TRANSLO, transmitting a valid "0". In the next clock period, the Encoder unconditionally enters state IDLEHI, sending a logic "1". The Decoder remains idle until it receives a valid pulse. External logic performs transmission data validity checks, stopping the decode cycle, until the Decoder receives a valid MANIN. When MANIN arrives, the external phase lock device starts the decode process; the Decoder generates high level output, the correct result, after it reads a Manchester "01".

Figure 4 illustrates the Altera State Machine Design file obtained from the state diagrams shown in Figures 3a and 3b. Both machines are described within one source file (SMF file). The design requires a total of four macrocells. Thus, only a small portion of the EP610 is required to implement both the Encoder and Decoder. The remaining EP610 resources may be used to implement the Encoder's valid generate and the Decoder's valid check circuits.

Figure 3. Encoder State Diagram and Table

a. Manchester Encoder's state machine diagram for the encryption of serial data input into a Manchester II data format.



Decoder State Diagram and Table

b. Manchester Decoder's state machine diagram for the decryption of Manchester II encoded data into original NRZ data format.

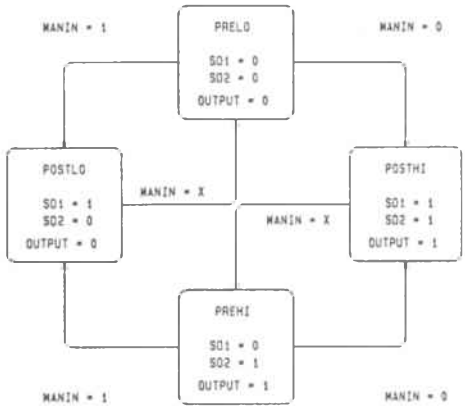


Figure 4. Manchester Encoder/Decoder State Machine Files

a. SMF file for the Manchester Encoder with serial input, SIGIN, and serial output, SE2.

b. SMF file for the Manchester Decoder with serial input, MANIN, and serial output, SD2.

```
Altera Corp

3-1-88
Rev B

Manchester Encod/Decod

PART:EP610

INPUTS: CLK, SIGIN, MANIN
OUTPUTS: SE2, SD2

MACHINE: ENCODER

CLOCK: CLK

STATES: [SE1 SE2]

IDLELO  [ 0   0 ]
IDLEHI  [ 0   1 ]
TRANSLO [ 1   0 ]
TRANSHI [ 1   1 ]

IDLELO:
        IF SIGIN THEN TRANSLO
        TRANSHI

TRANSLO:
        IDLEHI

 IDLEHI:
        IF SIGIN THEN TRANSLO
        TRANSHI

 TRANSHI:
        IDLELO


MACHINE: DECODER

CLOCK: CLK

STATES: [SD1 SD2]

  PRELO [ 0   0 ]
  PREHI [ 0   1 ]
 POSTLO [ 1   0 ]
 POSTHI [ 1   1 ]

PRELO:
        IF MANIN THEN POSTLO
        POSTHI

 POSTHI:
        PREHI

PREHI:
        IF MANIN THEN POSTLO
        POSTHI

POSTLO:
        PRELO

END$
```

AB3 Rev 2.0

FEATURES

- **Time-multiplexed 24 channel 8 bit data transmitter meeting telecommunications T1 carrier standards.**
- **Generates output as 193 bit serial data stream in T-1, D2, D3 or D4 Mode 3 data format.**
- **Accepts 8 bits of parallel data as inputs.**
- **Provides channel and frame synchronization timing signals.**
- **Provides automatic bit insertion for "All Zero" channel samples.**
- **Provides current bit, frame, and channel selection information.**
- **Provides both binary and paired unipolar outputs.**
- **Provides alternate control for alarm reporting and signaling.**

INTRODUCTION

In the early 1960's, the Bell System introduced a pulse code modulation system for digital voice communication. This system, called T-1, has since formed the basis for most terrestrial and satellite digital voice communication. In the T-1 system, 24 channels of data are time-multiplexed and coded into a 1.544 Mhz carrier frequency. As shown in Figure 1, each channel consists of 8 bits of serial data. The 24 channels constitute a Frame, and there are 12 Frames per Master Frame. Framing bits are inserted at the beginning of each Frame, and signaling bits are inserted as the eighth bit of each channel in Frame 6 and Frame 12.

The EP1210 is the logical choice for implementing a single chip T-1 transmitter. The EP1210 has input latches for storing parallel data bits. The 1210 also has variable product term distribution, allowing large amounts of combinatorial logic to be implemented in one macrocell.

FUNCTIONAL DESCRIPTION

Figure 2 is a functional block diagram of the transmitter. A 1.544 Mhz clock runs a counter for generation of bit and channel timing, and a seperate counter is used for frame timing. These counters may be externally synchronized by use of the SYNCIN and FRSYNC inputs. The outputs of these counters control the data selector.

All inputs are latched when the eighth bit of any channel is being transmitted. The Data Selector, controlled by the bit counter and the frame counter, outputs the proper sequence of bits.

The Zero Channel Monitor function causes Bit 8 to be transmitted as a "one" if the channel data sample is all 'zeros.' If a zero is detected in Frame 6 or Frame 12, this "ones stuffing" is done in Bit 7.

Figure 1. T-1 Transmission Protocol—*The T-1 protocol includes automatic framing bit insertion, and signaling bits in a 24 channel, 8 bit per channel serial format.*

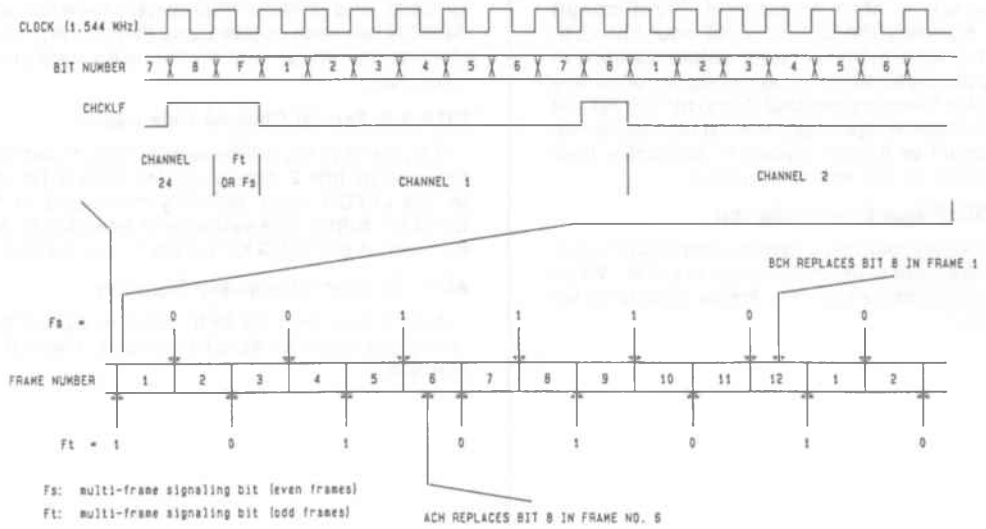
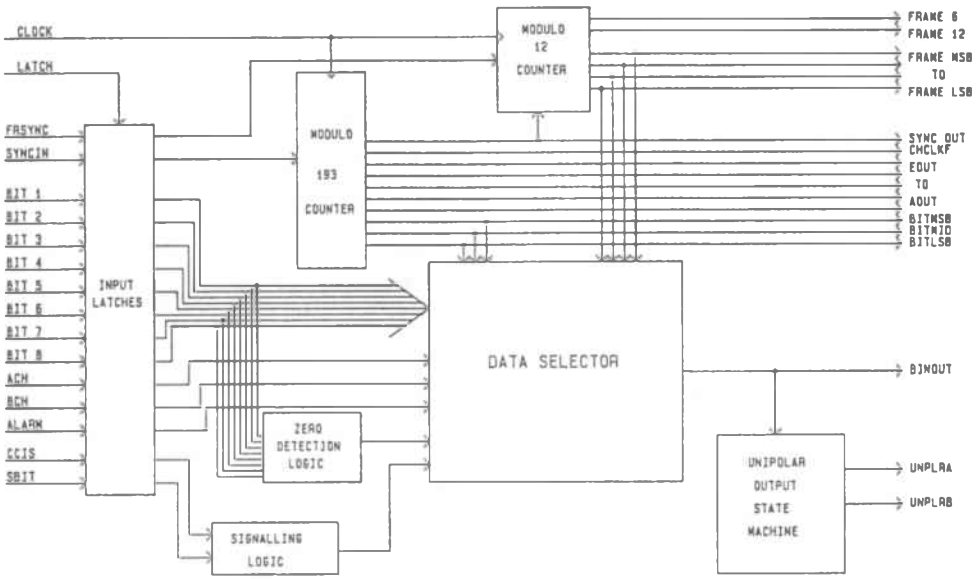


Figure 2. T-1 Serial Transmitter Block Diagram—*Shown are the functional blocks needed to implement a T-1 Serial Transmitter. The Data Selector takes information from the counters, signaling, and zero detection logic, and sends out a serial signal formatted to T-1 D2 or D3 specifications.*



T-1 TRANSMITTER INPUTS

All inputs except the CLOCK inputs pass through transparent latches that are controlled by the LATCH input. When the LATCH input is high, the latches are in their transparent (flowthrough) mode. Normally the LATCH input would be connected directly to the CHCLKF output resulting in the inputs being sampled as the eighth bit of any channel is being transmitted. Note, for this reason synchronization input signals must be asserted for a minimum of 9 clock cycles to guarantee their recognition by the internal counters.

FRSYNC: Frame Synchronization

Frame Sync permits external synchronization of the transmitter's internal frame counter. When FRSYNC becomes high, the Frame Counter is set to frame 1.

SYNCIN: Synchronization Input

SYNCIN permits external synchronization of the modulo 193 bit/channel counter. When SYNCIN becomes high, the counter is set to the state corresponding to the output of the Framing Bit. CHCLKF and SYNOUT will remain asserted until SYNCIN has been negated, and then the first bit of channel one will be sent on the rising edge of the next clock.

BITS 1-8: Parallel Channel Data Inputs

Bit1, the sign bit, will be serially transmitted first, followed by bits 2 through 8. The data is latched by the LATCH input, normally connected to the CHCLKF output. The data should be stable for the duration of the CHCLKF pulses to avoid errors.

ACH: "A" Channel Highway Signaling

ACH allows the user to transmit one bit of signaling per channel as bit 8 of each channel in Frame 6.

ACH: "B" Channel Highway Signaling

BCH allows the user to transmit one bit of signaling per channel as Bit 8 of each channel in Frame 12 only.

CCIS: Common Channel Interoffice Signaling Strap

Provides optional control for substitution of the SBIT in place of the automatic Multiframe Signaling Bit (Fs). When CCIS is held high, it creates a 4-kilobit common channel signaling path, the Fs bit is replaced with the SBIT input, and the insertion of ACH and BCH is suspended. The CCIS bit input may also be used to provide an alternate method of alarm reporting.

SBIT: Multiframe Signaling Bit

SBIT, in conjunction with CCIS, provides an alternate way to control the Multiframe Signaling Bit (Fs) transmission. The SBIT input is transmitted as the Fs bit if CCIS is held high.

LATCH: Input Latch Enable

Enables the transparent latches on all data and synchronization inputs. When LATCH is at a logic high, all latches are in their transparent mode. LATCH would normally be connected directly to the CHCLKF output resulting in the inputs being sampled every 8 clock cycles, or 9 when the frame bit is transmitted. If it is desirable to be able to synchronize the internal counters in only one clock cycle, LATCH must held high while asserting FRSYNC and SYNCIN.

CLOCK: T-1 Clock

The T-1 serial bit period is bounded by the rising edges of this clock. If it is desirable to use the falling edge, a NOT gate could be inserted in the schematic on the T-1 clock line. Normal T-1 Frequency is 1.544 Mhz, but the EP1210 will operate in excess of 15Mhz if desired.

T-1 TRANSMITTER OUTPUTS

AOUT-EOUT: Channel Number

These are the five most significant bits of the modulo 193 counter. They identify which channel is currently being transmitted.

BITMSB, BITMID, BITLSB: Bit Number

These are the three least significant bits of the modulo 193 counter. They identify which bit of a channel is next to be transmitted.

CHCLKF: Channel Clock

This signal can be used for external synchronization. It is high for one clock period during the tranmission of the eighth bit of any channel, or during the tranmission of a framing bit. This output will also be high when SYNCIN is held high.

SYNOUT: Channel Synchronization Output.

SYNOUT provides a means for external synchronization. SYNOUT is high for one bit period during the transmission of the framing bits, and is also high if SYNCIN is high.

FRMMSB, FRMMMID, FRMLMID, FRMLSB: Frame Number

These are the outputs of the Frame Counter, signifying which frame is being transmitted.

FRAME6, FRAME12: Signaling and Synchronization Frames

These outputs are at a logic high during the transmission of frame 6 (FRAME6 output) and frame 12 (FRAME12 output). These can be used signify when bit of each channel is being replaced by ACH (in frame 6) and BCH (in frame 12).

ZERODET: Zero Channel Detection

A logic high on this output indicates that the data to be sent is composed of all zeros, and that automatic bit insertion is in effect. A one will be placed in bit 8, except in frames 6 and 12, when the one will be placed in bit 7.

BINOUT: Serial Data Format

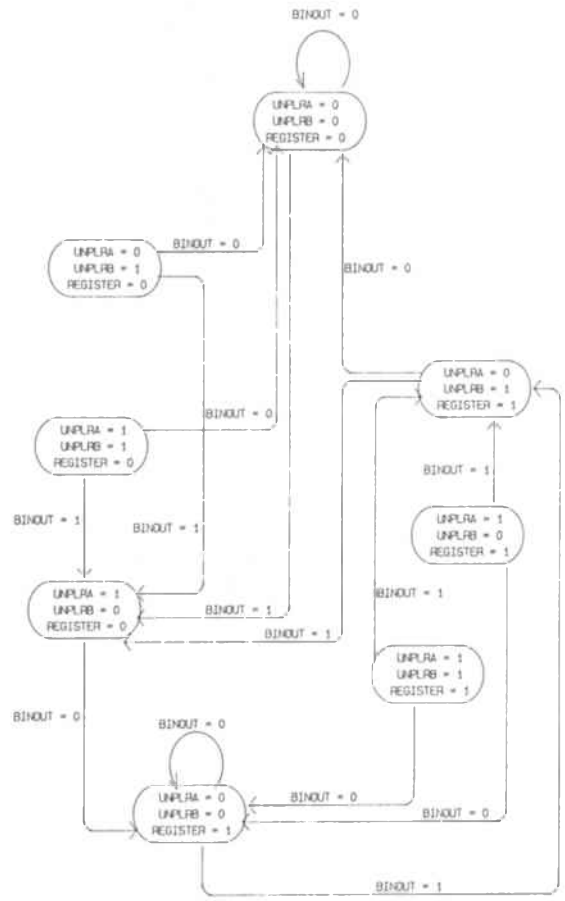
BINOUT is the binary serial output of the parallel input data and the signaling data. Binout is controlled by the internal counters, and is delayed by one clock cycle from their states.

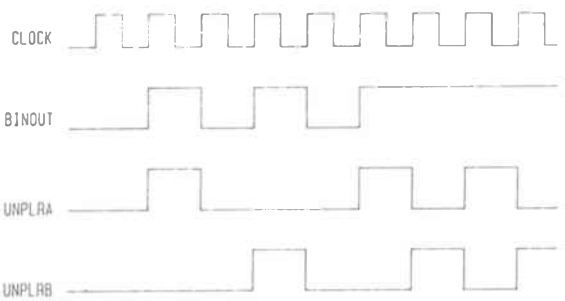
UNPLRA, UNPLRB: T-1 Serial Data Unipolar Outputs

These two paired unipolar outputs are provided for the purpose of creating a single bipolar serial data transmission. The two outputs are both at a logic low for a zero on the BINOUT output, and are complements of each other for a one. As per the T-1 spec., for each "one" that is transmitted, the sense of the two outputs are reversed. This section of the design has been implemented using a state machine. The bubble diagram and functional waveform are shown in Figure 3.

AB4 Rev 2.0

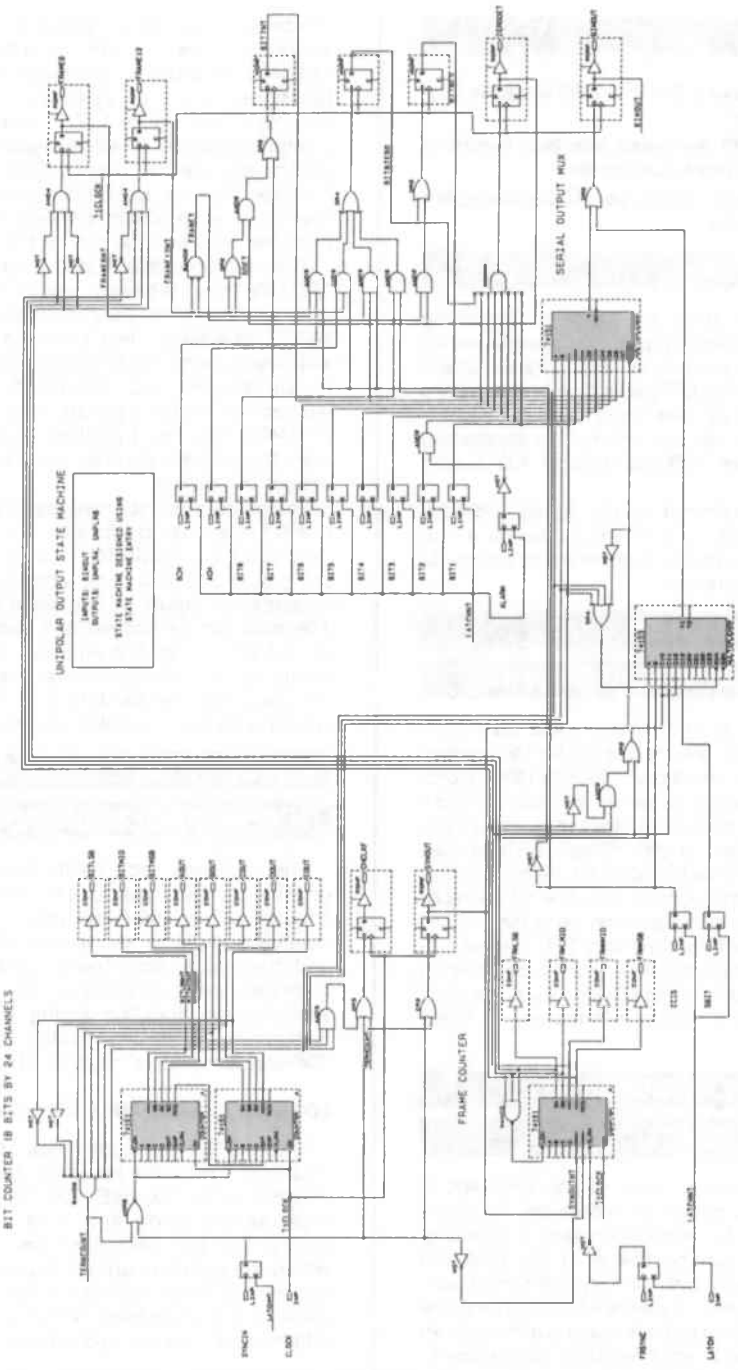
Figure 3. Timing and State Diagram for Unipolar Output—*The State Diagram for the Unipolar Output is implemented in the design using state machine design entry.*





NOTE: UNPLRA AND UNPLRB ARE DELAYED BY ONE CLOCK CYCLE

Figure 4. Schematic of T-1 Serial Transmitter—*This schematic is entered using the LogiCaps schematic capture program. The schematic then is processed using A+PLUS and programmed into an EP1210 to implement a single chip T-1 Serial Transmitter.*



FEATURES

- **Guidelines on using the EPB1400 dedicated interface functions.**
- **Linking EPB1400 dedicated interface functions and general purpose macrocells.**
- **Timing analysis of circuits containing dedicated interface functions.**

INTRODUCTION

This Application Brief is a guide for designing with the EPB1400 (Buster) user configurable microprocessor (MPU) peripheral. Both the programmable logic core and MPU interface functions are described. Examples illustrating the interconnection of these resources are shown. This application brief also discusses EPB1400 internal A.C. timing relationships.

The reader is referred to the Altera EPB1400 Data Sheet for details concerning device architecture and performance. A general knowledge of the EPB1400 is assumed.

USING THE EPB1400 PROGRAMMABLE LOGIC CORE

The EPB1400's programmable logic core contains 20 general purpose macrocells for the integration of custom logic functions. The EPB1400 macrocell structure offers a superset of features when compared to the macrocell structures of general purpose EPLDs (eg. EP600, EP900, EP1800. See the EPB1400 Data Sheet for details). When entering logic within the macrocells, use the same techniques that apply to the general purpose EPLDs: Logicaps schematic capture with TTL macrofunctions, high-level state machine syntax, or Boolean equation entry. Refer to the "Design Entry" section of this Handbook for more information on these entry methods.

USING MICROPROCESSOR INTERFACE FUNCTIONS

The general purpose core of the EPB1400 is surrounded by a series of byte-wide functions used specifically for microprocessor interfacing. These functions can be linked to the EPB1400 programmable logic core for fully integrated peripheral solutions. These byte-wide functions include 2 input registers, 2 output latches and a transceiver port. Note that the 2 input registers can be configured as edge-triggered or flow through latches. D-inputs to the input registers may come from actual I/O pins or from the internal bus of the EPB1400. In order to designate the use of these functions, a set of symbols is provided with Logicaps; see Figure 1. For example, the RINP8 symbol designates an edge-triggered input register which receives its D-input data from 8 General Purpose I/O pins. LBUSI designates a level sensitive input latch which receives its D-input data from the 8-bit internal bus of the EPB1400.

The microprocessor may write data into the EPB1400 input registers, which in turn may write data to the general purpose macrocells. During a WRITE operation, data from the microprocessor will pass through the EPB1400 bus port transceiver to the internal bus. Internal bus data may be latched into input registers on either side of the EPB1400. The input register Q outputs can then feed any general purpose macrocell on the same side of the device.

The microprocessor may read data from the general purpose macrocells after it is passed through the output latch (LBUSO). Macrocell outputs must be grouped into a byte and connected to the output latch inputs on the same side of the chip. The data can be latched and passed through the output latch onto the internal bus. By changing the direction of the transceiver, the microprocessor can pass the internal bus data to the microprocessor data bus, completing the read operation.

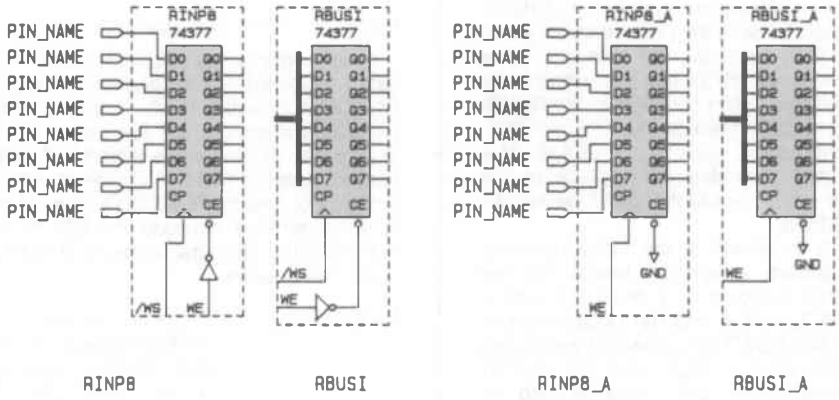
LINKING LOGIC CORE AND MICROPROCESSOR INTERFACE

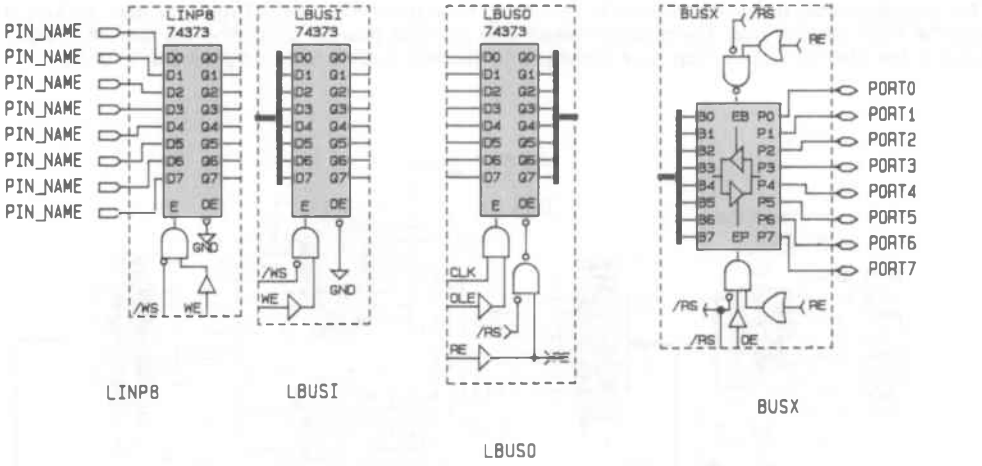
The following sections illustrate the linking of general purpose macrocells with the dedicated microprocessor interface functions. Each sample configuration is shown, its operation is described, and comments about the circuit's timing requirements are made. Readers unfamiliar with the conventions for modeling timing in EPLDs should refer to the article on timing simulation in the "Development Tools" section of this Handbook.

LOADABLE, READABLE UP/DOWN COUNTER

Figure 2 shows the LogiCaps schematic used to implement an 8-bit loadable, readable up/down counter within the EPB1400. The actual counter, implemented using a pair of cascaded 74191T binary counter macrofunctions, is implemented within the general purpose macrocells. The "T" in the symbol name represents the use of T-flipflops within the counter logic, which provides optimum efficiency in counter applications.

Figure 1. EPB1400 Microprocessor Interface Functions





The counter is "loadable" from the microprocessor data bus. The load operation is accomplished by asserting a desired preload count value on the counter data load inputs (74191T inputs A, B, C, and D) and setting LOADN low prior to the rising edge of CNTCLOCK. In this case the preload count value is stored in one of the dedicated microprocessor interface input registers: an RBUSI. The microprocessor loads the preload count value into the RBUSI by passing the desired value through the dedicated BUSX bus port transceiver onto the internal bus, which connects to the RBUSI D-inputs. The RBUSI is then clocked by the $\overline{CWS}$, which stores the count value in the RBUSI. If a flow-through latch was desired in place of the edge triggered latch, an LBUSI could be substituted for the RBUSI.

The counter is "readable" by the microprocessor. The counter outputs (macrocell feedbacks) are connected to the D-inputs of a dedicated output latch function (LBUSO). The current counter value flows through the LBUSO output latch by enabling the output latch enable (high) and the CLOCK (high). The stored count value is passed onto the internal bus when the READ ENABLE signal is high. The microprocessor completes the READ operation when OUTPUT ENABLE is high, and $\overline{CRS}$ (Read Strobe) is low.

One relevant timing path is the time it takes for microprocessor data bus values to become valid on the internal bus. When $\overline{CRS}$ is low, the BUSX bus port transceiver is driving off the device. In order for the bus to "turn around" the $\overline{CRS}$ signal must go high, and propagate through to the BUSX function. The amount of time required is:

$$t_{\overline{CRS}\text{ to I.B.}} = tin + t_{PZX\text{ PORT to I.B. [BUSX]}}$$

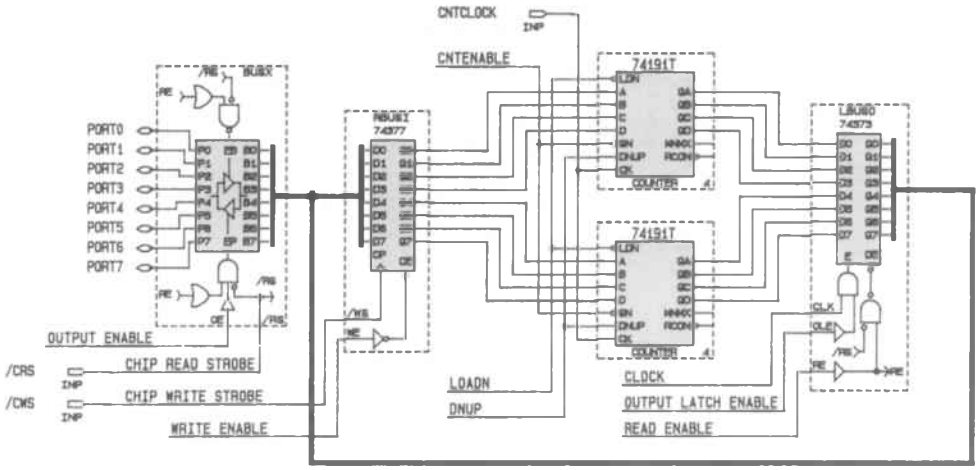
Another relevant timing path is the setup time of data just stored in the RBUSI input register to the CNTCLOCK which loads it into the counter. If CNTCLOCK occurs too soon after the RBUSI is loaded, the desired value stored in RBUSI will not have sufficient time to reach the counter data load inputs. The time that CNTCLOCK must be delayed is the difference between the RBUSI to counter delay and the propagation delay of CNTCLOCK to reach the counter.

$$\begin{aligned} t_{\overline{CWS}\text{ to CNTCLOCK}} &= tp_{\overline{CWS}\text{ to COUNTER}} \\ &\quad - tp_{CNTCLOCK\text{ to counter}} \\ &= [tin + tp_{\overline{CWS}\text{ to Q[RBUSI]}} \\ &\quad + tlad + tsu] - [tin + tics] \\ &= tp_{\overline{CWS}\text{ to Q[RBUSI]}} + tlad + tsu + tics \end{aligned}$$

If CNTCLOCK is an asynchronous clock (a clock not connected to a dedicated clock pin) substitute timing spec tic for tics.

Figure 2. Loadable, Readable Up/Down Counter

The microprocessor can load the counter by storing data in the RBUSI input registers and clocking it into the 74191 counter pair. The microprocessor can read the counter by storing the current counter value in the LBUSO output latch, and subsequently reading it over the bus transceiver (BUSX).



ADDITIONAL I/O PORTS

In cases where an additional input or output port is needed, the EPB1400 general purpose I/O macrocells may be used. Figure 3 shows the connection of 8 INP symbols as well as 8 CONF symbols in order to implement an additional byte-wide input port and byte-wide output port. The additional 8 input bits may then act as D-input data to an EPB1400 byte-wide output latch (LBUSO), ultimately feeding data to the internal bus.

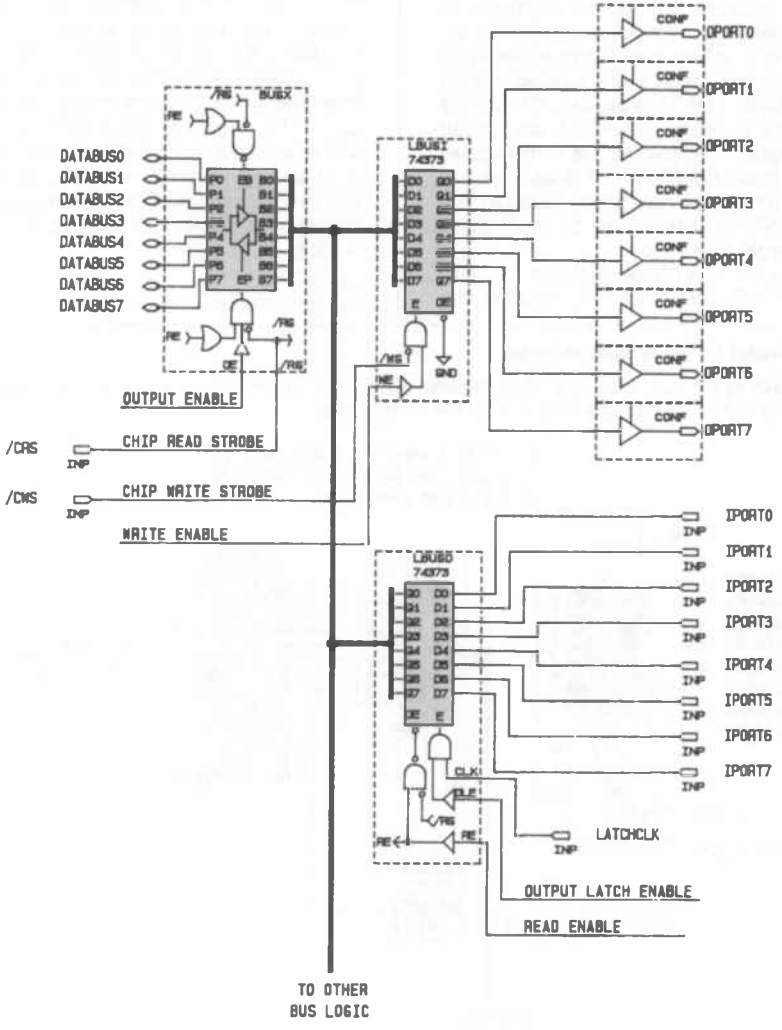
One critical timing path may be the delay from $\overline{CWS}$ going low to valid data at OPORT0- OPORT7. The delay is the sum of the time for $\overline{CWS}$ to enter the chip (tin), the time for the data to propagate through the array (tlad), and propagate to the output pin (tod).

$$t_{\overline{CWS}\text{ to OPORT}} = tin + tp_{\overline{CWS}\text{ to Q}} + tlad + tod$$

Another critical path may be the setup time of the IPORT0-IPORT7 inputs with respect to a dedi-

Figure 3. Additional Input/Output Ports

An additional output port is formed by taking LBUSI (input latch) outputs off the EPB1400 through I/O pins. An additional input port is formed by feeding an LBUSO output latch from I/O pins. Bidirectional ports may also be formed.



cated system clock (LATCHCLK). The setup time is the difference between IPORT data reaching the LBUSO, and the LATCHCLK reaching the LBUSO CLK control input, added to the default LBUSO data setup time.

$$tsu_{IPORT\ to\ LATCHCLK} = tp_{IPORT\ to\ LBUSO} - tp_{LATCHCLK\ to\ LBUSO} + tsu_{DATA\ to\ CLK[LBUSO]}$$

$$= (tin + tlad) - (tin + tics) + tsu_{DATA\ to\ CLK}$$

$$= tlad + tsu_{DATA\ to\ CLK} - tics$$

PARALLEL TO SERIAL SHIFT REGISTER

Figure 4 shows a partial schematic implementing a parallel (8 bits) to serial shift register. The byte of data to be loaded is written into the EPB1400 internal bus via the bus port (BUSX). The data is then latched from the internal bus (RBUSIA is an edge-triggered input register without the use of the WS input). When the appropriate logic levels on the control signals are applied (rising edge of SHIFTCLK, LOADN asserted, CLKINHIB not asserted), the outputs of RBUSIA are loaded into the shift register (74165). As the shift register is clocked, the most significant bit is used as the serial output. This signal may go on to a pin (by attaching a CONF) or it may also be used as an input to other logic functions.

Critical timing for this circuit includes the set-up time of the WE (Write Enable) against the SHIFTCLK. If the SHIFTCLK arrives too early, the RBUSIA data, which is enabled into the general purpose macrocells by WE, will not have sufficient time to reach the 74165 data inputs. In this instance SHIFTCLK is an asynchronous clock; it does not come from a dedicated clock pin.

$$tsu_{WE\ to\ SHIFTCLK} = tp_{WE\ to\ SHIFTER} - tp_{SHIFTCLK\ to\ SHIFTER}$$

$$= [tin + tcad + tp_{WE\ to\ Q[RBUSI\_A]} + tlad + tsu] - [tin + tic]$$

$$= tp_{WE\ to\ Q[RBUSI\_A]} + tic + tcad + tlad + tsu$$

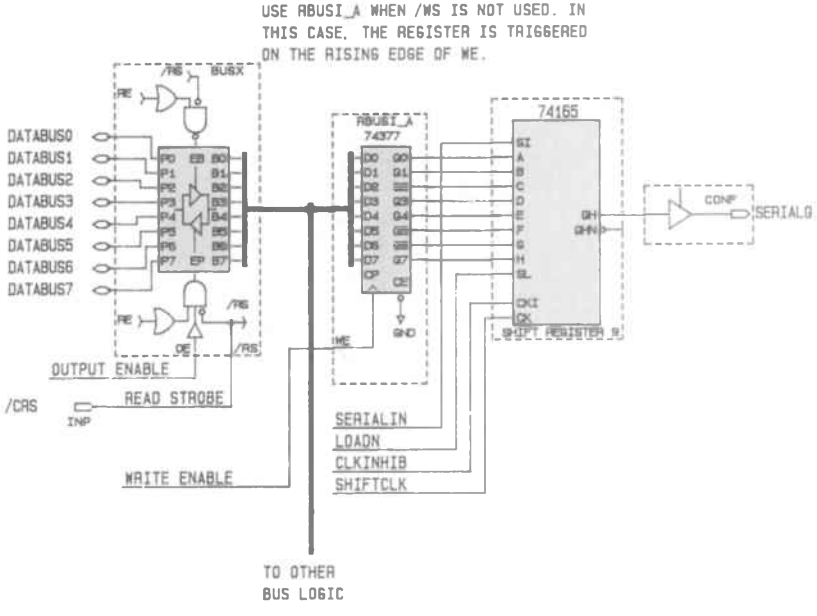
CONCLUSION

The 20 General Purpose macrocells within the logic core of the EPB1400 may be configured freely by the user to implement a variety of logical functions. Several additional configuration options are available using the byte-wide microprocessor interface functions surrounding the EPB1400 logic core. The LogiCaps Schematic Capture package and associated symbol libraries are used to specify architectural options within both the programmable logic core and the microprocessor interface functions, completing customizing of an EPB1400 device.

AB59 Rev. 1.0

Figure 4. Parallel to Serial Shift Register

The microprocessor can load the shift register with data by storing a value in the input register RBUSI_A and subsequently clocking it into the shift register.



INTRODUCTION

The EPB1400, a BUSTER (BUS I/O, regisTER intensive) user-configurable microprocessor peripheral device, is a function-specific EPLD tailored for microprocessor interface applications. This Application Note discusses common microprocessor interface requirements and describes typical EPB1400 solutions. A procedure for developing EPB1400 designs is described together with a summary of the capabilities and characteristics of the device.

A unique device programming file (the JEDEC file) is generated by an Altera development system to configure an EPB1400 for a given application (see Altera Data Book PLS2 description). Altera's PLCAD-SUPREME development system provides a complete EPB1400 design environment, including LogiCaps schematic capture with TTL macrofunction symbols, automatic design processing, functional simulation, and programming software and hardware. A complete description of Altera development system features exceeds the scope of this application note, but may be found in the Altera Data Book.

This application note should be read in conjunction with a current Altera Databook and EPB1400 Data Sheet. Familiarity with the basics of programmable logic architecture (product terms, AND/OR planes) is assumed, but specific knowledge on Altera EPLDs is not required.

THE TWO SIDES OF EPB1400 DESIGN

Two interfaces and the associated control logic must be specified in the design of a microprocessor (MPU) peripheral. The first is the MPU interface, which coordinates microprocessor bus activities, such as MPU-generated control hand-shaking and bidirectional data bus transfers. This interface is typically composed of bus transceivers, input registers, and output latches. The EPB1400 architecture contains dedicated registers and bidirectional buffers to match these MPU interface requirements.

A second interface either contains all or part of the peripheral logic, and interfaces to the peripheral sub-system. This interface has diverse logic requirements (e.g. serial or parallel transfers, synchronous or asynchronous operations, device-specific protocols) which are satisfied by the universal character of the EPB1400 programmable logic core.

MPU INTERFACE DESIGN

To implement the EPB1400/MPU interface, the MPU interface functions and their control must be selected, data bus current drive and capacitive loading effects must be analyzed, and AC timing compatability with the targeted MPU must be verified.

SELECT MPU INTERFACE FUNCTIONS

To implement the MPU interface, the EPB1400 contains both the internal bus and dedicated microprocessor interface functions, shown in Figure 1. These functions, which consist of the bus transceiver (BUSX), output latch (LBUSO), input registers (RBUSI, RBUSIA), and input latches (LBUSI, LBUSI_A), are used to write or read the MPU data bus contents into the EPB1400. The interface functions are similar to popular 74245, 74373, and 74377 octal devices. Figure 4 shows a simple EPB1400-MPU interface.

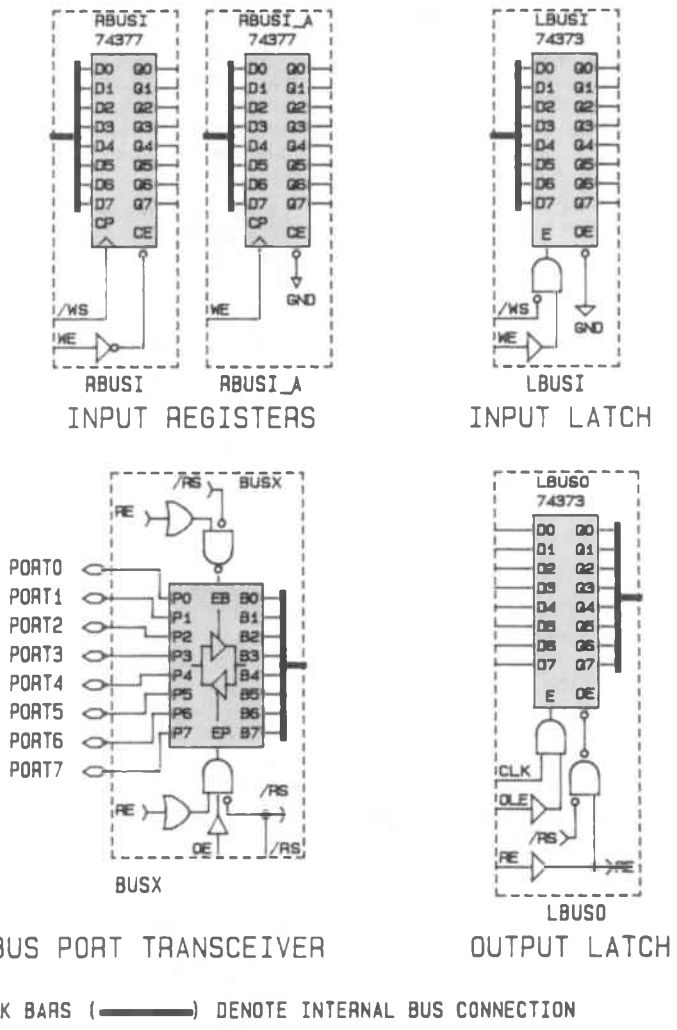
To write MPU data into the EPB1400, the MPU data bus is connected to the BUSX transceiver port inputs. Data may pass through the BUSX onto the internal bus, denoted by the dark line on the BUSX primitive (Figure 1). The internal bus acts as the D-input for the RBUSI input register. Once data is present on the internal bus, it may be clocked into RBUSI by a control signal connected to WS (Write strobe) provided WE (Write enable) is high. If a flow-through latch is desired, LBUSI is used in place of RBUSI.

For the MPU to read data from the EPB1400, the data word should first be latched in an LBUSO output latch, which is connected to the internal bus. When the LBUSO CLK (synchronous clock) is high and OLE (Output latch enable) is high, data flows through the output latch. The data is latched on the falling edge of CLK. When LBUSO $\overline{RS}$ (Read strobe) is low and RE (Read enable) is high, data flows from the output latch to the internal bus and through the BUSX transceiver onto the MPU bus. The EPB1400 bus port will tristate if the BUSX OE (output enable) control signal is low.

MPU interface control signals (strobes, clocks, address lines, and chip and output enable) are connected externally to EPB1400 input pins. These signals are then internally connected to the EPB1400 dedicated MPU interface resources

Figure 1. Dedicated Microprocessor Interface Functions

MPU interface functions are used to connect to the MPU data bus.



(BUSX, RBUSI, etc.). Dedicated read and write strobe signals (RS, WS) provide fast read or write operation. In addition, more complex control logic may be generated inside the EPB1400 control macrocells. Each control macrocell contains 2 product terms followed by an optional inversion. These logic functions may then be connected to the read enable, write enable, and output enable (WE, RE, OE) control inputs of the input register, output latch and bus transceiver port.

Systems with clock rates under 10MHz may not require the high speed response available with the RS and WS strobes, in which case these unused strobe pins may be used as general purpose inputs. RE controls output latch operation to the internal bus if RS is unused. WE controls input data clocking if the RBUSI_A function is used in place of RBUSI. If LBUSI WS is left disconnected, WE controls input latch operation.

The EPB1400 may be configured for multiple

ports, and increased data width. The EPB1400 implements up to three byte-wide ports. Multiple ports are created by feeding the LBUSO from bidirectional macrocells (Figure 2). The secondary port (P2) is read when secondary port pin data feeds the internal bus, and hence BUSX, through the LBUSO. The secondary port is written when macrocells, in this instance CO2F functions, drive the port.

Multiple EPB1400's may be cascaded to form 16 or 32 bit MPU peripherals. Distinct 8 bit segments of the MPU data bus are connected to each EPB1400 port. EPB1400 General Purpose I/O may be used to communicate between EPB1400 devices. The MPU interface design should be identical for each EPB1400 device.

MICROPROCESSOR DRIVE AND LOADING

The bus port of the EPB1400 has current drive of I_{OL} = 24mA, and input capacitance of 15pF, and is compatible with the drive requirements and loading characteristics of Board Level buses. Before using the EPB1400 in Local or Backplane bus applications, compatibility with MPU A.C. and D.C. characteristics should be carefully evaluated.

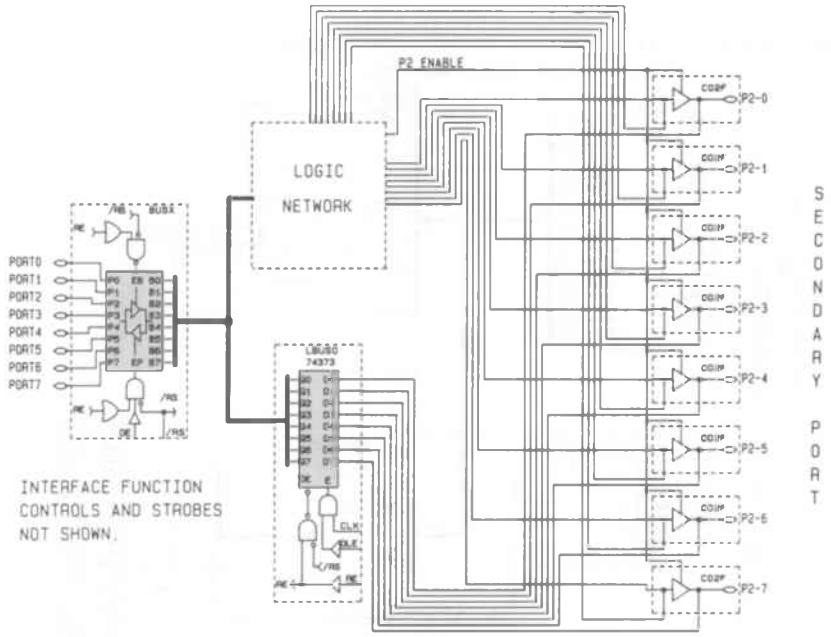
A Local bus consists of all lines connected directly to the microprocessor. Since no intermediate buffering is used, optimal performance is obtained from speed critical functions such as data buffers, address latches, coprocessors, floating point accelerators, and caches. The output drive of most microprocessors is relatively low. For example on the 8086 I_{OL} = 2.0mA; for the 68020 I_{OL} = 3.2mA. Capacitive loading of the bus must therefore be minimized to maintain system speed. Since the worst case EPB1400 bus port input capacitance is 15pF, each EPB1400 may add up to 9ns to the signal fall time under speed sensitive 2.0mA Local bus drive conditions.

Board Level buses are buffered versions of Local buses. These allow additional devices to interface indirectly to the processor. The EPB1400 is optimized for operation in this environment. The buffering typically provides 24mA drive capability, under which each EPB1400 adds an incremental

4

Figure 2.

Multiple byte-wide ports can be defined by using an LBUSO output latch to interface to the internal bus.



1.0ns to the fall time. Popular devices used to buffer the local bus include the 74LS240, 74LS245, and 74LS373. After including 5pF for stray capacitance, these devices typically present an input capacitive load of 15pF. The EPB1400 can therefore drive N such devices with rise times of (0.75)×(N) ns. This offers acceptable performance for general purpose applications.

The Backplane bus is a highly buffered version of the Board Level bus. It is used to connect seperate boards within a card cage. Although industry standard buses, such as VME, MULTIBUS, and S100 are increasing in popularity, the majority of backplane buses are still proprietary designs. The EPB1400 will drive 24mA buses, such as the IBM Micro Channel, directly. Additional buffering is required to drive standard buses which operate at 48mA.

MPU/EPB1400 TIMING COMPATABILITY

Each MPU requires customized peripherals to meet read and write cycle timing specifications; specific values depend on the MPU type and speed grade, MPU clock frequency, and buffer timing delays. The EPB1400 is fully compatible with the timing specifications of commercial microprocessors such as the 8086, 80286, and 68020 MPUs, as well as the IBM PS/2 Micro Channel bus. Following are suggested interconnect and timing analysis for the MPUs listed above, under worst case delay conditions and maximum clock rate; applications running at lower clock rates will also be compatible.

8086/8088

The 8086 is a true 16 bit MPU with an internal and external 16 bit data bus (Figure 3). Figure 4 shows the EPB1400 MPU interface used for timing analysis. The 8088 is identical to the 8086 with the exception that the external data bus is 8 bits wide. Address and data are time multiplexed on the AD0-AD16 pins; bus control signals generated by the 8288 Bus Controller, indicate when information at the pins is data or address. The 8288 Bus Controller is fed status information by the MPU at the beginning of a bus cycle.

The 8288 $\overline{\text{MEMR}}$, $\overline{\text{MEMW}}$, $\overline{\text{IORC}}$, and $\overline{\text{IOWC}}$ control signal outputs reflect the status of the bus cycle. A peripheral is said to be memory mapped

Figure 3. 8086/88 EPB1400 Interface

For a typical 8086/88 system, with latched addresses and buffered data buses, the EPB1400 connects to the lower 8 bits of the data bus.

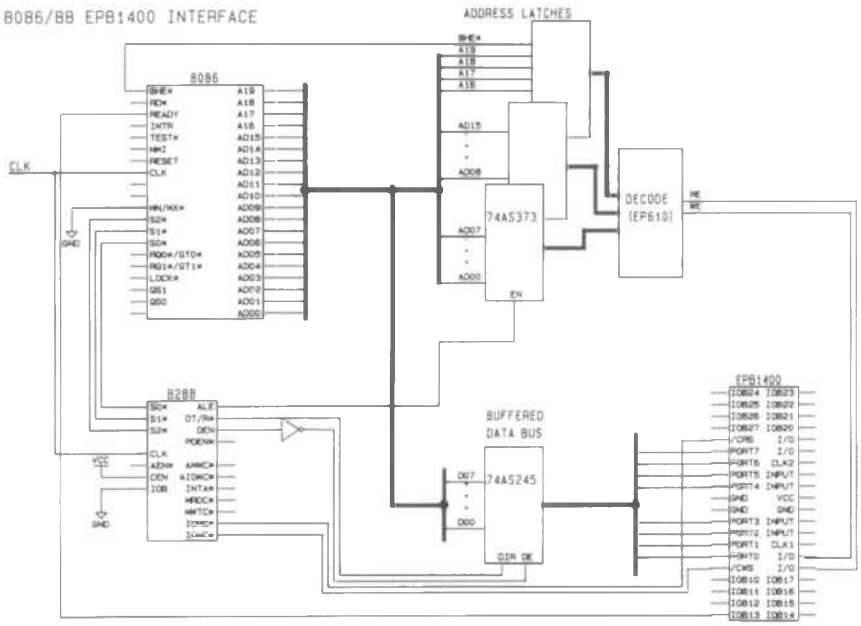
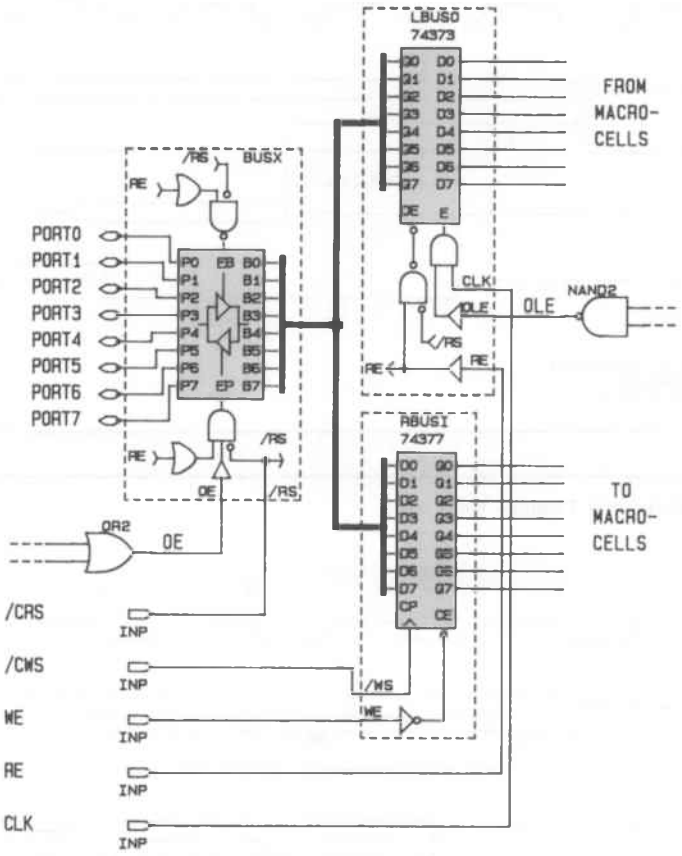


Figure 4. Simplified Microprocessor Interface

A simplified MPU interface is formed by connecting a BUSX port transceiver to a RBUSI input register and LBUSO output latch via the internal bus. This interface was used in the 8086 and 80286 timing analysis.



4

if $\overline{\text{MEMR}}$ and $\overline{\text{MEMW}}$ control signals are used to access it, or I/O mapped if $\overline{\text{IORC}}$ and $\overline{\text{IOWC}}$ are used to access it. Timing for memory mapped and I/O mapped control cycles is identical.

Figure 3 shows an 8086 I/O mapped EPB1400 peripheral. The EPB1400 $\overline{\text{CWS}}$ (Chip Write Strobe) and $\overline{\text{CRS}}$ (Chip Read Strobe) input pins are connected to the 8288 $\overline{\text{IOWC}}$ (I/O Write Command) and $\overline{\text{IORC}}$ (I/O Read Command) control signal respectively. An EP610 based address decoder generates enable inputs for the EPB1400 RBUSI WE and LBUSO RE control. The data bus, buffered by a 74AS245 transceiver, connects to the EPB1400 bus port transceiver pins.

Figure 5 shows the 8086 read cycle timing diagram. The EPB1400 bus port transceiver drives the data bus when the $\overline{\text{IORC}}$ signal is low and the latched address matches the EP610 decode value for RE. Critical timing paths are EPB1400 driving the MPU data bus in response to either a valid latched address (TAVDV) or the $\overline{\text{IORC}}$ signal going active low (TRLDV), and from $\overline{\text{IORC}}$ going inactive (high) to the onset of the next latched address (TRHDZ), which corresponds to the time allowed for the bus to stop driving.

Figure 6 shows the 8086 write cycle timing diagram. The data bus contents are written to an input latch or register when the $\overline{\text{IOWC}}$ signal is

Figure 5. 8086 Peripheral Timing—Read Cycle

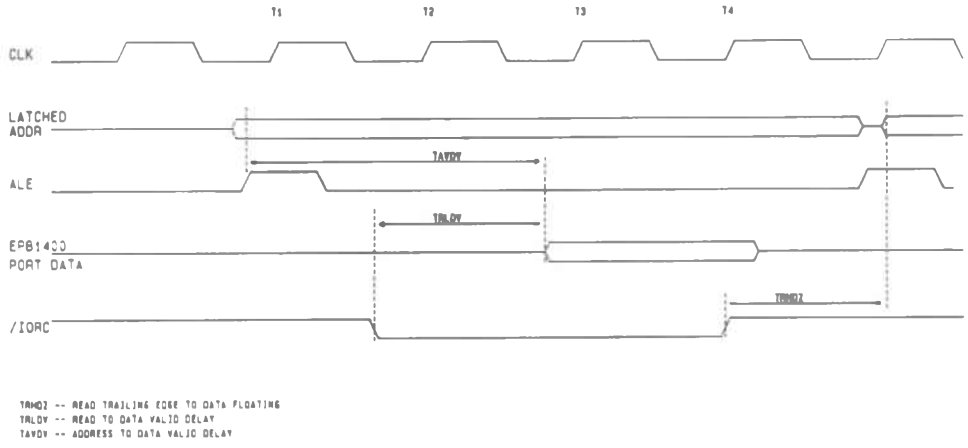
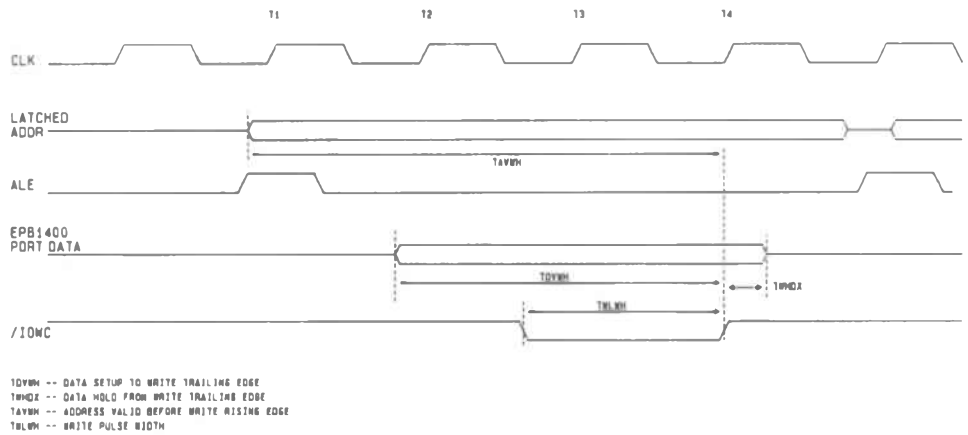


Figure 6. 8086 Peripheral Timing—Write Cycle



active low, and the latched address matches the EP610 decoded value for WE. Critical timing is from a valid latched address to the IOWC rising edge (TAVWH), valid data on the data bus to the rising edge of IOWC (TDVWH), and the required hold time for data after IOWC trailing edge (TWHDX).

Figure 7 shows the results of critical timing analysis for the 10MHz 8086 system shown in Figure 3, which confirms that the EPB1400 satisfies all critical timing. The worst case values for the EPB1400 were generated by adding the appropriate microparameter values, which are included in the EPB1400 data sheet, for each path.

80286

The 80286, a 16 bit MPU with 24 address lines and 16 data lines, has control handshaking similar to the 8086. The 82288 Bus Controller generates the MEMR, MEMW, IORC, and IOWC read and

Figure 7. 8086/88 Interface Timing

Parameter	8086-1	EPB1400-2
TRLDV	179	19
TRHDZ	65	16
TAVDV	243	65
TAVWL	254	51
TWLWH	75	10
TDVWH	154	7
TWHDX	33	16

EPB1400 Timing Calculations for 8086 System

TAVDV

T_{DECODE}	25
T_{IN}	4
T_{CAD}	16
LBUSO $T_{PZX\ RE\text{-}IB}$	8
BUSX $T_{P\ IB\text{-}P}$	12
TAVDV	65

TRLDV

T_{IN}	4
LBUSO $T_{PZX\ RS\text{-}IB}$	3
BUSX $T_{P\ IB\text{-}P}$	12
TRLDV	19

TWLWH

RBUSI $T_{W\ WS}$	10
TWLWH	10

TWHDX

T_{IN}	4
BUSX $T_{PXZ\ \overline{RS}\text{-}P}$	12
TWHDX	16

TRHDZ

T_{IN}	4
BUSX $T_{PXZ\ \overline{RS}}$	12
TRHDZ	16

TAVWL

T_{DECODE}	25
T_{IN}	4
T_{CAD}	16
RBUSI $T_{SU\ WE/\overline{WS}}$	6
TAVWH	51

TDVWH

BUSX $T_{P\ P\text{-}IB}$	4
RBUSI $T_{SU\ \overline{WS}}$	3
TDVWH	7

write control signals based on status information from the 80286 MPU, just as the 8288 did in the 8086 system. Figure 8 shows an 80286 I/O mapped EPB1400 peripheral. The EPB1400 $\overline{CWS}$ and $\overline{CRS}$ are tied to the 82288 $\overline{IOWC}$ and $\overline{IORC}$ control signals respectively. An EP610 based address decoder generates enable inputs for the EPB1400 RBUSI WE and LBUSO RE. The data bus, buffered by a 74AS245 transceiver, connects to the EPB1400 bus port transceiver pins.

Figure 9 shows the 80286 read cycle timing diagram. The EPB1400 bus port transceiver drives the data bus when the $\overline{IORC}$ signal is low and the latched address matches the EP610 decode value for RE. Critical timing is identical to the 8086 case: TAVDV, TRLDV, and TRHDZ. Figure 10 shows the 80286 write cycle timing diagram. The data bus contents are written to an input latch or register when the $\overline{IOWC}$ signal is active low, and the latched address matches the EP610 decoded value for WE. Critical timing is from a valid latched address to the $\overline{IOWC}$ active low edge (TAVWL), valid data on the data bus to the rising edge of $\overline{IOWC}$ (TDVWH).

Figure 8. 80286 EPB1400 Interface

A typical 80286 system, with buffered data and latched address buses.

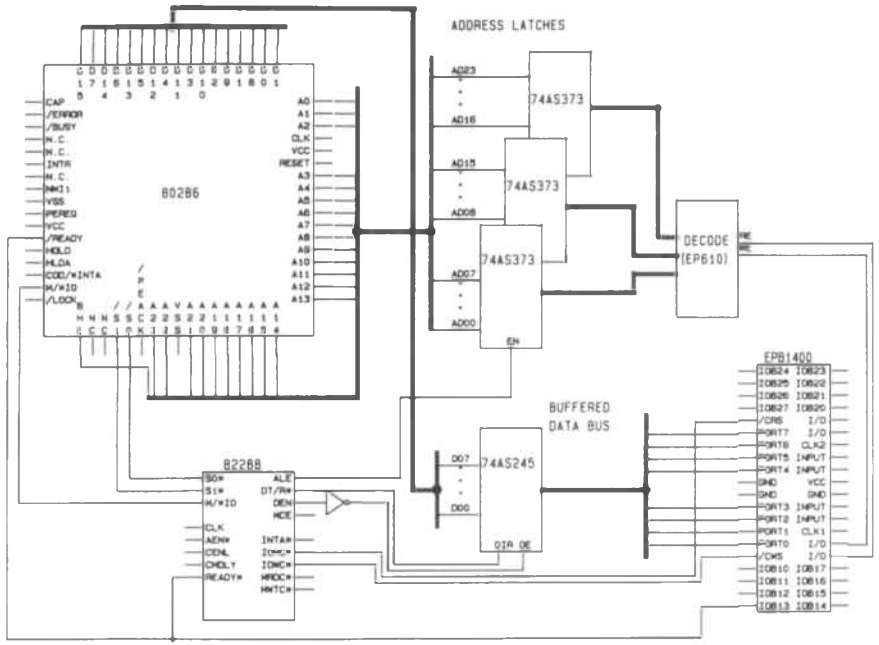
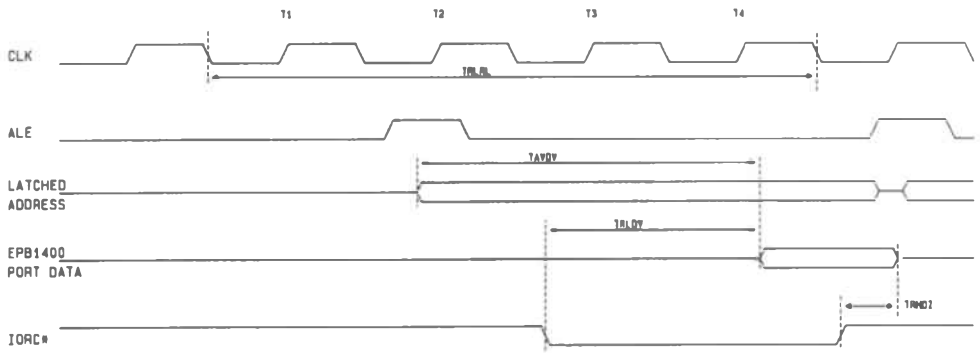


Figure 9. 80286 Peripheral Timing—Read Cycle



TAVDV -- ADDRESS TO DATA VALID DELAY
TRLDV -- READ TO DATA VALID DELAY
TRHDZ -- READ TRAILING EDGE TO DATA FLOATING
TRLRL -- READ CYCLE TIME

Figure 10. 80286 Peripheral Timing—Write Cycle

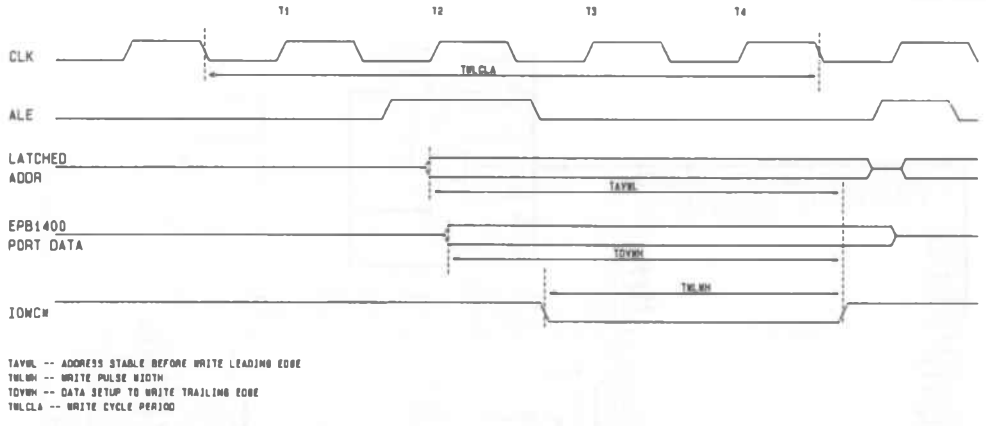


Figure 11 shows the results of critical timing analysis for the 12.5 MHz 80286 system shown in Figure 8, which confirms that the EPB1400 satisfies all critical timing.

Figure 11. 80286 Interface Timing

Parameter	80286-12	EPB1400-2
TRLDV	54	19
TRHDZ	—	16
TAVDV	76	65
TAVWL	101	51
TWLWH	64	10
TDVWH	87	7

68020

Figure 12 shows the 68020, a 32 bit MPU with separate 32 bit address and data buses, connected to an EPB1400 peripheral. Addresses are latched on the falling edge of AS by four 74AS373 flow-through latches. When the latched address matches the memory mapped EPB1400 address, the address decoder generates a CS (Chip Select) signal, which is an input to the EPB1400. The data bus is buffered by four 74AS245 transceivers controlled by the R/W (Read/Write) direction signal. Buffered data bits D24 to D31 are connected to the EPB1400 BUSX pins (PORT0-7). Data is valid when DS is low.

The address decode logic must generate DSACK0 and DSACK1 (Data Transfer and Size Acknowledge), which informs the MPU how wide and fast the next data transfer will be. For this example, whenever the MPU attempts a read or write operation over the EPB1400 BUSX pins, DSACK0 should be set low and DSACK1 should be set high, indicating an 8 bit data transfer without wait-states will follow.

Figure 13 shows the EPB1400 interface function selections for this 68020 application. The LBUSO RE is enabled when DS is low, and both R/W and CS are high. The RS is not used. The LBUSI input latch WE is generated when R/W and CS are high. The 68020 DS signal is connected to the LBUSI WS. Figure 14 shows the 68020 read cycle timing. Critical read cycle timing is from a valid latched address to MPU internal data latched on the S4 cycle falling edge clock (TAVDV), the falling edge of DS to the S4 clock cycle falling edge (TRLDV), and the time to disable data after DS (TRHDZ).

Figure 15 shows the 68020 write cycle. Critical timing is from the latched address to the falling edge of DS (TAVWL), valid data to the rising edge of DS (TDVWH), and the hold time of data after the rising edge of DS (TWHDX), and the width of DS (TWLWH).

Figure 12. 68020 EPB1400 Interface

In a typical 32 bit 68020 system, the EPB1400 connects to the high order bits (D24-31) of the buffered data bus.

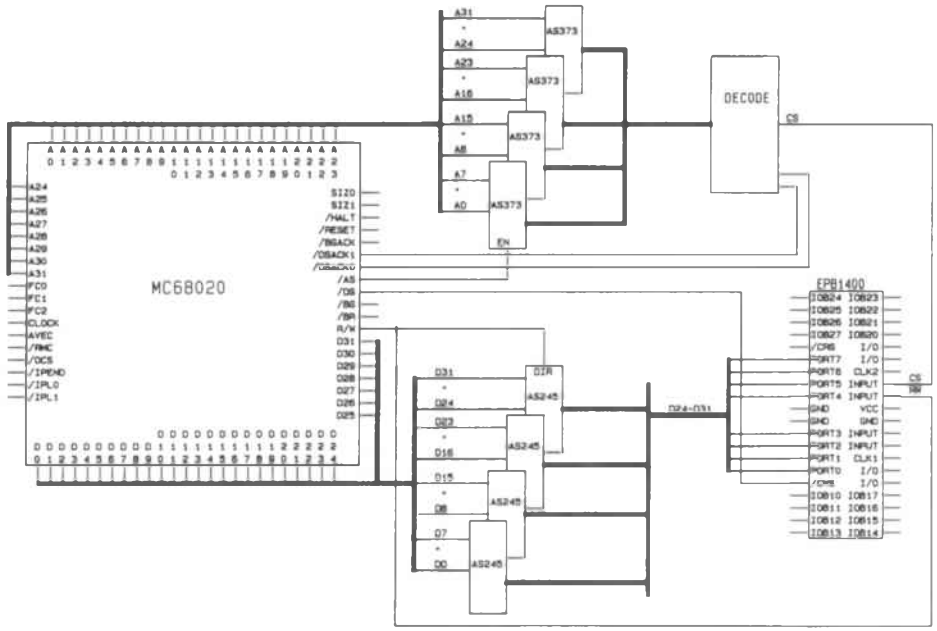


Figure 13. Simplified 68020 MPU Interface

The 68020 does not have a convenient $\overline{RS}$ (Read Strobe). Instead the LBUSO RE (Read Enable) controls MPU data read operations.

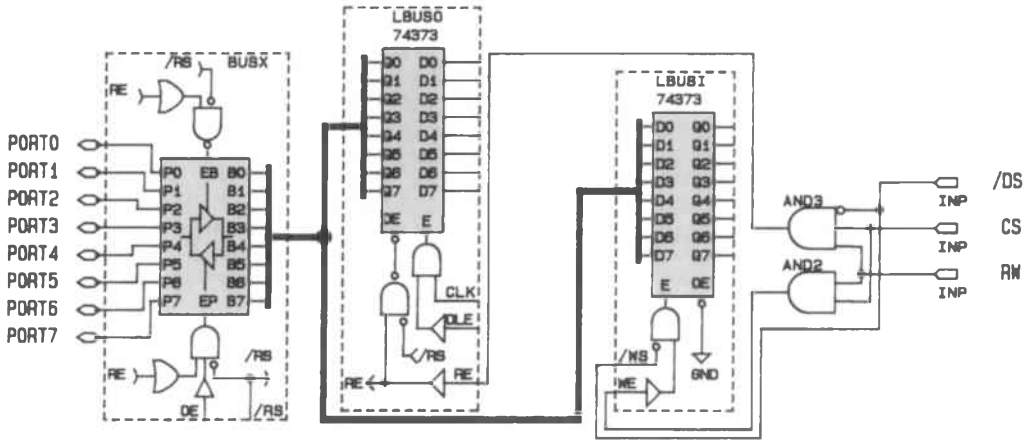
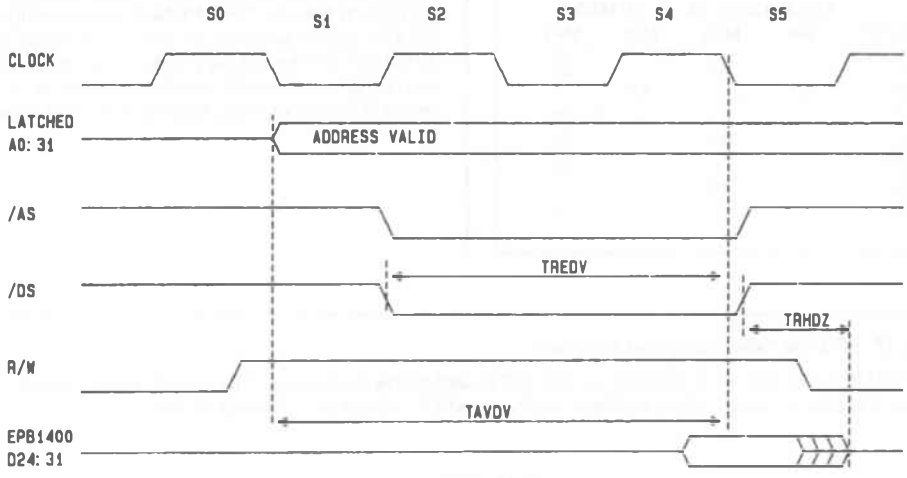
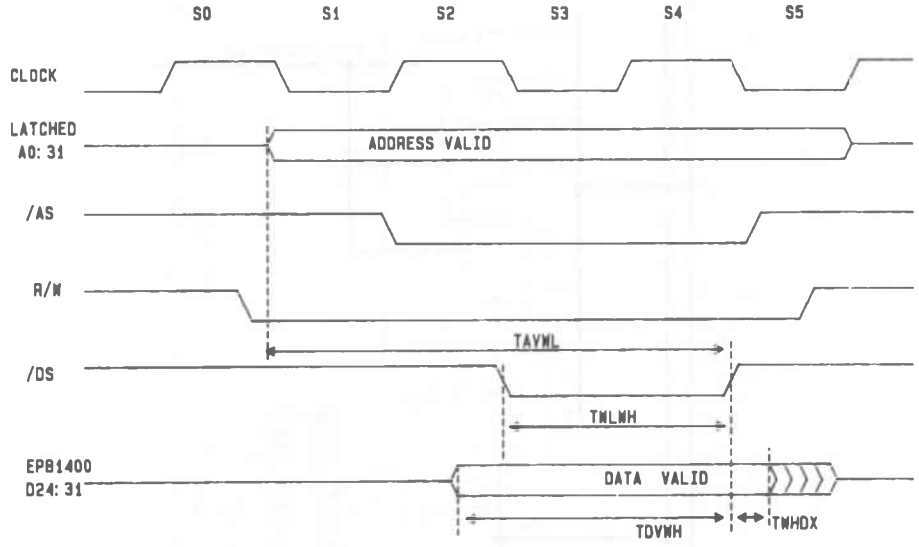


Figure 14. 68020 Read Cycle Timing



TAVDV -- LATCHED ADDRESS SETUP ON S4 FALLING CLOCK
TREDV -- /DS FALLING EDGE SETUP ON S4 FALLING CLOCK
TRHDZ -- /DS RISING EDGE TO DATA TRISTATED DELAY

Figure 15. 68020 Write Cycle Timing



TWLWH -- /DS WIDTH
TDVWH -- DATA SETUP ON /DS RISING EDGE
TWHDX -- DATA HOLD TIME AFTER /DS RISING EDGE
TAVWL -- LATCHED ADDRESS SETUP ON /DS RISING EDGE

4

Figure 16. 68020 Interface Timing

Parameter	MC68020RC20 Min	MC68020RC20 Max	EP1400-2 Min	EP1400-2 Max
TAVDV	—	83	—	59
TREDV	64	—	34	—
TRHDZ	—	50	—	32
TAVWL	—	94	—	51
TWLWH	—	25	—	10
TDVWH	—	48	—	7
TWHDX	—	10	—	7

Figure 16 shows the results of critical timing analysis for the 20MHz 68020 system shown in Figure 12, which confirms that the EPB1400 satisfies all critical timing. The EPB1400 is compatible with 25MHz 68020 systems, if read operations use the EPB1400 $\overline{RS}$ exclusively, otherwise the delay incurred generating RE, which includes the external address decoder, may exceed the TRLDV spec.

Figure 17. EPB1400 MicroChannel Interface

The EPB1400 can connect directly to the PS2 backplane because of its 24mA drive capability. Address decode is done before address latching, which simplifies interface timing.

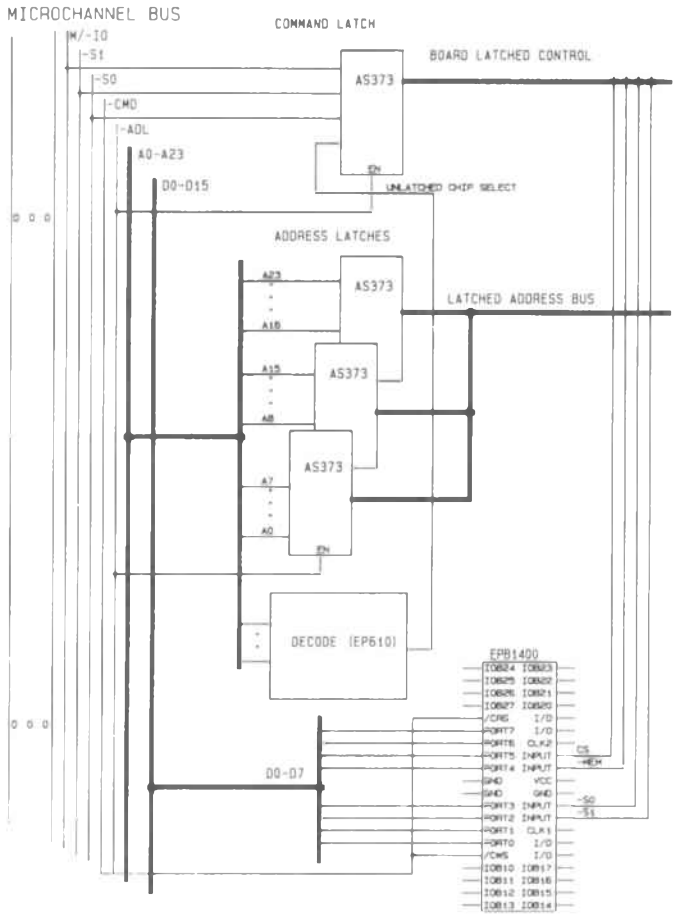
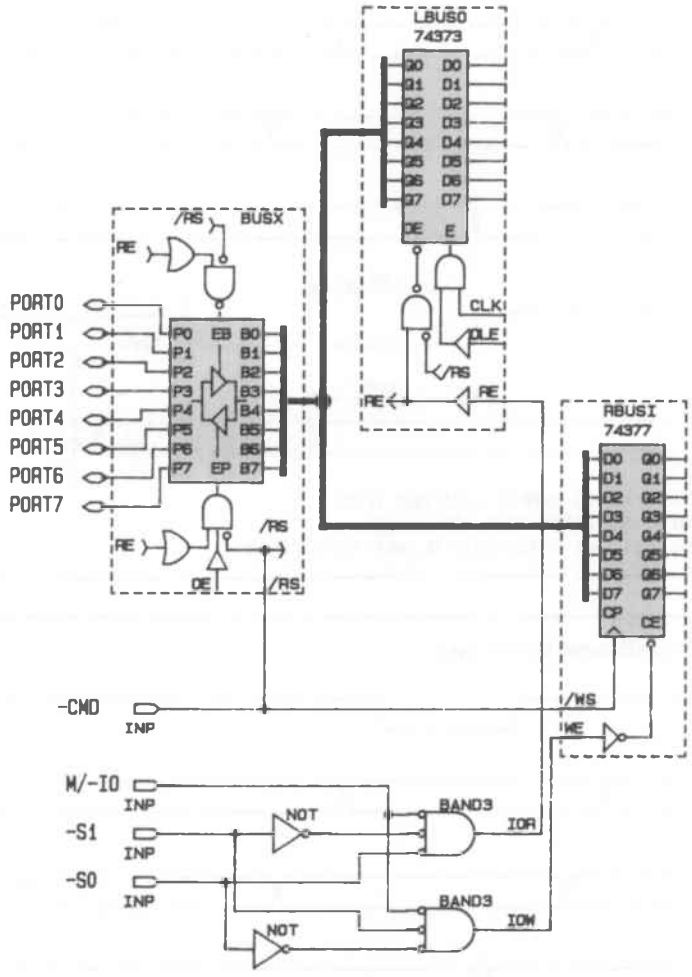


Figure 18. Simplified MicroChannel Interface

Status bits (S0 and S1) are internally decoded to generate IOR and IOW signals in this simplified interface. The CMD (Command) signal serves as both RS and WS.

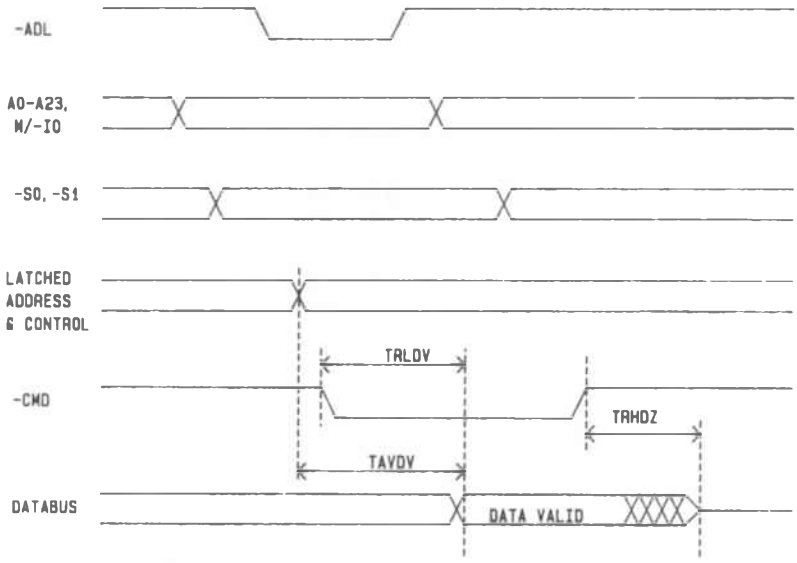


PS/2 MicroChannel

The PS/2 Micro Channel bus, a new backplane standard which links up to seven expansion cards in IBM Personal System 2 personal computers, has a 24mA current drive requirement, which can be driven by the EPB1400 BUSX pins (PORT0-7) directly. Figure 17 shows the MicroChannel-EPB1400 connection used on a typical PS/2 interface card. The EPB1400 BUSX pins connect directly to the data bus low byte (D0-D7).

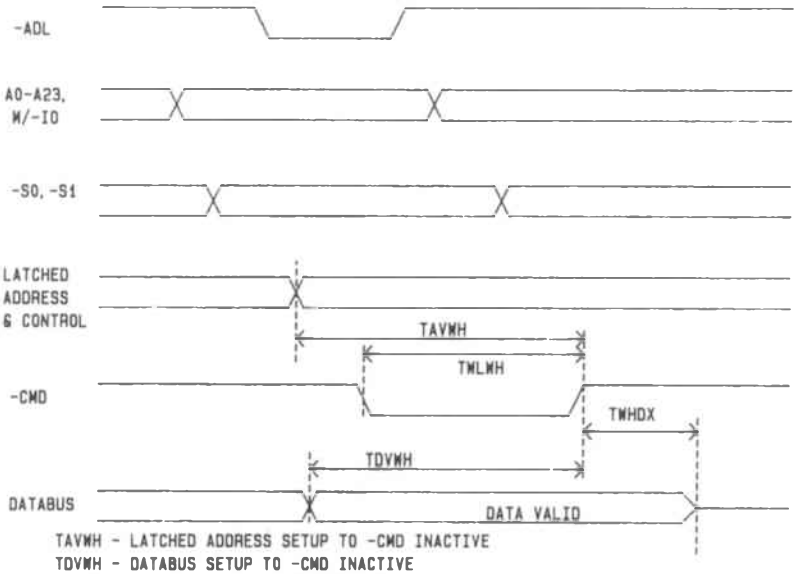
The address bus and command information are latched in four 74AS373 flow-through latches on the falling edge of ADL (Address Decode Latch). Latched control signals include M/IO (Memory/Input Output), S1 and S0 (Status Bits 1 and 0), and any address decodes relevant to the PS/2 interface card under development. A sample EPB1400 to PS/2 interface is shown in Figure 18. S1 and S2 are decoded in the EPB1400 to indicate the current cycle type: IOR (I/O Read) and IOW (I/O Write). The active low CMD (Command) signal defines when data is valid on the data bus, and is used as both RS and WS on the EPB1400.

Figure 19. PS/2 Micro Channel Read Cycle



TAVDV - LATCHED ADDRESS TO DATABUS VALID
TRLDV - -CMD ACTIVE TO DATABUS VALID
TRHDZ - DATABUS (READ) TRISTATE FROM -CMD INACTIVE

Figure 20. PS/2 Micro Channel Write Cycle



TAVWH - LATCHED ADDRESS SETUP TO -CMD INACTIVE
TDVWH - DATABUS SETUP TO -CMD INACTIVE
TWHDX - DATABUS HOLD TIME AFTER -CMD INACTIVE
TWLWH -CMD WIDTH

Figure 19 shows the PS/2 read cycle timing. Critical timing delays are from valid latched address to data valid (TAVDV), CMD falling edge to data valid (TRLDV), and CMD rising edge to data bus tristate (TRHDZ). Figure 20 shows the PS/2 write cycle timing. Critical timing delays are from valid address to CMD rising edge (TAVWH), data setup on CMD rising edge (TDVWH), hold time for data after CMD rising edge (TWHDX), and the CMD signal width (TWLWH). Figure 21 confirms that the EPB1400 is A.C. timing compatible with the PS/2 Micro Channel.

Figure 21. PS/2 MicroChannel Interface Timing

Parameter	MicroChannel Min	MicroChannel Max	EPB1400-2 Min	EPB1400-2 Max
TAVDV	—	94	—	39
TRLDV	—	54	—	20
TRHDZ	—	40	—	16
TAVWH	—	124	—	26
TDVWH	—	90	—	7
TWHDX	—	30	—	4
TWLWH	—	90	—	4

SPECIFYING PERIPHERAL FUNCTIONS

After designing the MPU bus interface portion, the peripheral functions are implemented in the EPB1400 programmable logic core. The programmable logic core is composed of 20 general purpose macrocells which can implement both sequential and combinatorial custom logic. Macrocell logic is generally specified using schematic capture and TTL macrofunctions. (See the following Application Note for a specific design example).

PROGRAMMABLE LOGIC CORE INSIGHTS

The following section highlights features of the 20 general purpose EPB1400 macrocells, which may be used to implement peripheral functions. This section should be read in conjunction with the EPB1400 data sheet.

THE BASIC MACROCELL

Figure 6 of the EPB1400 data sheet shows the architecture of the general purpose macrocell; each has 8 product terms (fed by the logic array), and an optional inversion, which permits both active high or low logic functions. Inputs to the logic array vary with the placement of the macrocell (see EPB1400 datasheet Figure 3); side 1 macrocells (MAC1-MAC10) are fed by the side 1 input register, side 2 macrocells (MAC11-MAC20) are fed by the side 2 input register. All macrocells are fed by the true and compliment of all dedicated inputs and general purpose macrocell feedbacks.

Other programmable macrocell features include flip-flop type selection, clock selection, clear selection and tri-state control.

PROGRAMMABLE FLIP-FLOP TYPE

Each macrocell contains a programmable flip-flop which may be independently configured as D, T, JK, or SR type without any additional logic requirements.

FLIP-FLOP CLOCKING MODES

Each of the 20 programmable flip-flops may select from two clock sources: either a direct synchronous clock pin input or a programmable clock structure generated within the logic array. The EPB1400 has two synchronous clock input pins (CLK1 and CLK2); CLK1 drives up to ten macrocells on the left side of the chip, CLK2 drives up to ten macrocells on the right side. To clock more than 10 macrocells synchronously, CLK1 and CLK2 should be externally tied to the same external clock signal.

The programmable clock structure contains 2 product terms followed by an optional invert function, and is replicated in every macrocell; allowing each flip-flop to be individually clocked by a unique signal. The invert bit not only transforms a leading edge-trigger into a falling edge-trigger, but permits De Morgan's inversion to create more complex clocking structures.

DUAL FEEDBACK I/O

Dual feedback I/O is a feature which conserves device pins. Dual feedback exists when there is one feedback path from the output macrocell, and a second feedback path from the I/O pin. A buried macrocell pin is saved by disabling the macrocell tristate that normally drives the pin, allowing the I/O pin to be used as a dedicated input back into the programmable logic array.

CONFIGURING MACROCELLS

Macrocells are configured using Altera's development tools. Altera's LogiCaps schematic capture provides access to a primitive symbol library, and a macrofunction library of over 120 TTL SSI/MSI symbols and behavioral models. Each TTL macrofunction symbol indicates worst case macrocell consumption in its lower right corner. These symbols are shown in the Altera Data Book TTL-Macrofunction description (PLSLIB-TTL). A first order estimate of a successful fit is accomplished

by subtracting macrocells consumed from those available. Refer to Applications Handbook Section, "Estimating a Design Fit" to calculate how much logic will fit into a single EPB1400. Designs may be alternately specified using high-level state machine syntax or boolean expressions.

CONCLUSION

The EPB1400 is a function specific solution tailored to the microprocessor interface problem. The EPB1400 function specific features include dedicated input and output registers, a port transceiver, and 7 control macrocells. These features absorb the interface burden without consuming general purpose macrocells. The EPB1400 contains 20 general purpose macrocells, available to implement the peripheral task. The result is a highly-integrated, user-configurable solution which can be easily and quickly customized to a specific peripheral design task. Figure 25 summarizes the features of th EPB1400.

AN12 Rev 2.0

Figure 25. EPB1400 Summary of Features

PACKAGE:	EITHER 40 PIN DIP OR 44 PIN JLCC
CURRENT CONSUMPTION:	70 mA
MICROPROCESSOR INTERFACE BLOCK	
ELEMENTS:	1 of 8 BIT PORT TRANSCEIVER 2 of 8 BIT INPUT REGISTER 2 of 8 BIT OUTPUT LATCH
CONTROL MACROCELLS:	7
BUS PORT DRIVE:	24 mA
CONTROL FEATURES:	2 PTERMS WITH OPTIONAL INVERSION DEDICATED FAST READ AND WRITE STROBES
DESIGN SUPPORT:	BUSTER PRIMITIVES IN LOGICAPS SCHEMATIC CAPTURE
PROGRAMMABLE LOGIC CORE	
ELEMENTS:	20 GENERAL PURPOSE MACROCELLS
MACROCELL FEATURES:	SELECT D, T, JK, or SR FLIP-FLOPS 8 PTERMS WITH OPTIONAL INVERT AT FLIP-FLOP INPUT 1 PTERM CLEAR PER MACROCELL 2 PTERMS WITH OPTIONAL INVERT FOR EITHER PROGRAMMABLE CLOCK OR OUTPUT ENABLE CONTROL DUAL FEEDBACK INTERFACE TO INPUT AND OUTPUT REGISTERS
DESIGN SUPPORT:	TTL MACROFUNCTIONS, STATE MACHINE ENTRY, BOOLEAN ENTRY.

FEATURES

- **Macrocell-based serializer demonstrates the EPB1400's architectural flexibility.**
- **Microprocessor interface simplified by EPB1400's dedicated functions.**
- **30 MHz operation.**

INTRODUCTION

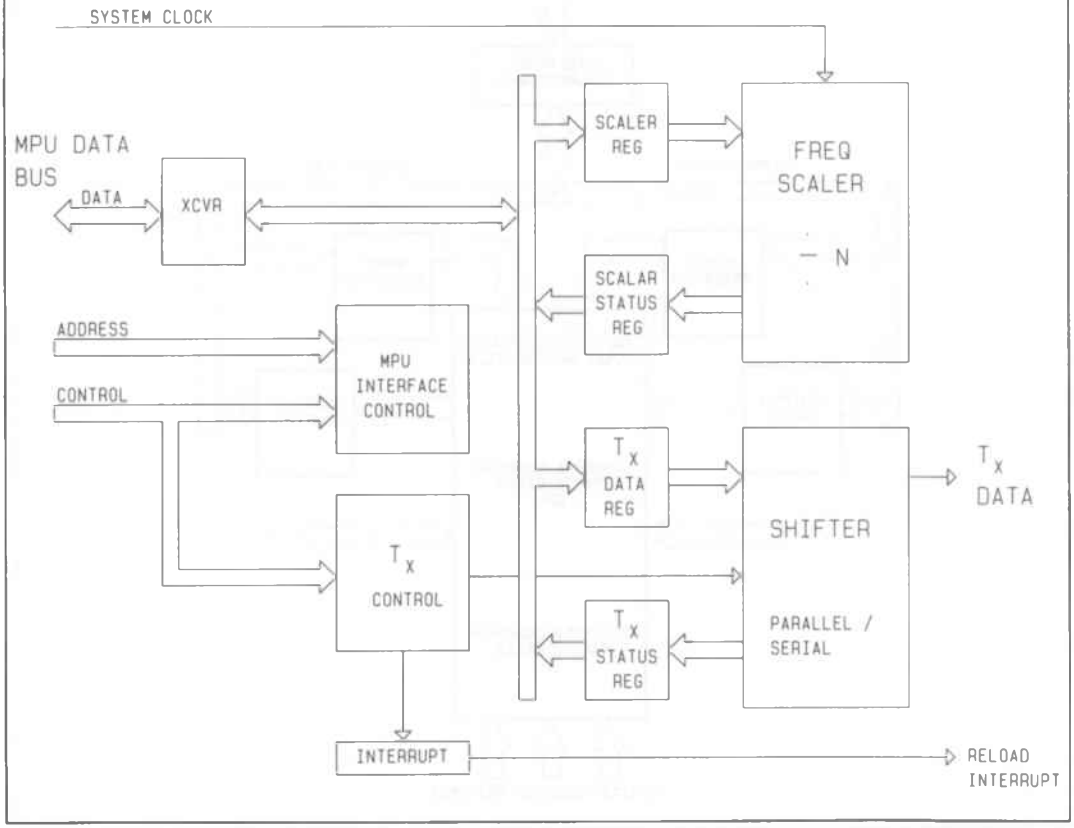
This Application Brief demonstrates the integration and architectural flexibility of the EPB1400 (BUSTER). An application using the EPB1400 as a data communications transmitter is used to illustrate the capability of this EPLD, and logic techniques employed when designing with the EPB1400.

The reader is referred to the Altera EPB1400 Data Sheet for details concerning device architecture and performance. A general knowledge of the EPB1400 is assumed.

SERIAL DATA COMMUNICATIONS TRANSMITTER REQUIREMENTS

Despite a wide acceptance for protocols (e.g. Ethernet) that encourage the use of standard peripheral solutions, the need for proprietary and customized data communications networks persists. Figure 1 shows the major functional elements of a serial transmitter peripheral. A microprocessor (MPU) submits a data byte to be serialized over its data bus through a bus transceiver (XCVR) to be stored in a dedicated register (Tx Data Register). The stored data can then be loaded into a Shifter,

Figure 1 Serial Transmitter Block Diagram



which serializes the data. The loading and serialization of data is coordinated by a state machine (Tx Control). Different communications protocols may require different transmission rates; it is best to make variable transmission rates a programmable feature. The Frequency Scaler controls transmission rate by deriving the transmission clock as a multiple of the system clock. The rate is programmed by the MPU writing the multiple value into a storage register (Scaler Register).

Diagnostic information is important during system power-on or reset tests. The contents of the Frequency Scaler and Shifter can be latched in dedicated registers (Scalar Status Register and Tx Status Register), and read by the MPU across the bus transceiver.

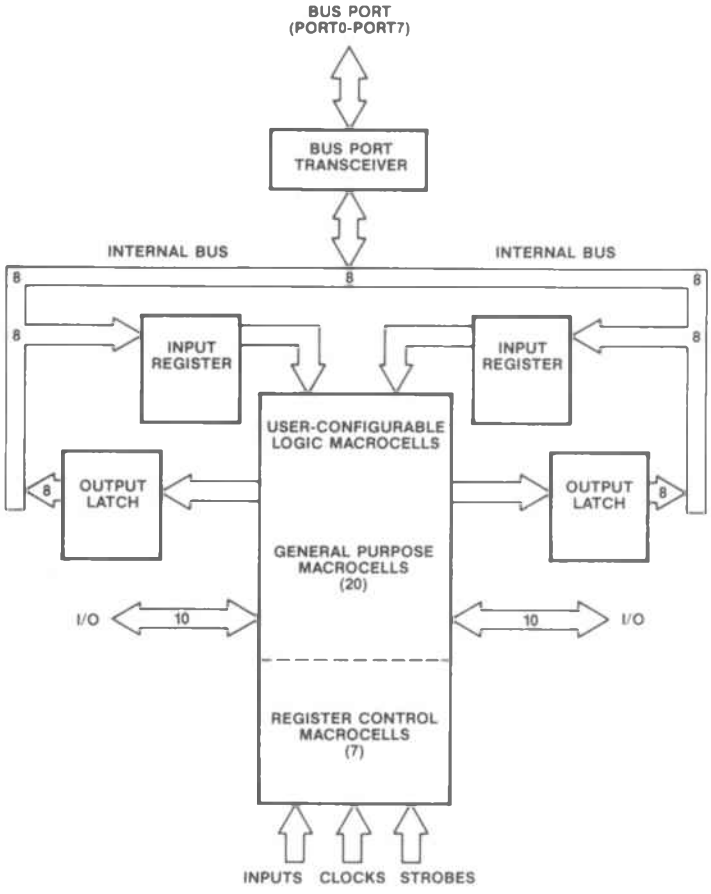
All access to configuration (Scaler and Tx Data) and diagnostic (Scalar Status and Tx Status) registers is coordinated by the MPU interface control block. For this example, assume the system clock is 25MHz.

EPB1400 SUITABILITY

Refer to Figure 2 for a block diagram of the EPB1400. This figure shows the EPB1400 resources that will be used by the Serial Data Communications Transmitter application.

The serializer design requires a byte-wide transceiver (XCVR), two input registers (Scalar Register and Tx Data Register), and two output latches (Scalar Status Register and Tx Status Register). The EPB1400 has these functions built into its programmable MPU interface, thus yielding very compact designs. Implementation within the EPB1400 allows for operation of a 25 MHz bus cycle without wait states. The built-in bus transceiver's 24mA I_{OL} current drive capability allows connection to MPU data buses. Maximum counter frequency is 30MHz (see f_{CNT} in A.C. specifications in the EPB1400 data sheet), allowing 30 Mbit serial data rates to be obtained through the transmitter,

Figure 2 EPB1400 Basic Block Diagram



with minimal intervention by the processor.

The Scaler, Shifter, Transmit Control and glue logic can be implemented in the EPB1400's general-purpose macrocells. The microprocessor interface control logic will be implemented in the Register Control macrocells. This application fits into a single EPB1400 device.

DETAILS OF THE EPB1400-BASED TRANSMITTER

Figure 3 is a Logicaps schematic of a serial transmitter implemented in a single EPB1400. The following device pins are required for this application:

$\overline{CE}$ (Chip Enable) is an active low input that selects the EPB1400 for programmed transfers with the host MPU.

A_0 and A_1 (Address Lines) are used in conjunction with the $\overline{CE}$ input to access the EPB1400 internal registers.

D_0-D_7 (Data Lines) are bi-directional data lines which transfer data between the EPB1400 and the MPU data bus.

$\overline{RD}$ (Read Control Signal) is an active low read control input. $\overline{RD}$ must be asserted before data can flow from the EPB1400 to the MPU data bus.

$\overline{WR}$ (Write Control Signal) is an active low write control strobe for the EPB1400 dedicated input registers. The current EPB1400 internal bus data is stored in an enabled register on the rising edge of $\overline{WS}$.

$\overline{RESET}$ (Reset Control Signal) clears the Shifter, Scaler, and RELDINT when low. $\overline{RESET}$ also re-initializes the Transmit Control.

RELDINT (Reload Interrupt), an active high output, signals an empty data buffer to the microprocessor.

SYSCLK (System Clock) is the scaler clock frequency. SYSCLK is divided by the factor loaded into the scaler register to form the Shifter clock rate.

TxDATA (Transmit Data) is the serial output of the data being transmitted through the parallel/ serial shift register.

Figure 3 shows the key functional blocks that compose the design of the serial transmitter, microprocessor interface, scaler, transmitter, and diagnostic status registers.

Figure 3 Serial Transmitter Detailed Design

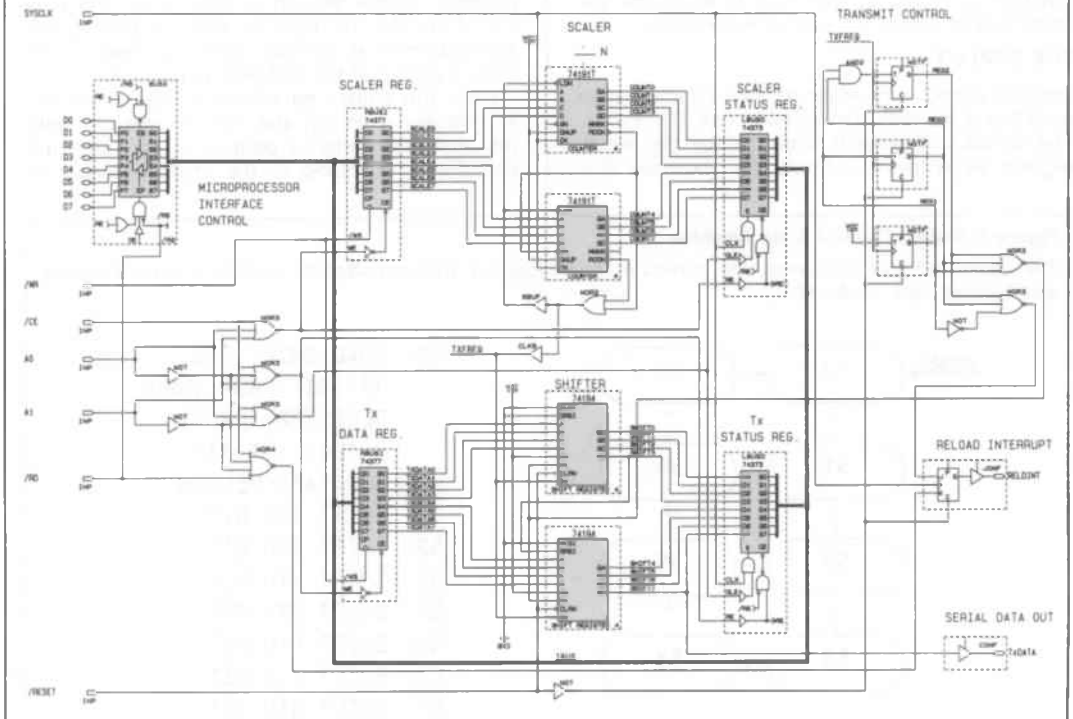


TABLE 1. TRANSMITTER COMMAND DECODE

$\overline{WR}$	$\overline{RD}$	$\overline{CE}$	A_0	A_1	SYSCLK	OPERATION
X	X	1	X	X	X	No I/O Operation
↑	1	0	1	0	X	Write Next Tx Byte
↑	1	0	0	0	X	Write New Scale Value
X	X	0	0	1	↑	Status Registers Enabld
X	0	0	0	0	X	Read Current Scaler Value
X	0	0	1	0	X	Read Current Tx Data Byte

THE MICROPROCESSOR INTERFACE

The microprocessor interface takes inputs from the microprocessor, including $\overline{WR}$, $\overline{RD}$, A_0, A_1 and the microprocessor data bus. The $\overline{CE}$ input is created by an external decoder which uses latched microprocessor address values to map the EPB1400 to a specific address range. Microprocessor data may be used as a scalar value or as transmission data. The EPB1400 Serial transmitter peripheral's I/O registers are mapped into I/O ports in the microprocessor's address space. The low order addresses, A_0 and A_1, are used to access resources inside the EPB1400. The internal decode information is required to write the software drivers necessary to access the customized peripheral. A table of the functions decoded from the address and chip select information is shown in Table 1. The decode circuitry is included just below the BUSX function in Figure 3. The decode functions are implemented within the control macrocells.

THE SCALER

The byte stored in the scaler register is a scale value used to set the data transmission rate (TXFREQ). The SYSCLK frequency is divided by the scaler register value to fix TXFREQ. For example, if a value of 4 is stored in the scaler register, then the transmit frequency would be ¼ the frequence of SYSCLK. The scaling rate is set by loading the scaler register into an 8-bit counter's data inputs: the two cascaded 74191 4-bit counter's A, B, C, and D inputs. A zero count value generates a load enable, which presets the counter with the scaler value. The counter counts down, eventually reaching a zero condition, which generates a TXFREQ pulse to clock the shifter, and reloads scaling value to reinitialize the counter.

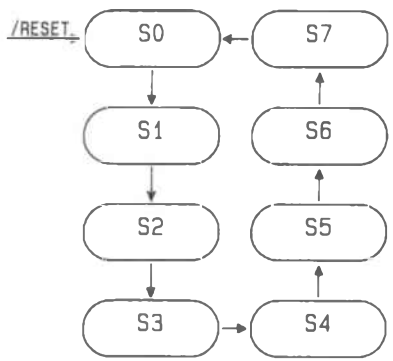
THE SHIFTER

Microprocessor data to be transmitted is loaded into the Tx Data register when the appropriate address is detected. The Tx Data register feeds the shifter: implemented by a pair of cascaded 74194 4-bit parallel loadable, serial shift registers. The transmit control section enables a parallel data load from the Tx Register, and the subsequent serialization of shifter data. Data is shifted out the TxData output at the TXFREQ clock rate.

After the shifter parallel load operation, the Tx Register is empty, and may be reloaded with the next transmission data. A reload interrupt (RELDINT) is issued to the microprocessor to

Figure 4 Transit Control State Machine

The transit control state machine coordinates the loading and serialization of data. It also activates an interrupt back to the MPU.



S0: LOAD SHIFT REG. (74191) FROM INPUT REGISTER (RBUSI) OUTPUT 1ST BIT.
S1: ACTIVATE RELDINT SHIFT 2ND BIT
S2: SHIFT 3RD BIT
S3: SHIFT 4TH BIT
S4: SHIFT 5TH BIT
S5: SHIFT 6TH BIT
S6: SHIFT 7TH BIT
S7: SHIFT 8TH BIT

signal that the Tx Data register is empty. The microprocessor clears the RELDINT interrupt by reading the EPB1400 with the $\overline{CE}$ signal low, A_0 and A_1 signals high.

The shifter is controlled by the transmit control state machine, generally described in a high-level state machine syntax, and then linked into the schematic. In this example, the state machine is represented by TTL logic, found in the upper-right hand corner of Figure 3. The state machine consists of three flip-flops and associated logic. Figure 4 shows the flow diagram, which describes the transmit control sequencing behavior.

DIAGNOSTIC STATUS REGISTERS

The microprocessor may issue a command to read the Scaler or Shifter contents at any time. With the appropriate address asserted (see Table 1), the microprocessor may latch the contents of the scaler into the Scaler Status Register, and the Shifter into the Tx Status Register. Subsequent read operations to designated addresses will pass these data words onto the microprocessor bus. This read capability is important in diagnostics because it allows the processor to monitor transmit data rates and if necessary, adjust them.

CONCLUSION

User-configurability allows the EPB1400 user to customize all aspects of the Transmitter to meet system needs. For example, inverting the polarity of TxDATA, adding start bits to each character, and modifying control sequences are all design changes that may be implemented within the EPB1400 without adding external components or decreasing performance. Dedicated microprocessor interface functions, including the bus port transceiver, input registers, and output latches, allow the EPB1400 to efficiently implement data bus-oriented applications. Easy-to-use design tools allow the design to be drawn with standard logic functions, compiled, simulated and programmed in a quick and efficient manner. The resultant customized serial transmitter design fits in a single EPB1400.

AB57 Rev 1.0

NOTES

STATE MACHINE AND MICROSEQUENCER APPLICATIONS **PAGE NO.**

5

FEATURES

- **Implementing state machines using:**
  - **SSI/MSI**
  - **PLDs**
  - **PROM based sequencers**
  - **Bit slice machines**
  - **Single chip microcoded sequencers**
- **Advanced state machine concepts.**

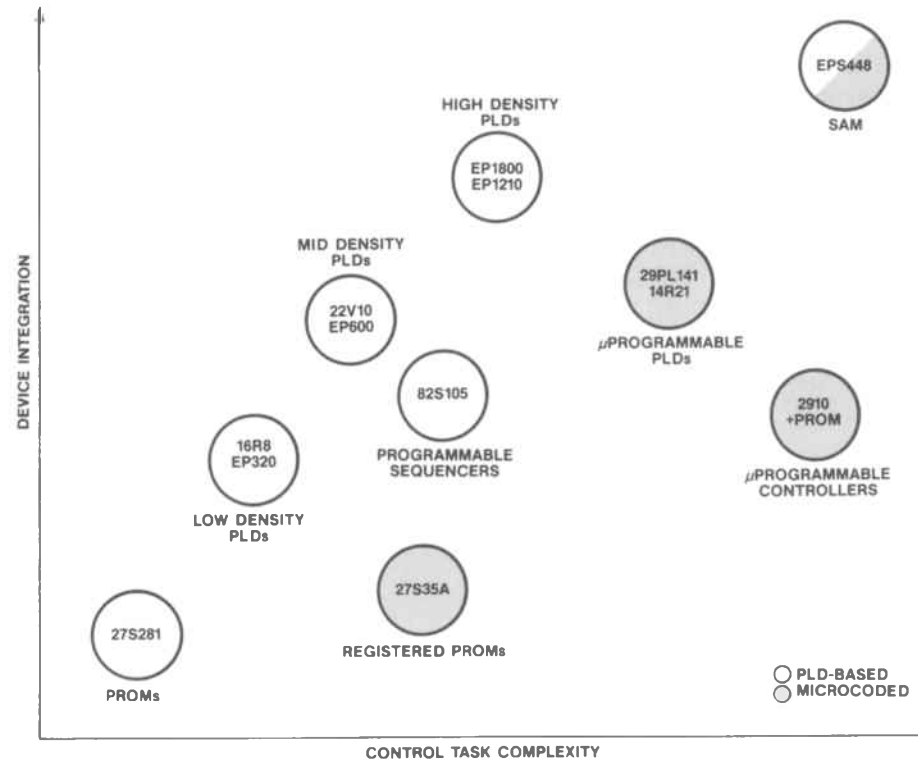
INTRODUCTION

State machines form the basis for all control logic design. As simple as a toggle flip-flop, or as complex as a supercomputer, control functions are all representable as state machines. State machine design skills are important to most digital logic designers.

The alternatives for implementing state machines have evolved over the past two decades from SSI/MSI TTL to include PLDs, PROMs, bit-slice sequencers, PLAs and most recently, single-chip microcoded sequencers, such as Altera's EPS448. This evolution is shown in Figure 1. Individual chip integration density is plotted versus the complexity of state machine task each device can handle.

Executing state machine designs effectively is accomplished by using the correct integrated circuit, as well as using the most efficient design tools. Altera's family of EPLDs and high-level design software fulfills both of these needs. Design entry options for state machines available from Altera are discussed elsewhere in this section.

Figure 1. High Performance Control Alternatives

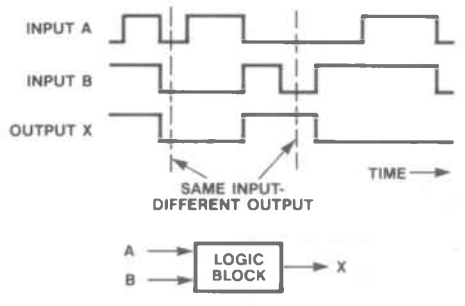


GENERAL STATE MACHINE DEFINITIONS

A **combinatorial** logic circuit is a digital logic system whose output at any given point in time is a boolean logic function of the current inputs to the system. For any given input combination, a defined output pattern is obtained.

State machines represent a class of **sequential** circuits or systems. A sequential system is a digital logic block whose outputs are a function of the current inputs and **previous** inputs as shown in Figure 2. Notice that the same logic input applied at two different times can cause two different output values. In other words, the logic has **memory** which records previous input history so it can be responded to in the present.

Figure 2. Sequential Logic



Given this definition, sequential circuits would seem to require enormous amounts of memory to record all previous inputs. However, for any real logic design task, the fact that previous input combinations result in only a finite number of distinct output classes reduces this memory requirement to manageable levels. This class of design is called a **finite state machine**, or just state machine.

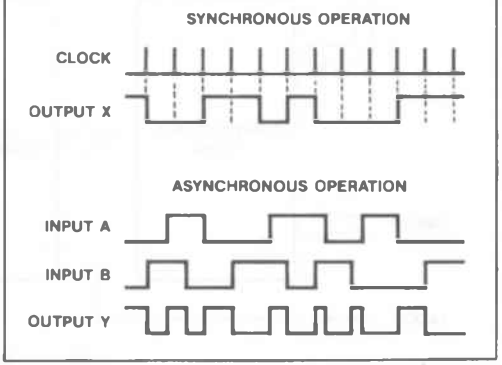
A **state** summarizes, for a given logic function, the influence of past inputs on present and future responses. A serial parity detector clearly illustrates this point. Such a circuit simply determines whether the total number of ones in a serial data stream is odd or even. The detector could do this by having sufficient memory to remember all previous inputs at each point in time and thereby compute the parity. However, it is considerably more efficient to use a single toggle flip-flop which has the serial data wired to its T input and is clocked in-step with the serial data. In the latter case, the toggle flip-flop's Q output value represents the state of the parity detector. As a state memory element, or state variable, it remembers whether the parity was last odd or even. When a new logic 1 is detected at the T input, the parity history contained in the flip-flop toggles for future references. The state changes, or a state **transition** is said to occur.

State variables are those discrete memory elements which in combination are used to represent states. A given state machine may have one or more state variables. The term **present state** is used to describe the state of a sequential circuit at an arbitrary point in time. The **next state** represents the successor state into which the machine will transition on the next input change. Transitions between states must be deterministic: for a given present state and inputs, the next state must always be predictable.

SYNCHRONOUS/ASYNCHRONOUS MACHINES

A **synchronous** state machine is a sequential circuit which has a single synchronizing input signal, frequently called the **Clock**. All state transitions occur in response to a transition on the Clock. Figure 3 illustrates this synchronous characteristic. **Asynchronous** state machines, on the other hand, do not have a predefined Clock. Instead, in the most general case, any input transition may trigger a state transition.

Figure 3. Synchronous/Asynchronous Machines

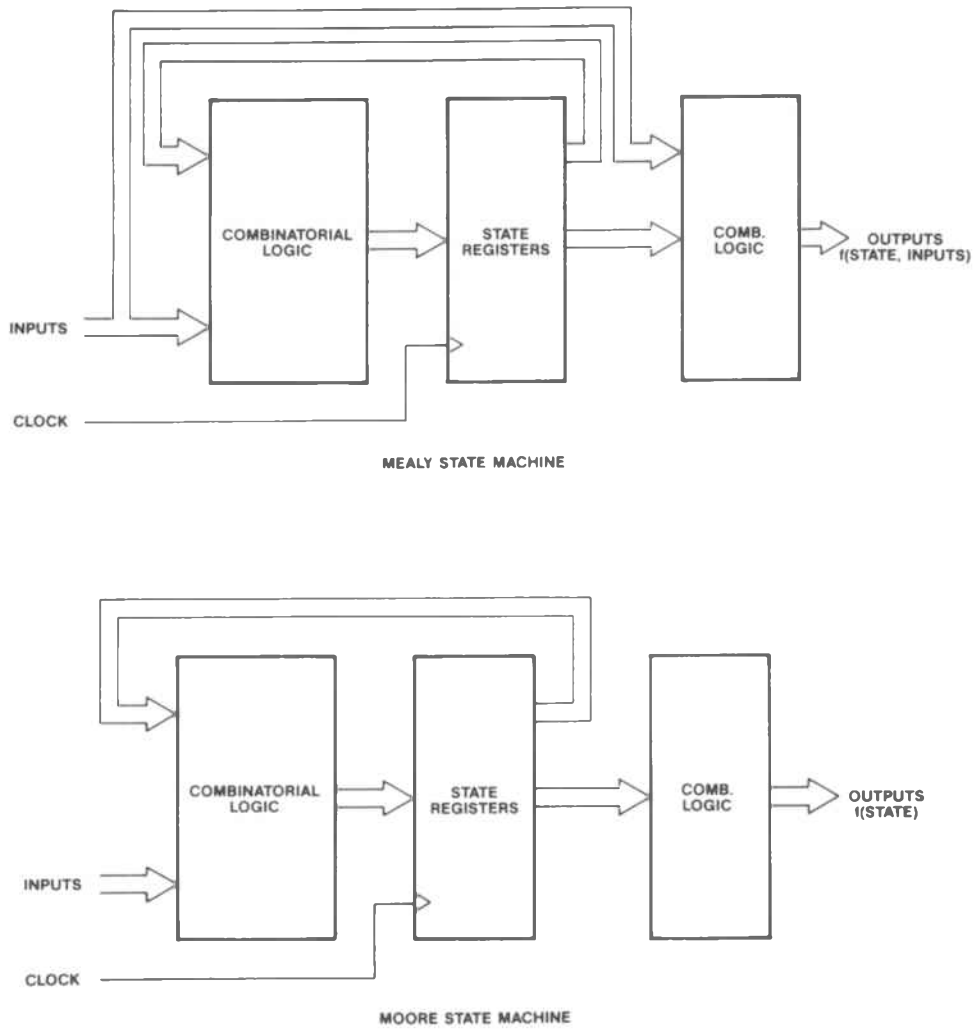


Asynchronous state machine design is in general a much more difficult task than synchronous design. Since any input may trigger a transition, the potential for spurious or false transitions (caused by logic **hazards** or just unforseen logical combinations) is much higher. These machines may be higher performance under certain circumstances, but this practical difficulty restricts their use to a relatively small percentage of all designs.

MEALY AND MOORE MACHINES

Mealy and Moore machines are two categories

Figure 4. Types of Synchronous State Machine



of state machines with different output characteristics. In a **Mealy machine**, the outputs at any given time are a function of the current machine state and current inputs. As a result, multiple output combinations are possible for any given state. In a **Moore machine**, outputs are a function only of current state. Thus, the outputs are still dependent on the previous input history as held in the state, but are not directly affected by input values.

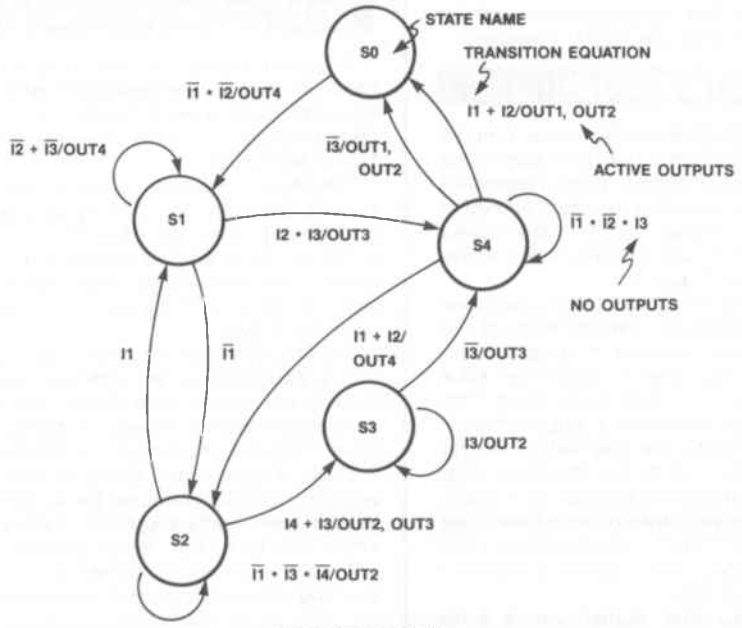
Figure 4 shows examples of each of these machines in block form.

Moore machines are a sub-class of Mealy machines. That an equivalent Moore machine can be defined for any Mealy machine is not obvious. This will be described below in the section on Mealy-Moore Transformation.

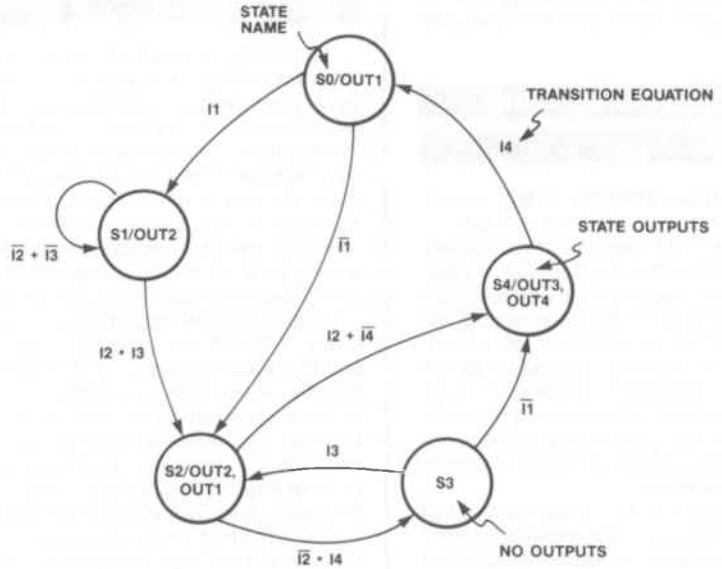
STATE DIAGRAMS AND TRANSITION FUNCTIONS

Describing and documenting the state machine's operation is an important part of the overall task.

Figure 5. State Transition Graphs



MEALY STATE GRAPH



MOORE STATE GRAPH

There are three primary mechanisms for documenting or describing a state machine's operation: state diagram, state transition table, and high-level state machine language. State diagrams take two common forms, that of a state transition graph or an Algorithmic State Machine (ASM) flowchart.

STATE TRANSITION GRAPH

The **state transition graph** consists of a directed graph which specifies states, transition conditions and outputs for a state machine. Circles represent states in the machine, and arcs represent allowable transitions between states. Input combinations which cause transitions are typically noted above the arcs, as shown in Figure 5.

Output notation for Mealy and Moore machines differs due to the different characteristics of the machines. For a Moore machine, outputs, a function only of state, are written within the state circle, separated from the state name by a slash ("/"). Mealy machines, which have outputs dependent on state and inputs, are graphed by writing output combinations next to the transition arcs, separated from the transition function by a slash. It is possible to have two transition arcs from State A to State B if in fact different input combinations result in different output combinations concurrent with the transition to State B.

Both syncrhonous and asynchronous state machines may be described in this manner. By convention, synchronous state machines do not explicitly note the Clock on the state diagram: all transitions are assumed to be synchronous with the Clock edge.

ALGORITHMIC STATE MACHINE DESCRIPTION

The **Algorithmic State Machine (ASM)** format for describing state machine behavior is a form of flowchart notation. In ASM, there are three primary symbols, conditional output (cylindrical shape), unconditional output (rectangular) and decision blocks (diamond). A machine state in such notation may actually consist of a collection of blocks, unlike state transition graphs discussed above where each circle represents a state. A state generally involves one unconditional output block, followed by optional decision boxes (conditional state transitions) and possibly conditional output blocks (for Mealy machines).

Each decision block in ASM represents a test of a given input condition and a corresponding true/false decision. The condition or inputs being tested is normally noted inside the box. 0 or 1 branch paths from the diamond represent false or true results obtained from this test. Multiple decision boxes which are connected to each other are evaluated in parallel: only when a new rectangle is encountered while traversing the flow chart is a new state "entered". Figure 6 illustrates this notation.

STATE TRANSITION TABLE

A second method for describing state machine behavior is the **state transition table**. The state transition table, as with the state transition graph, takes two forms, one for Mealy machines, one for Moore machines.

The Mealy form has a present state column on the left. Beneath the heading all possible states are noted. To the right of the present state section is the inputs header. Underneath the inputs are shown next state values, and outputs associated with the transition at the top of the column. This is shown in Figure 7.

The Moore machine transition table varies in the output specification. In a Moore machine table, outputs are shown immediately after the present state entries, and ommitted from the inputs section. Other than this, the tables are identical.

Either diagrammatic entry or table entry can become quite cumbersome for larger designs, although their "nuts-and-bolts" nature can bring added clarity to the design process. High-level descriptions provide a conceptual machine model, avoiding details of machine implementation.

HIGH-LEVEL STATE MACHINE DESCRIPTION

Describing a state machine in a high-level language makes the design readily understandable by anyone familiar with computer programming. Machine flow, or operation, is modeled in an algorithmic manner, using simple syntax and semantics. The requirements for such a specification include machine inputs and outputs declarations, state declaration and state-to-state-variable mapping, and state transition specifications. Figure 8 shows a sample state machine description using Altera State Machine Input Language (ASMILE).

State transition specification can take many forms. ASMILE, for example, supports IF...THEN and CASE constructs, as well as Truth Table Entry. Usually, entry form is a matter of personal preference, although certain classes of problems may be more appropriately handled by one means or another. For example, machines which have a high degree of regularity to their operation are frequently specified using Truth Tables.

Differentiating Mealy and Moore machines in a high-level language description once again revolves around output specification. The Outputs subsections in an ASMILE file are associated with the state transition specifications. Two types of outputs are allowed, conditional and unconditional. Unconditional outputs are associated with a given

Figure 6. ASM Diagram

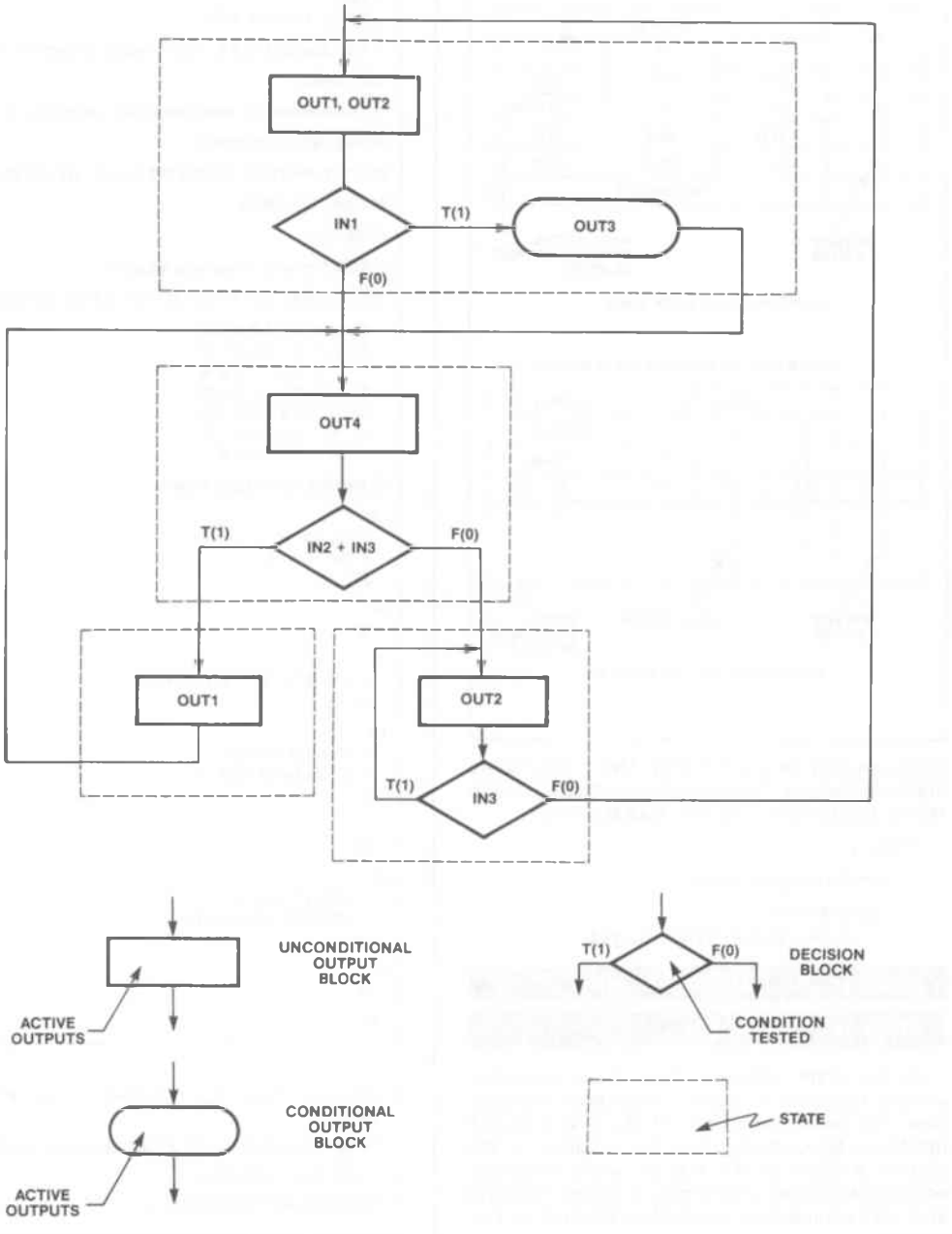


Figure 7. State Tables

INPUTS (MAY BE COMPLEX EXPRESSIONS)

PS	INPUTS		
	A	B	C
S0	S1,00	S2,01	S3,10
S1	S1,00	S2,01	S1,11
S2	S3,01	S0,10	S1,10
S3	S0,11	S0,10	S3,01
	NS, OUTPUTS		

PRESENT STATES

NEXT STATES, INPUT-DEPENDENT OUTPUTS

MEALY MACHINE STATE TABLE

INPUTS (MAY BE COMPLEX EXPRESSIONS)

PS	INPUTS			OUTPUTS
	A	B	C	OUT1, OUT2
S0	S1	S2	S3	00
S1	S1	S2	S1	01
S2	S3	S0	S1	10
S3	S0	S0	S3	11
	NS			

PRESENT STATES

NEXT STATES

STATE-DEPENDENT OUTPUTS

MOORE MACHINE STATE TABLE

state and may be used in both Mealy and Moore machine designs. Conditional outputs are used in Mealy descriptions only and take the form

```
State_i:
    transition specification
        OUTPUTS:
            IF expression THEN output
```

PRACTICAL STATE MACHINE ARCHITECTURES

So far, state machines have been discussed without reference to actual circuit-level architecture. The broad definitions of Mealy and Moore machines leave much room for variation in the implementation of the design while obtaining equivalent behavior. Performance, integration-level and cost are practical parameters affected by this choice.

The most common choices available for implementing state machines include:

- SSI/MSI

Figure 8. ASMILE Input Format

```
STAN KOPEC
ALTERA CORP.
3/10/87
68020 Bus Arbiter for SAM

% This description uses IF...THEN Transition Specifications %

PART: EPS444

% Pin Assignments (an option) are made by the designer %

INPUTS: REQUEST@1 ACK@2

OUTPUTS: GRANT@23 TRISTATE@22 OS0 OS1 OS2 OS3 OS4 OS5 OS6

MACHINE: BUSARBITER

CLOCK: CLK

% STATES gives the output value mapping %

STATES: [GRANT TRISTATE OS0 OS1 OS2 OS3 OS4 OS5 OS6]

  S0  [0 0 1 0 0 0 0 0 0]
  S1  [1 1 0 1 0 0 0 0 0]
  S2  [1 1 0 0 1 0 0 0 0]
  S3  [1 1 0 0 0 1 0 0 0]
  S4  [1 1 0 0 0 0 1 0 0]
  S5  [0 1 0 0 1 0 0 1 0]
  S6  [0 1 0 0 1 0 0 0 1]

% Transition Specifications follow %

S0:
  IF REQUEST*/ACK THEN S1
  IF ACK THEN S5
  S0

S1:
  S2

S2:
  IF /REQUEST*/ACK + ACK THEN S6
  S2

S3:
  IF /REQUEST THEN S6
  IF REQUEST*/ACK THEN S2
  S3

S4:
  S3

S5:
  IF REQUEST THEN S4
  IF /REQUEST*/ACK THEN S0
  S5

S6:
  S5

END$
```

- Programmble Logic Devices (PLAs, PALs and EPLDs)
- Programmable Read-Only Memory (PROM)
- Multi-Chip Bit-Slice
- Microcoded Sequencers

SSI/MSI

SSI/MSI state machine implementations give the lowest level of design integration. The canonical

state machine forms discussed earlier can be mapped onto an SSI/MSI implementation, but the number of components required limits this technology to very small-scale machine designs. State variables are implemented using individual edge-triggered flip-flops (7474 or the like), and transition terms generated with an assortment of small-scale gates. Depending on SSI/MSI technology employed (LS/FAST/HCT/etc.) performance can be fair to good.

Efficient design tools are not available due to the inconsistent architectural structure of SSI/MSI implementations. Automatic logic minimization must target a specific logic structure to give a minimal solution. The tool issue becomes a very real road-block when constructing larger designs, and the designer is by-in-large constrained to using hand techniques.

PROGRAMMABLE LOGIC DEVICES

Programmable Logic Devices (PLDs) employ architectures which implement canonical state machine structure quite well. Figure 9 shows a general PLD block diagram. The benefits of PLDs when used for state machines include:

- High Gate Count
- High Flip-Flop Count
- Programmable I/O
- Programmable Flip-Flops
- Structured Architecture

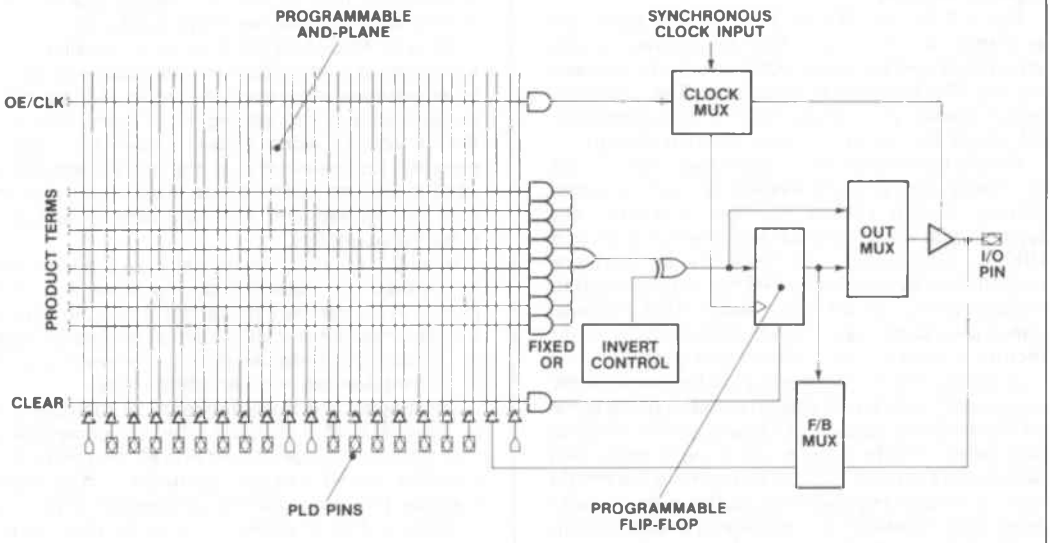
Not all devices provide all of these benefits (older PALs do not support Programmable I/O and Flip-Flops) but the overall fit is a good one for many state machine applications.

In targeting a state machine design for a PLD, key resources include flip-flops, I/O, and product-terms. Product-terms represent single-term logical ANDs of any PLD inputs and/or flip-flop outputs. The connections into these terms may be any set of true or complemented variables as defined by the user in the design. Since product-terms represent a measure of the combinatorial logic capability available in a PLD, their total number and distribution is an important consideration.

PLAs have both programmable AND and programmable OR logic planes. As a result, product-terms may be distributed in any fashion among state variable (flip-flop) inputs and machine outputs. Product-terms may even be shared between flip-flops or outputs. Programmable-AND/Fixed-OR PLDs (such as PALs) have a fixed number of product-terms associated with each flip-flop, eight product-terms is a good average. If more than the allotted number of product-terms is required for a given function, one or more cells may be used to expand the product-term count at the cost of an additional array delay.
of an additional array delay.

More advanced PLDs have programmable flip-flop types, wherein the type (D/T/JK/SR) of flip-flop can be selected by the user on a cell-by-cell basis. This can be very valuable when designing state machines, because certain functions require less product-terms when implemented with D flip-flops instead of T flip-flops. The converse can also be true. More discussion on flip-flop type will be

Figure 9. PLD Structure



found in the State Machine Synthesis section.

Altera EPLDs provide a user-configurable selection of features from which any state machine can benefit. Figure 10 shows a comparison of Altera's EPLDs with other popular PLDs used for state machine applications. The Altera EPLD family concept allows the user to select the device appropriate for his problem and still obtain, in most cases, a single-chip solution.

Figure 10. Logic-Array State Machines

DEVICE	INPUTS	OUTPUTS	POSSIBLE STATES	P-TERMS	FLIP-FLOP TYPE
82S105	16	8	64	48	SR
PLS167	14	6	64	48	SR
16R8	16	8	16+	64	D
22V10	11(21)	10	32+	120	D
EP600	4(20)	16	256+	128	D/T/JK
EP900	12(36)	24	4K+	240	D/T/JK

+ = Half Used for State Variables/Half for Output Functions.

PROM-BASED STATE MACHINES

PROMs can be used to generate very efficient state machines, if the machine can be implemented within the constraints of the PROM architecture. PROMs satisfy the basic requirement for memory elements within the state machine: thousands of memory locations and potential states can be found in a single PROM component. Typically an edge-triggered register is used at the PROM outputs to synchronize machine operation. This is called a **Pipeline Register**.

For a 2**n byte PROM, 2**n product terms are available with n inputs. This exhaustive decode may be viewed as a fixed-AND structure, followed by the PROM memory locations. The programmable memory locations form a programmable-OR plane for use in the state machine design.

PROM-based state machines have a set number of inputs and outputs associated with a single device. Output counts of eight maximum are typical, with eleven inputs available for a 2K × 8 PROM. Single-chip I/O flexibility is therefore limited. However, multiple PROMs may be used to expand output count fairly easily. This process, called **cascading**, uses the PROM's memory architecture to advantage as shown in Figure 11.

A PROM has no internal signal feedback paths. A typical PROM-based design involves using some of the PROM's outputs as state machine outputs, and other PROM outputs as a next state field which can feedback state information to the PROM inputs. This arrangement decouples output values from state transition functions and allows the implementation of Moore machine architectures directly. Valuable input pins are consumed closing the feedback loop, however.

MULTI-CHIP BIT-SLICE

Bit-slice-based sequencers represent an extension of the PROM-based state machine approach, but with higher-level sequencing features included to handle high-complexity tasks. These higher-level functions are integrated into a single LSI chip called a bit-slice controller. The bit-slice controller's job is to control sequencing (addressing) of external microcode memory. This memory may be PROM (as above), or EPROM or static RAM. The bit-slice controller typically has resources such as a microprogram counter, loop counter, and stack integrated within it. Bit-slice controllers introduce the notion of Microinstructions.

To effect a state branch on external conditions, bit-slice controllers have a single external condition code input which may be tested under microinstruction control. This limits branching to a two-way yes/no decision. Complex branches (multi-way) must typically be executed by a series of such simple branches.

Microinstructions are carried in each microcode memory location along with output vectors. They control the method used for next state selection. Classes of microinstructions include conditional branching on external inputs or internal conditions, initializing state variables, or temporary storage of current state information.

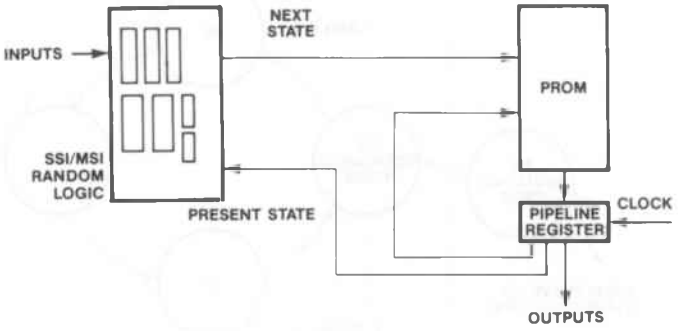
As mentioned, functions such as counters or LIFO stacks are incorporated into these controllers. Counters permit implementation of repetitive state sequences by the construction of microinstruction loops, while stacks permit saving state information temporarily, as in the case of nested machines or subroutines (see Figure 12). These ease the implementation of complex state machines.

As with PROM-based machines, bit-slice controller designs are ideal for state machines which must generate many control outputs. Adding more outputs amounts to adding more memory bits to the microcode word. Output values are held in memory independently of the microinstructions used to control machine flow. As a result, outputs may be defined and modified without affecting machine operation.

In addition, output entry is simple. If a logic one is required on Output X during State A, a 1 is placed in the memory bit corresponding to Output X at the memory location corresponding to State A. Decoding of state variables to generate outputs is not required as in other approaches.

Once again, the main drawback of such designs is their lack of integration. Twenty components may typically be used to provide all the pieces of a bit-slice based design: controller, microcode memory, pipeline register and condition logic. The solution to this problem is the single-chip microcoded sequencer.

Figure 11. PROM-Based State Machine



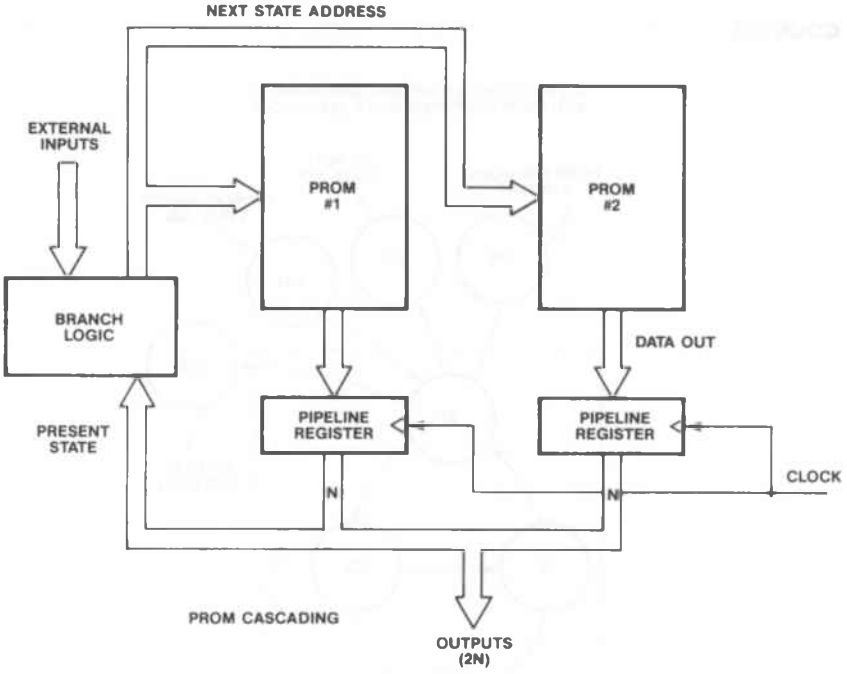
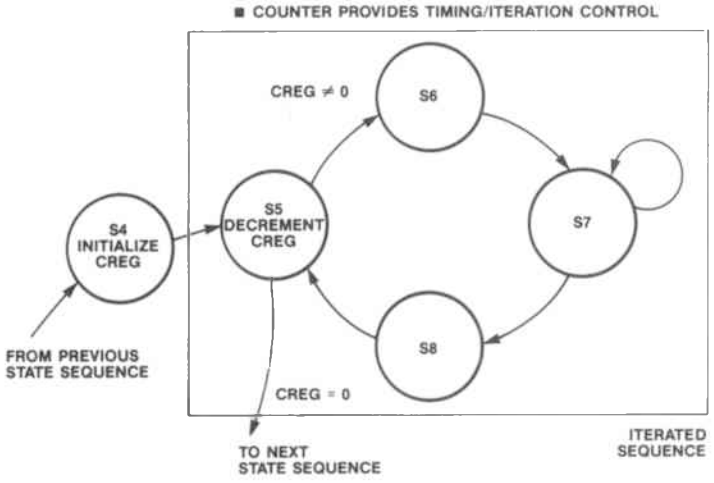
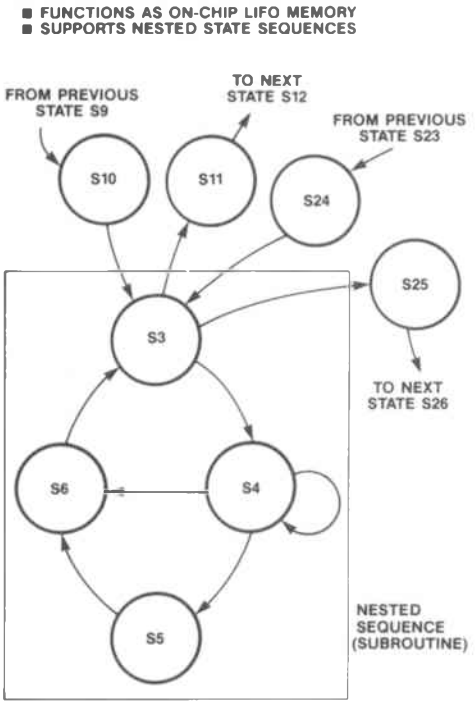


Figure 12. Counter and Stack

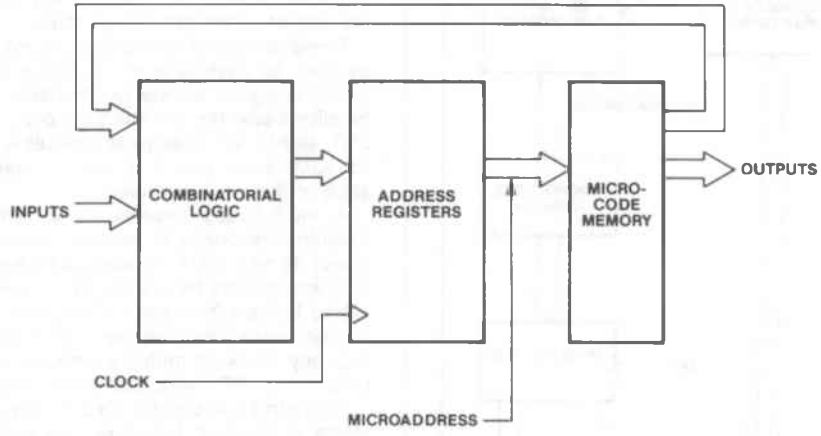


COUNTER



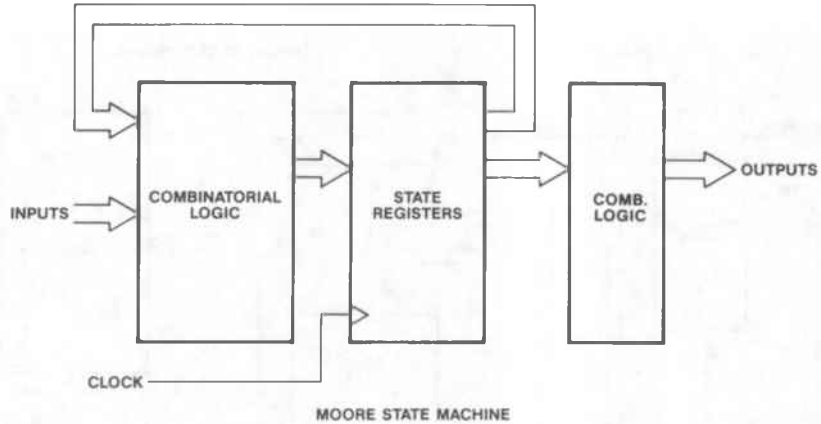
STACK

Figure 13. Microcoded Output Structure



MICROCODED STATE MACHINE
- MICROCODED OUTPUTS PROVIDE:
  - Easy output count expansion
  - Complex waveforms

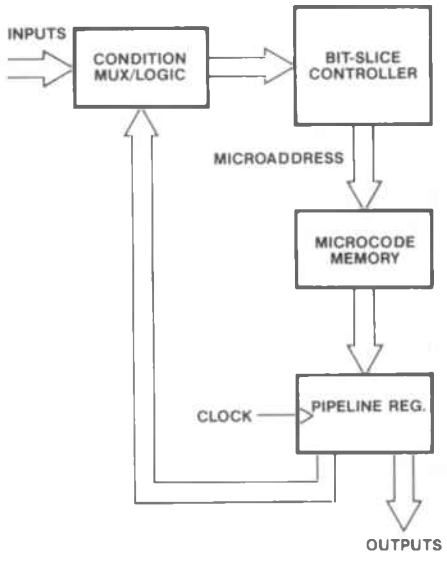
    Without p-term consumption



MOORE STATE MACHINE

5

Figure 14. Bit-Slice Machine



MICROCODED SEQUENCERS

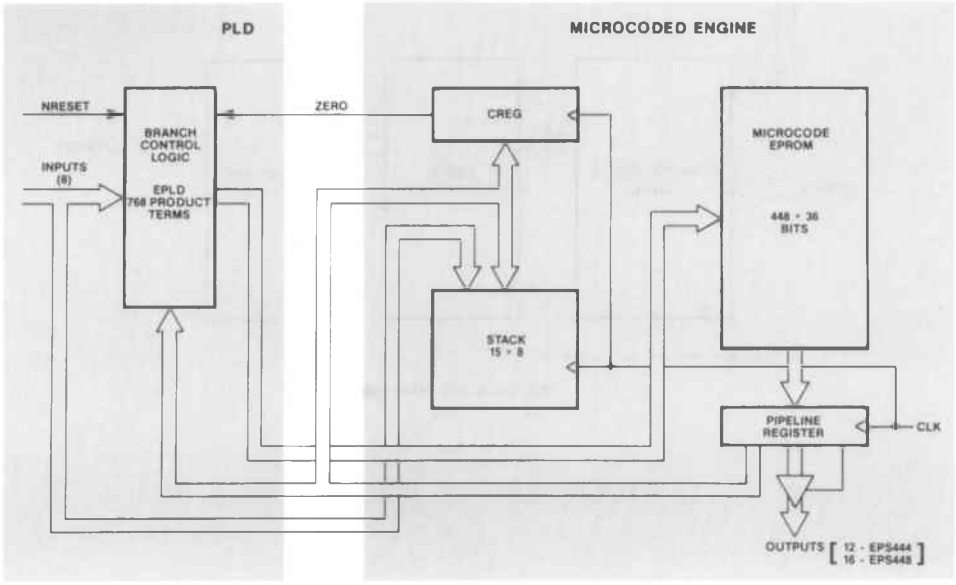
Microcoded Sequencers provide a single-chip microcoded state machine vehicle. The benefits of this approach include user programmability, higher performance, less p.c. board area, lower power, and a consistent architecture allowing the development of optimized design tools.

These devices range from simple PAL plus PROM devices, to devices such as Altera's SAM family which integrate all the functionality present in a bit-slice-based design. The SAM device, with nearly 500 words of user-programmable microcode EPROM, stack and loop counter represents the state-of-the-art in this area.

A microcoded sequencer (as with any state machine) responds to external inputs in order to select its next state. As discussed earlier, bit-slice microcontrollers have a single input which can be tested for zero/one and a branch executed based on the result. One step up from this approach is logically masking multiple external inputs with a predefined AND mask, and testing the result under microinstruction control. One of two possible next states is selected based on the outcome of the test. Both of these schemes allow only two-way state branching in a single clock.

The most advanced approach, utilized in Altera's SAM device family, inputs multiple external signals

Figure 15. SAM Block Diagram



into a PLD "front-end" of the microsequencer. This PLD is coded with user-defined transition equations which are functions of current machine state and inputs. The output of the PLD selects one of four destination states, giving four-way single-clock state machine branching. In addition, these transition functions are prioritized, implementing the priority implied in a state transition statement such as

```
IF (expression1) THEN State_A
ELSEIF (expression2) THEN STATE_B
ELSEIF (expression3) THEN STATE_C
ELSE STATE_D
```

STATE_D is a default transition which is executed if none of the above expressions is true.

This multi-way branch capability, combined with SAM's 15-level stack, counter and microinstructions gives an excellent vehicle for complex designs.

Figure 16 shows a features comparison for Altera's SAM device compared to the other available single-chip microsequencer options. The SAM devices achieve higher density, higher performance and lower power than any other available option.

Figure 16. Microcoded Sequencer Comparison

FEATURE	29PL141	14R21	SAM
Memory Words	64	128	448
Outputs	16	8	16
Inputs	7	8	8
Loop Counter	YES	NO	YES
Stack	2×6	None	15×8
Branching	2-Way	4-Way	4-Way
Instruction-Based	YES	NO	YES
Package	28/0.6	20/0.3	28/0.3

STATE MACHINE SYNTHESIS

This section will discuss general state machine design concepts. Specific design issues, such as using a particular device's microinstruction set to full advantage, will not be covered. The Altera literature reference below covers these specific topics.

The state machine design process begins with thoroughly defining "... what the machine does." Writing a verbal description of what job the machine performs in the overall system is a good way to start. Describing other blocks it will communicate with and associated inputs and outputs defines the external interface characteristics. Noting critical timing constraints and required sequences of operations (our rudimentary states) describes the internal operation of the machine.

Next, the process of translating the design into one of the standard state machine documentation forms mentioned earlier can begin. In general, ASM notation and high-level language descriptions are preferred for larger designs, particularly those with greater than ten states.

Initially, the state machine description should be generated without undue worry about state minimization or overall efficiency. Comprehensiveness of the description is the first goal. Minimization techniques (to be described next) will be employed to compact the design as a second step.

STATE MINIMIZATION

State minimization is a process of reducing the total number of states in a state machine to a minimum. Minimization is based on a process of identifying **equivalent states** and combining them into a single state. Two (or more) states are said to be equivalent if the state machine generates identical output sequences for all input sequences applied, given any of the equivalent states as a starting point. The states are therefore not **distinguishable**.

The process for discovering equivalent states is an iterative search for states which produce identical output sequences for all input sequences. Using a state table notation is the clearest means of presenting state information for this analysis. Figure 17 shows a simple state machine before and after reduction. The process involved consists of:

1. Locate all states that generate the same output for all input vectors (directly from the initial state table) and construct initial state groups. This grouping is called a **partition**.
2. Analyze each group over all input combinations to determine if the next states reachable from the group are equivalent as measured by the previous partition. If different, a new partition is required: the group is subdivided.
3. The process is continued until the partition at step i + 1 is the same as the partition at step i. At this point, the states are minimized. The states in each group are equivalent and can be reduced to a single state.

For the example shown in Figure 11, the first partition isolates S4 (it is the only state from which a one output can be obtained). S0 and S5 are isolated from S1-S3 and S6 in the second partition since they have S4 as a possible next state. S1 and S3 are isolated from S2 and S6 in the third partition because they have S0 as a potential next state. Finally, S0 and S5 are divided because they have S2 and S3 as possible next states, which are not grouped together in the previous partition (implying S2 and S3 are distinguishable).

This process can become tedious for large designs. Most designers utilize the technique on smaller sub-machines before integrating them into

Figure 17. State Table Minimization

Unminimized Machine

PS	NS, OUTPUT	
	INPUT = 0	INPUT = 1
S0	S4,0	S2,0
S1	S2,0	S0,0
S2	S1,0	S6,0
S3	S6,0	S0,0
S4	S5,1	S1,0
S5	S4,0	S3,0
S6	S3,0	S6,0

Initial = (S0,S1,S2,S3,S4,S5,S6)
1st Partition = (S0,S1,S2,S3,S5,S6) (S4)
2nd = (S0,S5) (S1,S2,S3,S6) (S4)
3rd = (S0,S5) (S1,S3) (S2,S6) (S4)
4th = (S0) (S5) (S1,S3) (S2,S6) (S4)
5th = (S0) (S5) (S1,S3) (S2,S6) (S4)

Minimized Machine

PS	NS, OUTPUT	
	INPUT = 0	INPUT = 1
S0	S4,0	S2/6,0
S1/3	S2/6,0	S0,0
S2/6	S1/3,0	S2/6,0
S4	S5,1	S1,0
S5	S4,0	S3,0

a larger design. The minimization process remains manageable when approached this way.

STATE MACHINE PARTITIONING

Partitioning complex state machine designs into smaller machines can utilize state machine device resources more effectively. For example, it was mentioned earlier that PLD devices have a fixed number of product-terms per macrocell. For complex machines, the equations for a state variable input can require a large number of product terms. By partitioning a large state machine into smaller state machines, the individual equations can be reduced in complexity, albeit at an increase in total number of state variables.

Figure 18 shows a state machine which can be divided into two simpler state machines. The idea behind this partition is that only one of the two sub-machines is active at any one time, the other being idle. Control transfer is passed back and forth between machines each time the original dividing line is crossed.

In the figure, states SA and SB are these idle states. Transitions leaving the idle states are dependent on both the original transition equation and current state of the other sub-machine. The holding transitions for the idle states have equations that are simply the logical inverse of all transitions leaving each idle state.

MEALY-MOORE MACHINE TRANSFORMATION

Mealy machines may be converted to equivalent Moore machines. The rules for this transformation are straightforward: for each transition into a state in the Mealy machine with a different output value, define a separate state in the Moore machine. The Moore machine produced has identical input-output behavior to the original Mealy machine. Figure 20 illustrates this process for a simple state diagram.

Even though a Mealy machine may be synchronously clocked, outputs from the machine may respond to input transitions without reference to the clock. If such a Mealy machine has been designed, conversion may affect a.c. timing. This should be analyzed once the design is completed for any system impact.

Mealy/Moore conversion can mean an increase in the number of required states for the machine. Conversion is frequently employed when designs originally targeted for PLD-based state machines are redesigned for implementation in a PROM-based or microcoded sequencer design.

IMPLEMENTING THE STATE MACHINE

The final steps of implementing a state machine consists of:

- Minimizing transition logic
- Select a target device or set of devices
- Select flip-flop type (optional)
- Assign state variables (optional)
- Enter design

Minimizing transition logic may be done by hand, or automatically provided by a logic compiler such as A+PLUS (Altera Programmable Logic User System). Flip-flop type is relevant only if an SSI/MSI implementation or PLD with programmable flip-flop type is available, such as Altera's EP600/900/1800 family. State variable assignment is also required only in SSI/MSI designs or PLD designs: PROM or microsequencer designs map states onto memory locations and do not have state variables requiring definition. Where programmable devices are the target vehicle, the last step is to enter the design into the logic compiler for that device. After design processing, the compiler generates a JEDEC programming file to allow programming of the state machine component.

Figure 18. State Machine Partitioning Example

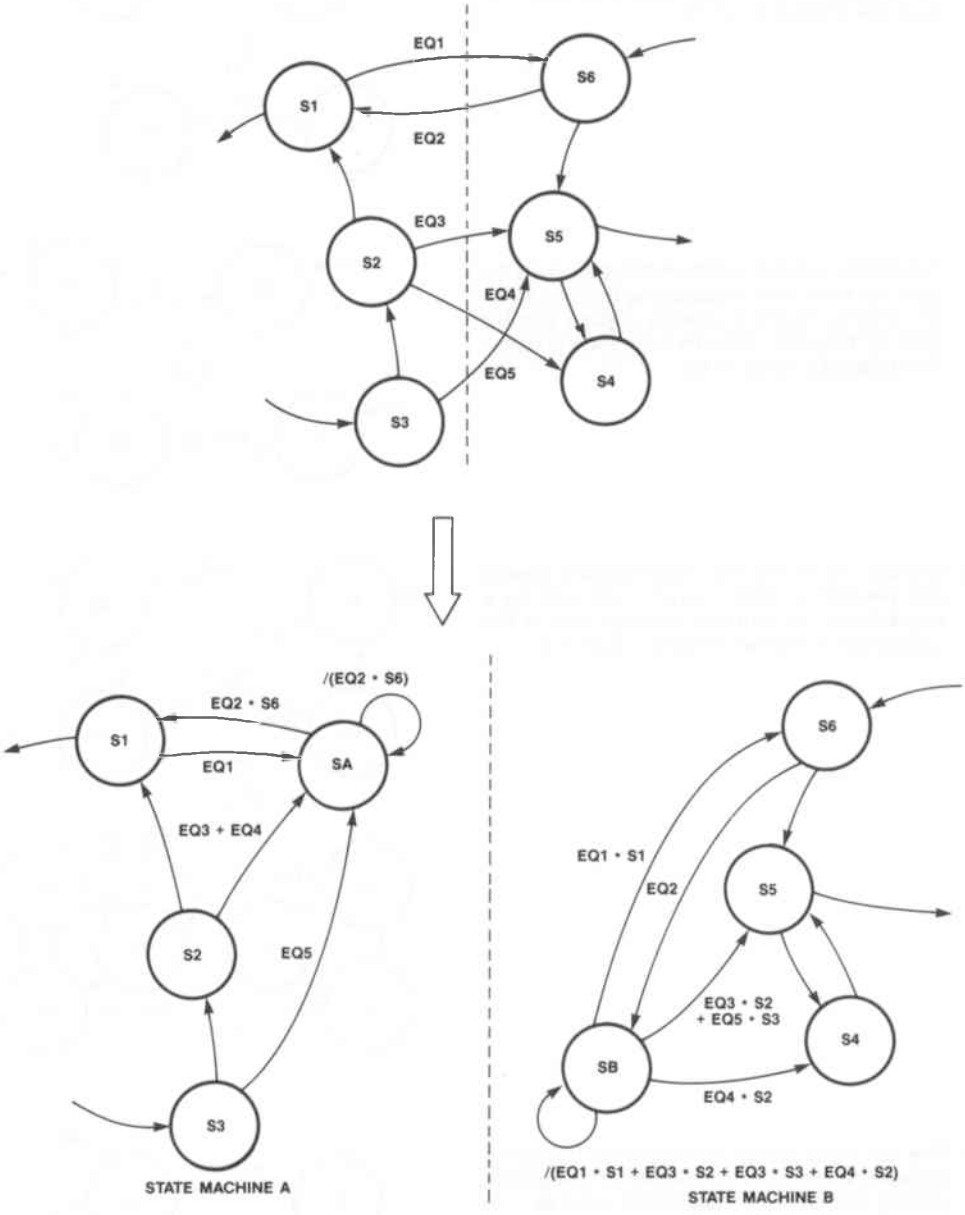
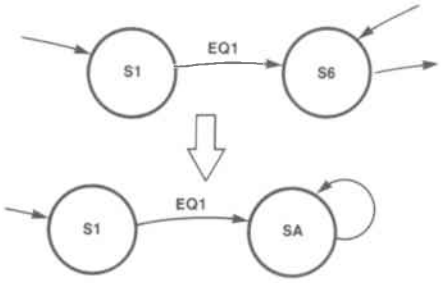
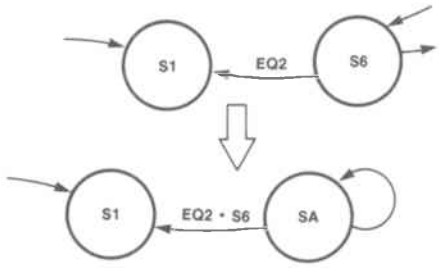


Figure 19. State Machine Partitioning Rules

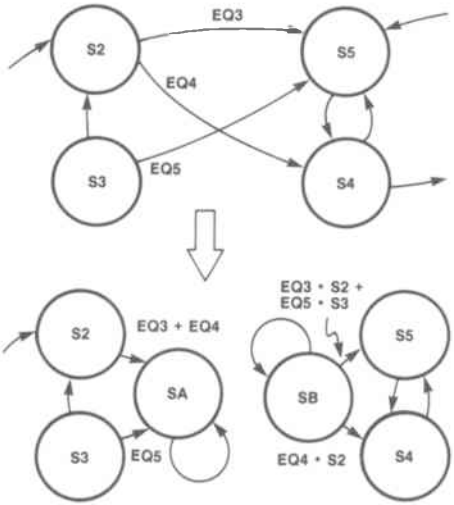
1. Transitions leading into an idle state keep the same equation as the corresponding transition from the original machine.



2. Transitions leading out of an idle state have the equation from the corresponding transition from the original machine, logically ANDed with the state (of the other machine half) that the transition originally came from.



3. Multiple transitions that have the same source and destination states may be replaced by a single transition with an equation that is the logical OR of the two original equations.



4. The idle states have a hold equation that is the logical inverse of all equations for transitions leaving that state logically ORed together.

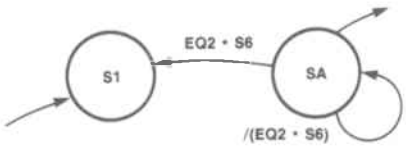
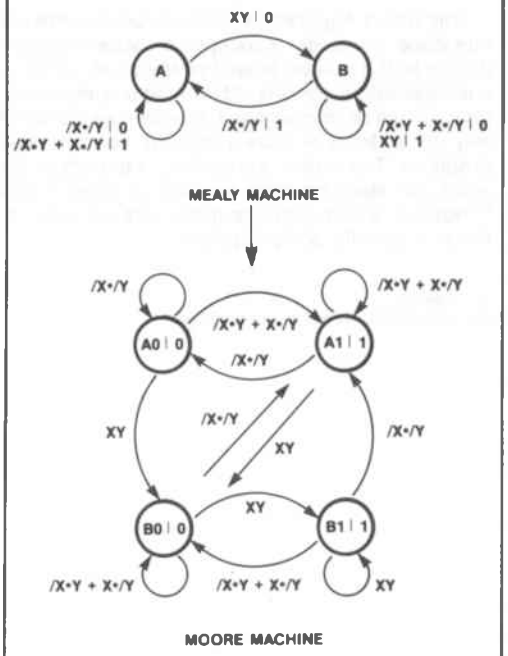


Figure 20. MEALY/MOORE Transformation



FLIP-FLOP TYPES

For state machines targeted for SSI/MSI or PLD implementations, flip-flop type can be an issue in selecting the minimal state machine implementation. As mentioned earlier, several Altera EPLDs allow the selection of either D, T, JK or SR operation for the device macrocells. Depending on the state machine, either D or T flip-flops can give the minimal design. It can be shown that JK (and therefore SR flip-flops) give results no better than these.

Figure 21 shows the excitation table for the various flip-flops. In the table, a one means a variable must be excited, a zero means the variable must not be excited, and an X means the excitation does not matter. Note that D and T columns all have two "1"s in their transition specifications. J and K columns only have one "1", and at first glance these seem as though they might require fewer excitation terms. However, since both J and K inputs are required for any such flip-flop, the number of ones is in fact two for both J and K inputs, and no economy is obtained over D or T flip-flops.

Figure 21. Flip-Flop Excitation Functions

STATE VARIABLE TRANSITION	FLIP-FLOP TYPE			
	D	T	J	K
0 → 0	0	0	0	X
0 → 1	1	1	1	X
1 → 0	0	1	X	1
1 → 1	1	0	X	0

By analyzing state transition expressions in a state machine description, it is possible to determine the optimal flip-flop type for a given design. Several methods for performing this trade-off are described in the references below. This is also connected with the problem of state variable assignment described in the next section. To assist the designer, Altera's ASMILE state machine language processor automatically performs optimum flip-flop selection during the compilation of a high-level state machine specification.

SELECTING STATE ASSIGNMENTS

State assignment is the process of associating specific state variable codings to the various states in a state machine. For a state machine with M states, N state variables are sufficient assuming 2**N is greater than or equal to M. A clever state assignment can minimize required transition logic.

For example, T flip-flops may be used as state variables. There is a probable reduction in transition logic complexity if states that have a high degree of connectivity (transitions between them are functions of many product terms) have state variable codings which are as similar as possible. This means the codings may differ in only one or two state variables. The amount of variable toggling required to transition is therefore reduced. This is illustrated in Figure 22.

Figure 22. State Variable Assignment

STATE	STATE VARIABLES	
	Q0	Q1
S0	0	0
S1	0	1

CASE A

STATE	STATE VARIABLES	
	Q0	Q1
S0	0	0
S1	①	1

CASE B

CASE B REQUIRES ACTIVE LOGIC TERMS FOR BOTH Q0 & Q1 ON S0 → S1 TRANSITION

As mentioned earlier, increasing state variables can reduce individual flip-flop transition function complexity. Particularly if state variables are used as machine outputs, adding variables can be a very viable way to simplify the design. In the degenerate case (used quite frequently for simple machines) each state may be represented by a single state variable.

5

The references below provide some methods for attacking the state assignment problem. There exists no concise, exhaustive means of defining a minimal state assignment. State assignment is, at a practical level, one of those parameters juggled when fitting a tight state machine design. Otherwise, assignments tend to be less rigorously defined for all but the simplest designs.

Figure 23. Adding State Variables

STATE	STATE VARIABLES		
	Q0	Q1	Q2
S0	0	0	0
S1	0	1	0
S2	1	0	0
S3	1	1	0
S4	0	0	1

MINIMUM STATE VARIABLES — REQUIRES STATE DECODING

STATE	STATE VARIABLES				
	Q0	Q1	Q2	Q3	Q4
S0	1	0	0	0	0
S1	0	1	0	0	0
S2	0	0	1	0	0
S3	0	0	0	1	0
S4	0	0	0	0	1

MAXIMUM STATE VARIABLES — USES EXTRA FLIP-FLOPS

CONCLUSION

This Altera Application Note has been written to introduce the basic concepts of state machine design to the reader, as well as the most common implementation options. State machine methodology, as has been discussed, provides a structured way to address a wide range of control logic problems. The reader is encouraged to consult the additional state machine application notes in this handbook which provides more detailed descriptions of specific design options.

AN7 Rev 2.0

INTRODUCTION

State machine design entry provides a high-level approach to synchronous logic design. This Application Brief provides recommendations for reliable state machine design with Altera EPLDs.

POWER-UP ZERO

Altera EPLDs automatically clear all flip-flop outputs to a logical '0' on power-up. Every state machine must begin in a state which all state variables are set to zero. Be sure to define in an "all zero" power-up state in the state table.

TRAP ILLEGAL STATES

State machines should be designed so they cannot get stuck in an illegal state. Either all possible combinations should be defined in the STATES: section of the state machine file, or the machine should reset any time an illegal state is entered. The following code, added to the state machine file, will clear the machine when an illegal state is entered.

```
NETWORK:
CLEAR = NORF(ILLEGAL,CLOCK,GND,GND)
EQUATIONS:
ILLEGAL = /(STATE0 + STATE1 + ... +
                                STATEN);
```

The Clear variable in the network section provides the asynchronous clear function to the machine. The input to the register (ILLEGAL), is defined as the NOT of all the possible legal states. Thus, if an illegal state is entered, ILLEGAL = 1 and all state variables are reset to "0".

ASYNCHRONOUS INPUTS

To assure reliable operation of any state machine, care should be taken when using asynchronous inputs. An asynchronous input signal, which may violate a state register setup time, can cause a machine to enter an illegal state. Whenever possible, the inputs to a state machine should be synchronized with the machines' clock so that all signals are guaranteed to meet the input setup times.

When synchronization of inputs is not possible, adhere to the following rules:

1. Asynchronous inputs that determine the next state must be twice as wide as the state-machine clock.

   If an asynchronous input lasts for less than two clock periods, there is some probability it will be missed by the machine.
2. A state should change by only one bit in moving from one state to a next if the change depends on asynchronous inputs.

   If more than one state register changes value between states, the asynchronous input may meet the setup time on one register, but miss the setup time of the other register. The machine would then enter an illegal state.
3. State transitions should never depend on more than one asynchronous input.

   If a multi-way branch depends on several asynchronous inputs, both branches could appear valid. The machine would then enter an illegal state.

REDUCING PRODUCT TERM CONSTRAINTS

As the complexity of a state machine increases, product term demands increase as well. When product term needs exceed the number available in a device, the A+PLUS software will produce the error message, "Too Many P-Terms".

When this message occurs, the state machine, as it now stands, will not fit into the selected EPLD. Either a new device must be selected, or the design must be slightly modified.

The SAM family of EPLDs have been optimized for state machine applications and offers the surest relief from product term limitations. The first member of this family, the EPS448, can implement up to 448 states with no practical limitation on the sequencing complexity. It also offers 768 product terms to help define transition conditions (for more information on this device, refer to the Altera EPLD Databook and the SAM section of this HandBook).

If a general-purpose EPLD is chosen, the following techniques will help fit the machine.

Change State Variable Assignments

If the machine will not fit, first change the state variable assignments for all buried state variables. Different state variable assignments can result in simpler equations for a given machine. As a general rule, try to minimize the number of 1's or minimize the number of transitions for a given column in the state assignment section.

Adding State Variables

Adding a buried state variable doubles the number of possible states but the equations leading to each state variable may be simplified. After adding the state variable, try to change the state variable assignments as described above.

Remove Large Counters and Shift Registers

The LogiCaps schematic capture program provides a powerful complement to the state machine

software. For fast and efficient design, they should be used together; with each being used where it is most applicable. For example, when designing large counters and shift registers, a schematic approach proves to be more efficient. State machines, prove superior for a wide range of control functions, including control of a counter or shift register defined in a schematic.

Partition State Machine.

When faced with a large state machine of 10–30 states that must fit into an EPLD, the machine should probably be partitioned into 2 smaller machines. Partitioning eases design entry and simplifies the excitation equations for each machine (see 'Partitioning State Machines' Application Brief in this Handbook. As state machines grow a more effective approach is the SAM family can easily implement state machines with up to 448 states.

AB17 Rev 2.0

FEATURES

- **Useful tool for implementing state machine designs in EPLDs.**
- **Complex state machines are reduced to two or more simple state machines.**
- **Systematic approach works at the state diagram level.**

INTRODUCTION

This Application Brief describes a systematic method for partitioning large complex state machines into two or more smaller and simpler linked state machines. The technique is useful in implementing a state machine in an EPLD where a finite limitation exists on the size (number of product terms) of logic equations for the next state decoder.

In splitting a state machine, usually the resulting design requires a larger number of state registers. This may have a detrimental effect on the efficiency of implementing the design in silicon. However splitting a state machine does, usually results in smaller equations for each register and thus is useful for putting designs into fixed-width array programmable logic devices such as EPLDs.

Splitting state machines by the following technique does not always reduce the size of the equations, especially for smaller or "tightly connected" machines. The results are highly dependent on how the machine is divided, the complexity of dependent transition equations, and the state assignments. Often it will be necessary to try different divisions and state assignments prior to finding an optimum implementation. The high-level language syntax of the A+PLUS state machine conversion processor eases design iteration very easy. An optimum solution may be quickly found using this tool.

SELECT THE PARTITION

Figure 2 shows part of a state diagram describing a complex state machine. The states are annotated as S1, S2, ... S6, and each represents a unique combination of the state variables V1 to VN, forming a single logical product. (for example, S3 could represent the combination V1*/V2*/V3*V4.)

The states are connected by directional arrows marked with the symbolic labels E1 to E5. These arrows show possible transitions to and from the states and their labels represent arbitrarily complex (or simple) equations. These equations would normally be functions of inputs to the state machine allowing the machine to respond to external events.

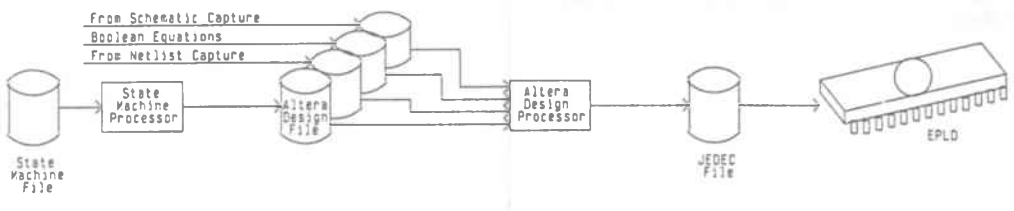
The intent is to split the state machine such that states S1, S2, S3, and all (not shown) states to the left end up as a separate machine. It is assumed that E1 to E5 represent all transitions between the two halves of the diagram.

SPLIT THE MACHINE

Draw a line that crosses all transitions between the two halves of the machine to be separated. For each half of the machine, replace the other half with a new state, called an 'idle' state. While things are happening in one half of the machine, the other half will be sitting in its idle state. When an event occurs that would have caused one of the connecting transitions of the original machine to be transversed, then the currently active half of the machine goes into its idle state, while the currently idle half leaves its idle state and enters whatever state is appropriate for that event.

Figure 3 shows the example machine after splitting. States SA and SB were inserted as the idle states for each half. Note that all transitions that would have resulted in crossing from one half to the other are routed to the idle states. Transitions leaving the idle states are dependent on both the original transition equation (E1, E2, etc.), and on

Figure 1. *Altera Programmable Logic User System (A+PLUS) combines State Machine Files with Schematic Capture files, Netlist files, and Boolean Equations prior to programming EPLDs.*



5

Figure 2. State Machine prior to partitioning.

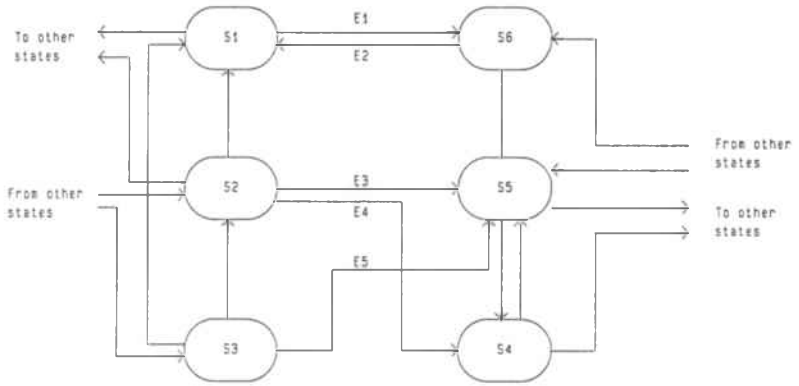
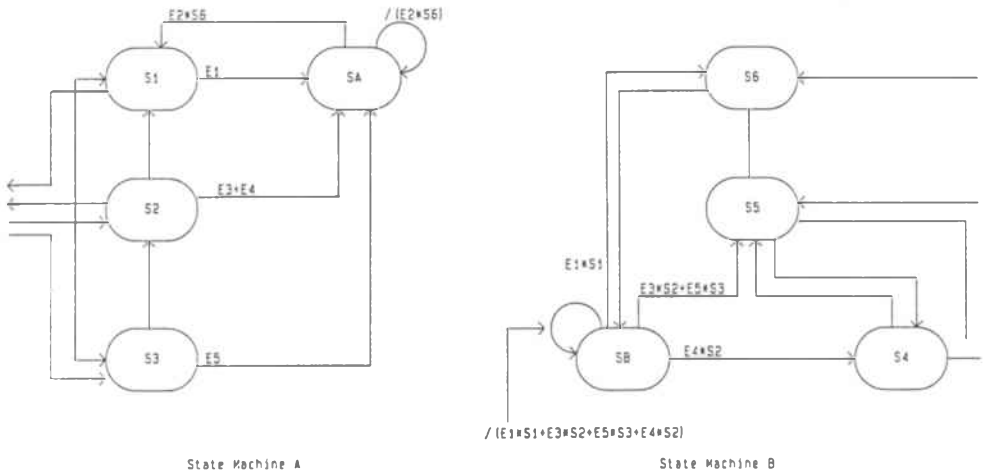


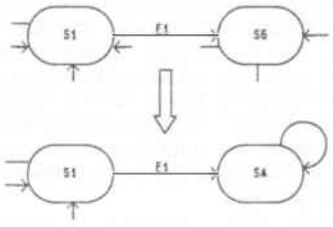
Figure 3. State Machine after partitioning.



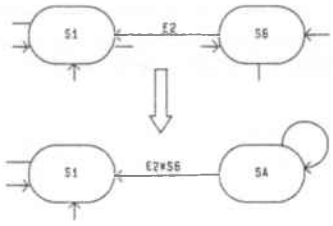
the current state of the opposing state machine half. The idle states have equations that are simply the logical inverse of all transitions leaving each idle state.

RULES FOR PARTITIONING:

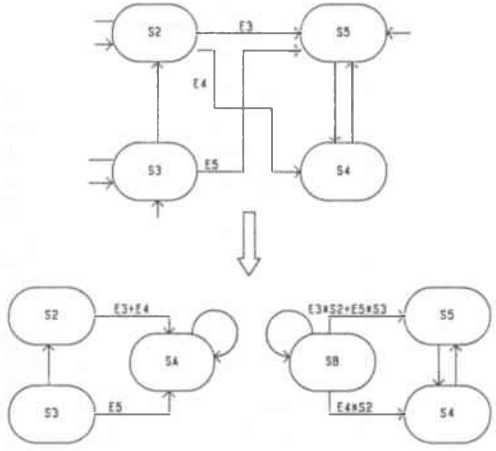
1. Transitions leading into an idle state keep the same equation as the corresponding transition from the original machine.



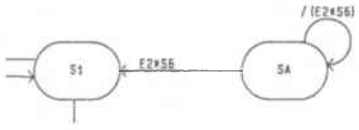
2. Transitions leading out of an idle state have the equation from the corresponding transition from the original machine, logically ANDed with the state (of the other machine half) that the transition originally came from.



3. Multiple transitions that have the same source and destination states may be replaced by a single transition with an equation that is the logical OR of the two original equations. This corresponds to two "If Then" statements with the same destination in the State Machine File (SMF), making modification of the SMF easier when splitting a design.



4. Idle states have a loopback or hold transition. The transition has as its equation the logical inverse of all equations for transitions leaving that state, logically ORed together. This happens automatically when A+PLUS does not see an ELSE transition for a given state (it assumes HOLD).



EXAMPLE DESIGN

The following state diagram (Figure 4) is from a design that converts a manchester encoded serial data stream into a standard UART-compatible serial data stream, and extracts information concerning the information being received. Details of the design are beyond the scope of this Application Brief, however the state diagram is useful in showing the technique as well as the value of state machine partitioning.

This particular design resulted in five equations with requirements ranging from 8 to 14 product terms after minimization, even with the additional benefit of selecting flip flop types (note the first equation will feed a Toggle type flip flop), see Figures 5 and 6. With these product term requirements this design would not 'fit' into any of the available EPLDs. This design is clearly a candidate for partitioning.

The state diagram in Figure 7 and the State Machine File in Figure 8 show one possible way to partition the design. The state previously marked SW has been split into idle states WL and WR for the left and right resulting state machines. This was possible because the state SW was a hold state in the original design (the conditions for holding are ORed together). The equivalent condition of the original machine is when both of the partitioned machine halves are in their idle states (WL and WR) at the same time.

Figure 4. *Manchester Sync Detection original state diagram reaches states DAT (COM) if a valid DATA (COMMAND) manchester serial word is detected.*

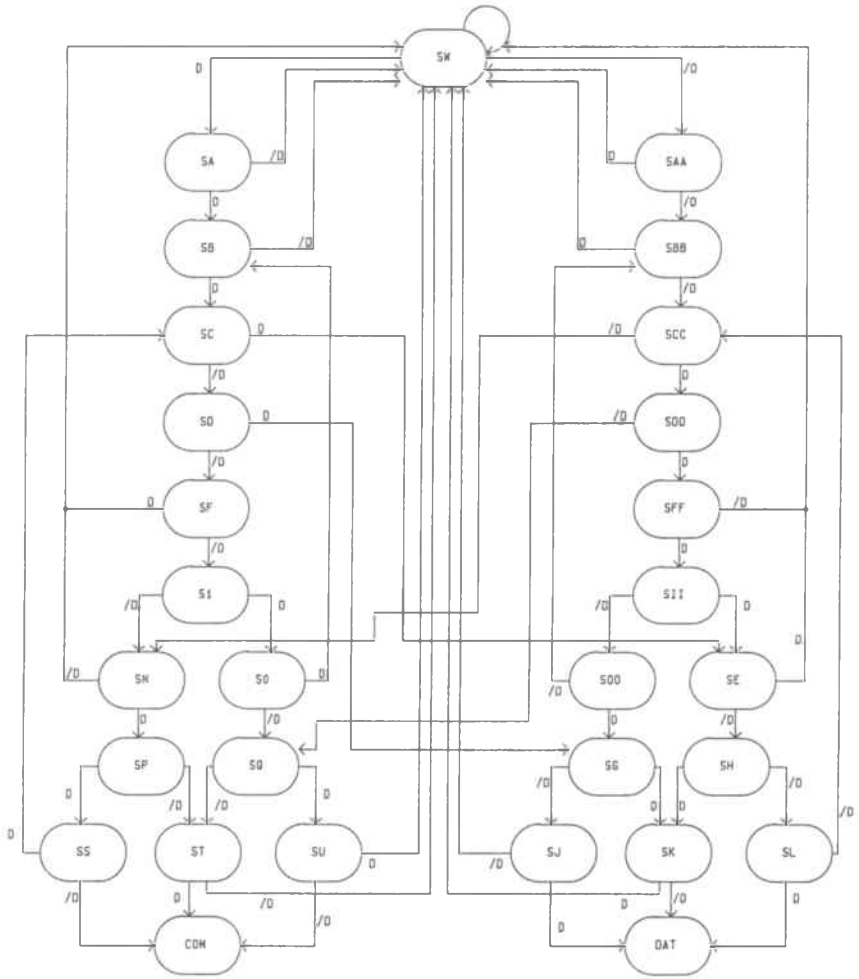


Figure 5. *State Machine File for the original manchester design, ready to be processed by A×PLUS.*

```
PART: KP900J

INPUTS: D, clk

OUTPUTS:  % no outputs %

MACHINE: hdl6530_state_machine

    CLOCK: clk

            % state assignments %
      STATES:  [ SV4 SV3 SV2 SV1 SV0  ]

         SW    [  0   0   0   0   0   ]
         SA    [  0   1   0   0   0   ]
         SH    [  0   1   1   0   0   ]
         SC    [  1   1   1   0   0   ]
         SD    [  1   0   1   0   0   ]
         SF    [  0   0   1   0   0   ]
         SI    [  0   0   1   1   0   ]
         SN    [  1   0   1   1   0   ]
         SO    [  0   1   1   1   0   ]
         SP    [  1   0   0   1   0   ]
         SQ    [  1   1   1   1   0   ]
         SS    [  0   0   0   1   0   ]
         ST    [  1   1   0   1   0   ]
         SU    [  0   1   0   1   0   ]
         COM   [  0   0   0   1   1   ]
         SAA   [  1   0   0   0   0   ]
         SDD   [  1   0   0   0   1   ]
         SCC   [  0   0   0   0   1   ]
         SDD   [  0   1   0   0   1   ]
         SFF   [  1   1   0   0   1   ]
         SII   [  1   1   1   0   1   ]
         SOO   [  1   0   1   0   1   ]
         SR    [  0   1   1   0   1   ]
         SG    [  0   0   1   0   1   ]
         SH    [  0   1   1   1   1   ]
         SJ    [  1   0   1   1   1   ]
         SK    [  0   0   1   1   1   ]
         SL    [  1   1   1   1   1   ]
         DAT   [  1   0   0   1   1   ]

            % transitions %
         SW:  IF  D   THEN    SA   % ELSE %    SAA
         SCC: IF  D   THEN    SDD  % ELSE %    SN
         SS:  IF  D   THEN    SC   % ELSE %    COM
         COM: IF  D   THEN    COM  % ELSE %    COM
         SF:  IF  D   THEN    SW   % ELSE %    SI
         SG:  IF  D   THEN    SK   % ELSE %    SJ
         SI:  IF  D   THEN    SO   % ELSE %    SN
         SK:  IF  D   THEN    SW   % ELSE %    DAT
         SA:  IF  D   THEN    SH   % ELSE %    SW
         SDD: IF  D   THEN    SFF  % ELSE %    SQ
         SU:  IF  D   THEN    SW   % ELSE %    COM
         SR:  IF  D   THEN    SC   % ELSE %    SW
         SR:  IF  D   THEN    SW   % ELSE %    SH
         SO:  IF  D   THEN    SR   % ELSE %    SQ
         SH:  IF  D   THEN    SK   % ELSE %    SL
         SAA: IF  D   THEN    SW   % ELSE %    SDD
         SDD: IF  D   THEN    SW   % ELSE %    SCC
         SP:  IF  D   THEN    SS   % ELSE %    ST
         DAT: IF  D   THEN    DAT  % ELSE %    DAT
         SD:  IF  D   THEN    SG   % ELSE %    SF
         SOO: IF  D   THEN    SG   % ELSE %    SDD
         SN:  IF  D   THEN    SP   % ELSE %    SW
         SJ:  IF  D   THEN    DAT  % ELSE %    SW
         SFF: IF  D   THEN    SII  % ELSE %    SW
         ST:  IF  D   THEN    COM  % ELSE %    SW
         SC:  IF  D   THEN    SR   % ELSE %    SD
         SII: IF  D   THEN    SR   % ELSE %    SOO
         SQ:  IF  D   THEN    SH   % ELSE %    ST
         SL:  IF  D   THEN    DAT  % ELSE %    SCC
END$
```

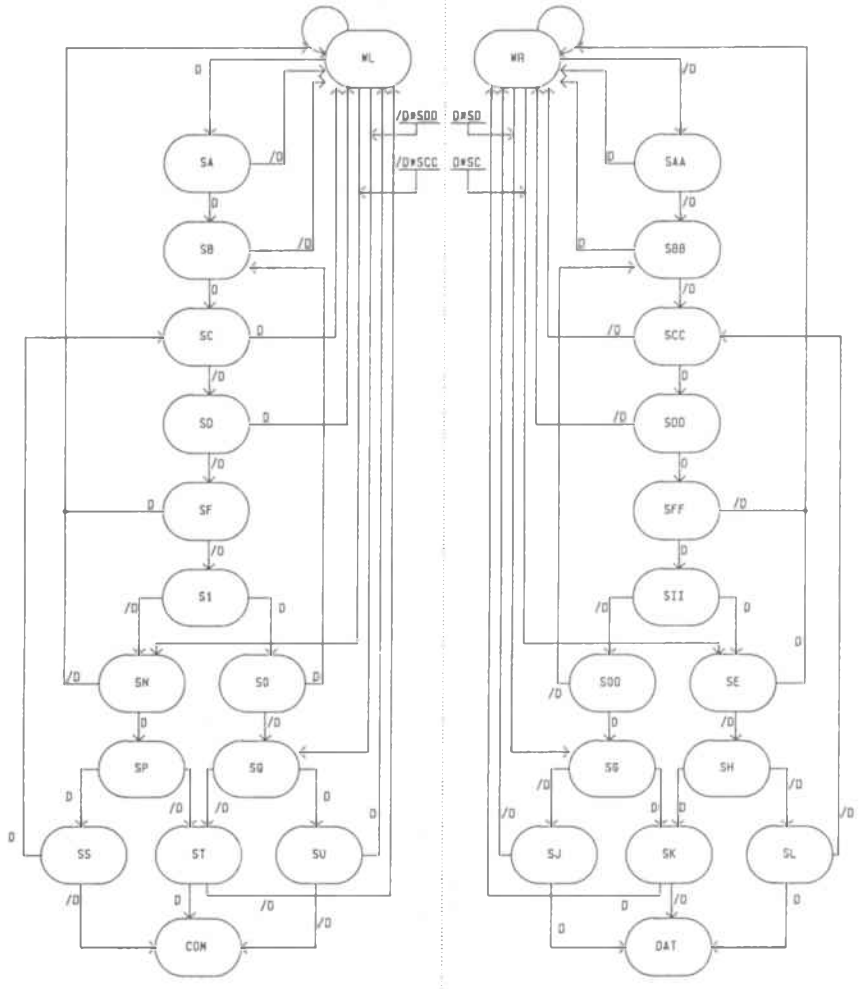
Figure 6. *Resulting minimized Boolean equations from the original manchester design. Up to 14 product terms are required in one of the EPLD macrocells for this design to fit.*

```
OPTIONS: TURBO = ON, SECURITY = OFF
PART:
        KP900J
INPUTS:
        D, clk
OUTPUTS:
        SV4, SV3, SV2, SV1, SV0
NETWORK:
        clk = INP(clk)
        D = INP(D)
        SV4 = NORF(SV4.d, clk, GND, GND)
        SV3 = NOTF(SV3.t, clk, GND, GND)
        SV2 = NOTF(SV2.t, clk, GND, GND)
        SV1 = NOTF(SV1.t, clk, GND, GND)
        SV0 = NOTF(SV0.t, clk, GND, GND)
EQUATIONS:
        SV0.t = SV4' * SV2' * SV1 * SV0' * D'
              + SV4' * SV2' * SV1' * SV0 * D'
              + SV3 * SV2' * SV1' * SV0 * D'
              + SV4 * SV2 * SV1' * SV0' * D
              + SV4 * SV3' * SV2' * SV1' * SV0' * D'
              + SV4 * SV3' * SV2' * SV1' * SV0 * D
              + SV4' * SV3' * SV2 * SV1 * SV0 * D
              + SV4' * SV3 * SV2 * SV1' * SV0 * D
              + SV4 * SV3' * SV2 * SV1 * SV0 * D'
              + SV4 * SV3 * SV2' * SV1 * SV0' * D;

        SV1.t = SV4' * SV1' * SV0 * D'
              + SV4' * SV3' * SV2 * SV1' * D'
              + SV4' * SV2' * SV1 * SV0' * D
              + SV4 * SV3' * SV2 * SV1 * D'
              + SV4' * SV3' * SV2 * SV0 * D
              + SV4' * SV3 * SV1 * SV0' * D
              + SV4 * SV2 * SV1 * SV0 * D'
              + SV4 * SV3 * SV2' * SV1 * SV0' * D';

        SV2.t = SV4 * SV2 * SV1
              + SV3' * SV2 * SV1 * SV0
              + SV4' * SV2' * SV1' * SV0 * D'
              + SV4 * SV3' * SV2 * SV0 * D'
              + SV4' * SV3 * SV2' * SV1' * SV0' * D
              + SV4' * SV3' * SV2 * SV1 * SV0' * D'
              + SV4' * SV3 * SV2' * SV1 * SV0' * D
              + SV4 * SV3 * SV2' * SV1 * SV0 * D
              + SV4' * SV3 * SV2 * SV1 * SV0 * D;

        SV3.t = SV3 * SV2' * SV1 * SV0'
              + SV4' * SV3 * SV2' * SV0' * D'
              + SV4' * SV3' * SV2' * SV1' * D
              + SV4' * SV3' * SV1 * SV0' * D
              + SV3 * SV2 * SV1' * SV0' * D'
              + SV4 * SV2' * SV1 * SV0' * D'
              + SV4 * SV3 * SV1' * SV0 * D'
              + SV4' * SV3 * SV2 * SV0 * D
              + SV4 * SV3 * SV2 * SV1 * SV0;

        SV4.d = SV4' * SV2 * SV1 * D'
              + SV3' * SV2' * SV1' * SV0' * D'
              + SV4 * SV3' * SV2' * SV0' * D'
              + SV4' * SV2' * SV1' * SV0 * D'
              + SV3' * SV2 * SV1' * SV0 * D'
              + SV4 * SV3 * SV2 * SV0' * D'
              + SV3 * SV2' * SV1' * SV0 * D
              + SV4 * SV3 * SV2 * SV1' * D'
              + SV4 * SV3' * SV2' * SV1 * SV0
              + SV4 * SV3' * SV2 * SV1 * D
              + SV4 * SV2 * SV1 * SV0 * D
              + SV4' * SV3' * SV2' * SV1 * SV0' * D
              + SV4' * SV3 * SV2 * SV1 * SV0' * D;

END$
```

CONCLUSION

State Machine designs often result in a small number of large equations. There is a finite limit on the size (number of product terms) of logic equations that may be implemented in programmable logic devices. Thus, some form of partitioning is often required to fit a design. Use of the described technique results in more equations, but they are often smaller. Not all state machines will benefit from this form of partitioning, and several tries may be required before a design does benefit. However, partitioning should prove useful in many cases.

Figure 7. *Manchester design state diagram is partitioned into two less complex linked state machines.*



Summary of partitioning the Manchester Sync Detection design.

	Original Machine	Partitioned Machine
Number of states	29	15 + 15 = 30
State variables	5	4 + 4 = 8
Total Number of ProductTerms	56	54
Maximum product terms per state variable	14	8

Figure 8. *State Machine File for the partitioned design. Note there are two 'MACHINE' sections in the file, one for each half of the manchester design.*

```
PART: EP900J

INPUTS: D, clk

OUTPUTS:  % no outputs %

MACHINE: hdl6530_state_machine

    CLOCK: clk

          % state assignments %
      STATES:  [ SVL3 SVL2 SVL1 SVL0  ]

         SWL   [  0    0    0    0    ]

         SAA   [  0    1    0    0    ]
         SB    [  1    1    1    1    ]
         SC    [  1    0    1    1    ]
         SD    [  1    0    0    1    ]
         SF    [  1    0    0    0    ]
         SI    [  0    0    0    1    ]
         SN    [  0    1    0    1    ]
         SO    [  1    1    0    1    ]
         SP    [  0    0    1    1    ]
         SQ    [  0    1    1    1    ]
         SR    [  1    1    1    0    ]
         ST    [  0    1    1    0    ]
         SU    [  0    0    1    0    ]
         COM   [  1    0    1    0    ]

          % transitions %
         SWL:   IF  D # SWR      THEN    SA
                IF /D # SCC      THEN    SN
                IF /D # SDD      THEN    SQ
            % ELSE %                     SWL

         SS:    IF D    THEN     SC  % ELSE %    COM
         COM:   IF D    THEN     COM % ELSE %    COM
         SF:    IF D    THEN     SWL % ELSE %    SI
         SI:    IF D    THEN     SO  % ELSE %    SN
         SA:    IF D    THEN     SD  % ELSE %    SWL
         SU:    IF D    THEN     SWL % ELSE %    COM
         SR:    IF D    THEN     SC  % ELSE %    SWL
         SQ:    IF D    THEN     SR  % ELSE %    SQ
         SP:    IF D    THEN     SS  % ELSE %    ST
         SO:    IF D    THEN     SWL % ELSE %    SF
         SN:    IF D    THEN     SP  % ELSE %    SWL
         ST:    IF D    THEN     COM % ELSE %    SWL
         SC:    IF D    THEN     SWL % ELSE %    SR
         SQ:    IF D    THEN     SU  % ELSE %    ST

MACHINE:
     hdl6530_right_machine

    CLOCK: clk

          % state assignments %
      STATES:  | SVR3 SVR2 SVR1 SVR0  ]

         SWR   [  0    0    0    0    ]

         SAA   [  0    1    0    0    ]
         SBB   [  1    1    1    1    ]
         SCC   [  1    0    1    1    ]
         SDD   [  1    0    0    1    ]
         SFF   [  1    0    0    0    ]
         SII   [  0    0    0    1    ]
         SR    [  0    1    0    1    ]
         SOO   [  1    1    0    1    ]
         SH    [  0    0    1    1    ]
         SG    [  0    1    1    1    ]
         SL    [  1    1    1    0    ]
         SK    [  0    1    1    0    ]
         SJ    [  0    0    1    0    ]
         DAT   [  1    0    1    0    ]

          % transitions %
         SWR:   IF  D # SWL      THEN    SAA
                IF /D # SC       THEN    SE
                IF /D # SD       THEN    SG
            % ELSE %                     SWR

         SCC:   IF D    THEN     SDD % ELSE %    SWR
         SG:    IF D    THEN     SK  % ELSE %    SJ
         SK:    IF D    THEN     SWR % ELSE %    DAT
         SDD:   IF D    THEN     SFF % ELSE %    SWR
         SR:    IF D    THEN     SWR % ELSE %    SH
         SH:    IF D    THEN     SK  % ELSE %    SL
         SAA:   IF D    THEN     SWR % ELSE %    SBB
         SRR:   IF D    THEN     SWR % ELSE %    SCC
         DAT:   IF D    THEN     DAT % ELSE %    DAT
         SOO:   IF D    THEN     SG  % ELSE %    SBB
         SJ:    IF D    THEN     DAT % ELSE %    SWR
         SFF:   IF D    THEN     SII % ELSE %    SWR
         SII:   IF D    THEN     SR  % ELSE %    SOO
         SL:    IF D    THEN     DAT % ELSE %    SCC

END$
```

Figure 9. *Minimized Boolean equations from the partitioned manchester design shows 8 macrocells, but none require more than 8 product terms.*

```
OPTIONS: TURBO = ON, SECURITY = OFF
PART:
        EP900J
INPUTS:
        D, clk
OUTPUTS:
        SVL3, SVL2, SVL1, SVL0, SVR3, SVR2, SVR1, SVR0
NETWORK:
        clk = INP(clk)
        D = INP(D)
        SVL3 = MOTF(SVL3.t, clk, GND, GND)
        SVL2 = MOHF(SVL2.d, clk, GND, GND)
        SVL1 = MOTF(SVL1.t, clk, GND, GND)
        SVL0 = MOHF(SVL0.d, clk, GND, GND)
        SVR3 = MOTF(SVR3.t, clk, GND, GND)
        SVR2 = MOHF(SVR2.d, clk, GND, GND)
        SVR1 = MOTF(SVR1.t, clk, GND, GND)
        SVR0 = MOHF(SVR0.d, clk, GND, GND)
EQUATIONS:
        SVR0.d = SVR3' * SVR2 * SVR1' * D'
               + SVR3' * SVR2' * SVR1' * SVR0
               + SVR3 * SVR2 * SVR1 * D'
               + SVR3 * SVR2 * SVR1' * SVR0
               + SVR3 * SVR2' * SVR1 * SVR0 * D
               + SVR3' * SVR1' * D' * SVL3 * SVL2' * SVL0;

        SVR1.t = SVR3' * SVR2 * SVR1' * D'
               + SVR2 * SVR1' * SVR0 * D'
               + SVR3 * SVR2' * SVR1 * SVR0
               + SVR3 * SVR2 * SVR0 * D
               + SVR3' * SVR2' * SVR1 * SVR0' * D'
               + SVR3' * SVR2 * SVR1 * SVR0' * D
               + SVR3' * SVR2' * SVR0' * D' * SVL3 * SVL2'
               * SVL1' * SVL0;

        SVR2.d = SVR3' * SVR2' * SVR0
               + SVR3 * SVR2 * SVR1' * SVR0
               + SVR3' * SVR1 * SVR0 * D
               + SVR3' * SVR2 * SVR1' * SVR0' * D'
               + SVR3' * SVR2' * SVR1' * D' * SVL3 * SVL2'
               * SVL0
               + SVR3' * SVR2' * SVR1' * D * SVL3' * SVL2'
               * SVL1' * SVL0';

        SVR3.t = SVR2' * SVR0 * D'
               + SVR3' * SVR2 * SVR0' * D'
               + SVR3 * SVR2' * SVR1' * SVR0'
               + SVR3 * SVR2 * SVR0 * D
               + SVR3' * SVR2' * SVR1 * SVR0' * D;

        SVL0.d = SVL3' * SVL2' * SVL1' * SVL0
               + SVL3' * SVL2 * SVL1' * D
               + SVL3 * SVL2 * SVL1' * SVL0
               + SVL3' * SVL2' * SVL1 * D
               + SVL3 * SVL2' * SVL1' * SVL0' * D'
               + SVL3' * SVL2' * SVL1 * SVL0 * D'
               + SVL2' * SVL1' * SVL0' * D' * SVR3 * SVR2'
               * SVR0;

        SVL1.t = SVL3' * SVL2 * SVL1' * D
               + SVL3 * SVL2' * SVL1 * SVL0
               + SVL3 * SVL2 * SVL1' * SVL0
               + SVL3' * SVL2 * SVL0 * D'
               + SVL3' * SVL2 * SVL1' * SVL0' * D
               + SVL3' * SVL2' * SVL1 * SVL0' * D'
               + SVL3' * SVL2' * SVL1' * SVL0 * D' * SVR3
               * SVR2' * SVR1' *
                 SVR0;

        SVL2.d = SVL3' * SVL2' * SVL0
               + SVL3' * SVL1 * SVL0 * D'
               + SVL3 * SVL2 * SVL1' * SVL0
               + SVL3' * SVL2 * SVL1' * SVL0' * D
               + SVL3' * SVL2' * SVL1' * D' * SVR3 * SVR2'
               * SVR0
               + SVL3' * SVL2' * SVL1' * D * SVR3' * SVR2'
               * SVR1' * SVR0';

        SVL3.t = SVL2' * SVL0 * D
               + SVL3 * SVL2' * SVL1' * SVL0'
               + SVL3' * SVL2 * SVL0' * D
               + SVL3 * SVL2 * SVL0 * D'
               + SVL3' * SVL2' * SVL1 * SVL0' * D';

END$
```

INTRODUCTION

Mealy and Moore machines are two categories of state machines with different output characteristics. Altera's SAM (Stand-Alone Microsequencer) family of function-specific programmable logic devices can directly implement Moore type machines. But can they implement Mealy as well? While unable to directly implement Mealy machines, one can transform a Mealy machine into one of Moore type using a simple technique. This Applications Brief will describe the transformation technique, and provide an illustrative example of the transformation of an EP600-based Mealy machine into a EPS444-based Moore machine.

MOORE AND MEALY CIRCUIT OVERVIEW

Digital logic design divides all sequential circuits into two classes: Moore and Mealy state machines. Moore machines have outputs which are a function only of the machine's current state, a value stored in the machine's state register. Mealy machines have outputs which are a function of the machine's current state and the state machine's input values. Figure 1 highlights the distinction between Moore and Mealy machines.

Figure 2 shows state machine behavior described using Next-State Tables and State Diagrams (aka. Bubble Diagrams). Each state is given a unique mnemonic, which is mapped to a state's unique state register assignment (the state assignment). In Figure 2, A, B and C are mnemonics for various states, Q0 is the state register, and the state assignment table describes the relationship between the value of Q0 and the various states. The Moore machine also uses OUT1, a registered output, as a state register. The Next-State table considers each state under all input combinations, and notes the proper next state for these conditions. The Mealy machine also indicates the output value during these transitions, shown as the value beneath the slash. Moore machine outputs are not associated with output transitions because the output is coded into the state register assignment, and hence only changes as the state variables do.

State diagrams graphically describe state machine behavior. Each bubble represents a single state, and transitions between states are indicated by arrows linking the different bubbles. Transitions between the bubbles occur only on the rising edge of the state machine's clock. Input combinations required for transitions between states are noted by labeling the arrows between states. The Mealy machine also indicates the output values during transitions, shown below the corresponding input combination value.

Figure 1 Types of Synchronous State Machines

Moore machine has fixed outputs for each state. Mealy machine allows asynchronous outputs in response to inputs.

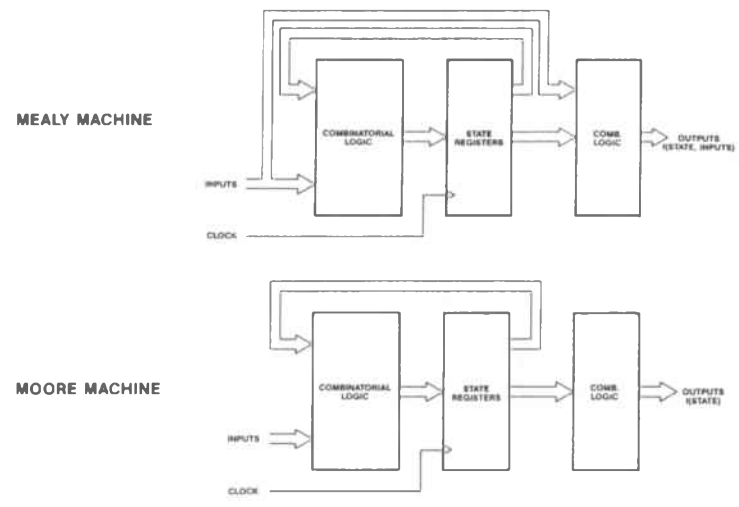
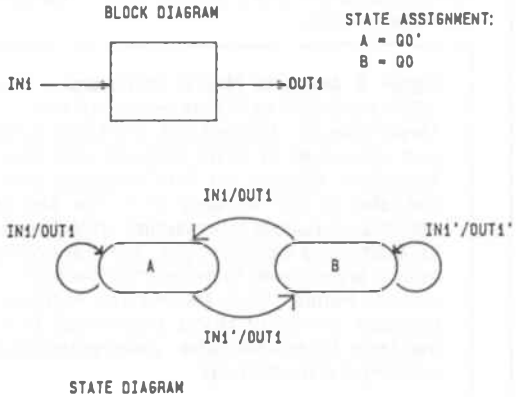


Figure 2. Describing State Machine Behavior

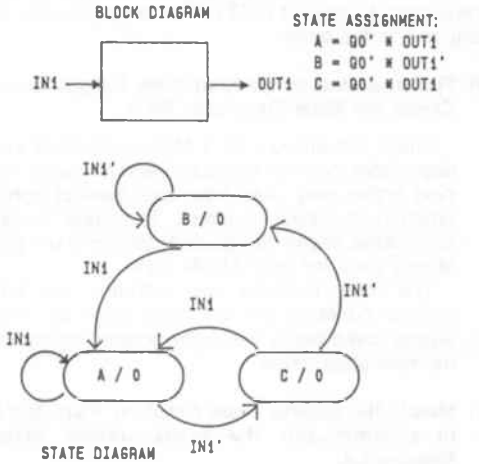
(a) The behavior of a two state Mealy machine with input IN1 and output OUT1 is described using a state diagram and a next state table. Output values are tied to transitions, not to states.



CURRENT STATE	NEXT STATE / OUTPUT	
	IN1 = 0	IN1 = 1
A	B / OUT1	A / OUT1
B	B / OUT1'	A / OUT1

NEXT STATE TABLE

(b) The behavior of an analagous Moore machine with input IN1 and output OUT1 is also described using a state diagram and a next state table. Output values are tied to states in a Moore machine. The machine in (b) emulates the behavior of the machine in (a).



CURRENT STATE	NEXT STATE	
	IN1 = 0	IN1= 1
A	C	A
B	B	A
C	B	A

NEXT STATE TABLE

CONVERTING MEALY MACHINES TO MOORE MACHINES

The following procedure details how to convert the Mealy machine in Figure 2a to the Moore machine in Figure 2b:

1) **Create a state diagram of the Mealy machine to be converted.**
2) **Build the state table for the Mealy machine based on the state diagram.**
3) **List all output combinations for each state.**

The State Diagram and State Table are shown in Figure 2a.

The Next State Output Table (Figure 3) describes the output transitions for each state. State mnemonics are listed along the left margin for each state. These mnemonics define the rows of the table. If a Mealy state has input transitions with different output combinations, then a unique

Figure 3. Next State Output Table

The Next State Output Table is used to identify Mealy states which need to be modified to remove output dependence on state machine inputs. The next state appears on the left side of the table, and the outputs which can occur during transitions into that next state appear on the right.

NEXT STATE	OUTPUT COMBINATIONS
A	OUT1
B	OUT1'
B	OUT1

Moore state is required for each output combination. For example, Mealy state B has output combinations of OUT1 and OUT1' for it's input transitions. The table can be filled in by using either the State diagram or the State Table.

An entry is made for each transition into the Next State column. Duplicate output combinations need to be listed only once. From Figure 2, state A has two transitions leading into it, both which have output value of OUT1, thus only one row is required in the table.

4) Flatten entries in the Next State Output Table. Create the State Transform Table.

Since the outputs of a Moore machine are dependent only on the state register value, all next states may only have one distinct combination of output variables. The State Transform Table, shown in Figure 4, assigns a unique Moore state for each Mealy state.

The table contains two columns: the left column contains the old state name and the output conditions, the right column contains the new state name.

5) Modify the original State Graph or State Table to accommodate the newly created State Mnemonics.

A Moore State Diagram, or State Table, can be constructed using the original Mealy State Diagram and the State Transform Table. Though the entire State Diagram can be constructed at once, it is generally helpful to graphically model each of the new states listed in the State Transform Table.

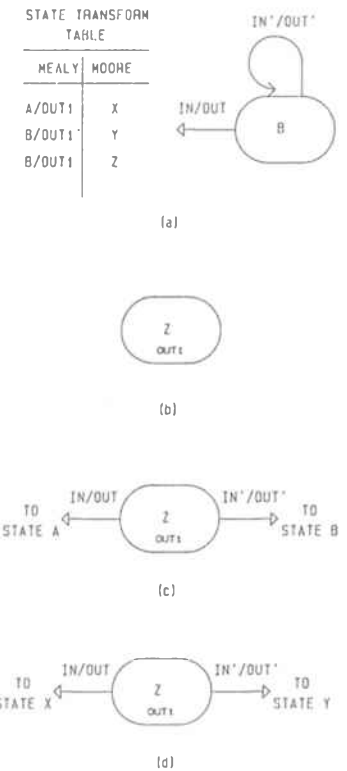
To graphically reconstruct a single Moore state (Figure 5 (a)), a single bubble is drawn with the appropriate next state transitions. Draw a bubble and label it with the Moore state name from the State Transform Table (Figure 5 (b)). Find the next state transitions by replicating and modifying those for the analagous Mealy state. Use the State Transform Table to obtain the Mealy state name, and copy all next state transitions from the Mealy State diagram to the Moore state bubble. Make sure to record the next state name and output values (Figure 5 (c)). Complete the Moore state transform by replacing the Mealy next states with the appropriate Moore state, found in the State Transform Table (Figure 5 (d)). Repeat the transform process for every Moore state in the State Transform Table.

Figure 4. State Transform Table

The State Transform Table lists the correspondence between Mealy states in the original machine, and Moore states in the converted machine.

MEALY	MOORE
A/OUT1	X
B/OUT1'	Y
B/OUT1	Z

Figure 5. Mealy to Moore Conversion—*The following describes the conversion of a single Mealy state to a Moore state. The Mealy state B is described by State Diagram and State Transform Table in (a). The transform table indicates B will become Z in the Moore machine. A bubble representing state Z, with its outputs, is drawn in (b). The transitions from C are mapped from the state diagram of Z in (c). References to Mealy states in (c) are replaced with their Moore equivalents from the State Transform Table, yielding the fully converted state B in (d).*



TRANSFORM TECHNIQUE EXAMPLE

The following example illustrates the transformation of the Mealy state diagram in Figure 6. The state machine has inputs CLR, SKIP, and HOP, and outputs ODD and EVEN. Figure 7 shows equivalent next State Table for this diagram.

1) Construct the Next State Output Table

List each state mnemonic down the left hand side of the table. Begin filling the table by identifying all transitions in the state diagram that terminate in the RESET state. There are only two such transitions. The first occurs when CLR is high (From Any State) with outputs (00). Record this output value on the first line of the table. The second occurs when CLR is high and RESET loops back on itself. Since the outputs are also (00) during this transition, no additional entry is made to the table. Repeat this for the remaining states. In this case all entries to the table result in only one output combination, except for state D.

There are two transition paths into state D: one with outputs (01), the other with outputs (10). There are thus two entries in the table for state D. Figure 8 shows the completed table.

Figure 6. Mealy State Diagram

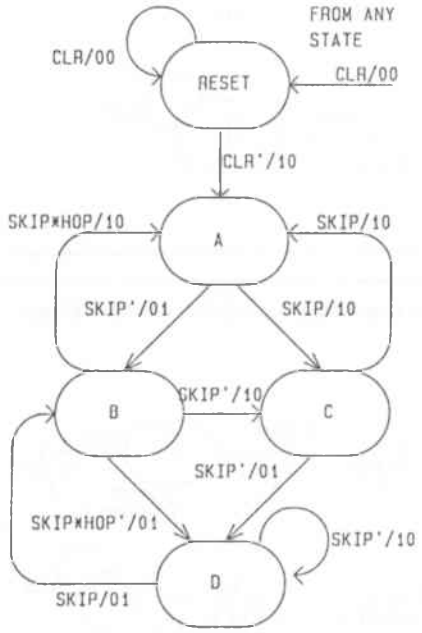


Figure 7. Next State Table for Mealy State Machine

CURRENT STATE	NEXT STATE / (ODD, EVEN) CLR, SKIP, HOP							
	000	001	010	011	100	101	110	111
RESET (R)	A/10	A/10	A/10	A/10	R/00	R/00	R/00	R/00
A	B/01	B/01	C/10	C/10	R/00	R/00	R/00	R/00
B	C/10	C/10	D/01	A/10	R/00	R/00	R/00	R/00
C	D/01	D/01	A/10	A/10	R/00	R/00	R/00	R/00
D	D/10	D/10	B/01	B/01	R/00	R/00	R/00	R/00

2) Create the State Transform Table.

The table is flattened by removing State D, which has multiple output combinations, and replacing it with states D0 and D1. D0 and D1 each have only a single output combination. The conversion information is stored in the state transform table. The complete State Transform table is shown in Figure 9.

Figure 8. Next State Output Table for Mealy Machine

NEXT STATE	OUTPUT COMBINATIONS
RESET	00
A	10
B	01
C	10
D	01
D	10

5

Figure 9. State Transform Table

State Transform table maps Mealy states to Moore states.

MEALY STATE	MOORE STATE
RESET| 00	RESET
A| 10	A
B| 01	B
C| 10	C
D| 01	D0
D| 10	D1

3) Graphically transform Mealy States to Moore States

The individual Moore state's connectivity must be established to construct the final Moore state diagram. Consider state D0. By examining the State Transition Table (Figure 9), the equivalent Mealy state is D with output (01). The transitions for Mealy state D are shown in Figure 10 (a). Figure 10 (b) replicates these transitions on Moore state D0, including the next state outputs. Each of the next state/output combinations in Figure 10 (b) are transformed into their Moore equivalents using the State Transform table. The result is shown in Figure 10 (c). The transform for Moore state D1 is shown in Figure 10 (d).

Transformations for all states are conducted in the same fashion. The resulting diagrams are shown in Figure 11.

Figure 10. Conversion of Mealy State D—*Mealy state D, shown in (a), is converted to Moore states D0 and D1. Conversion to D0 starts with the modeling of state D transition behavior (b), and the substitution of Moore states for transition values (c). An analagous conversion is performed for D1 (d).*

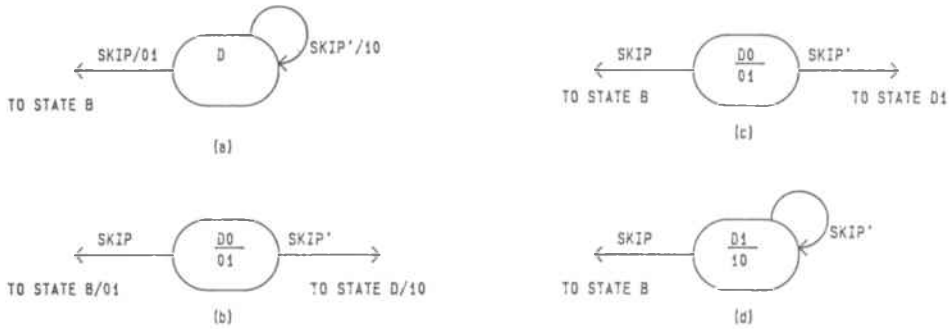
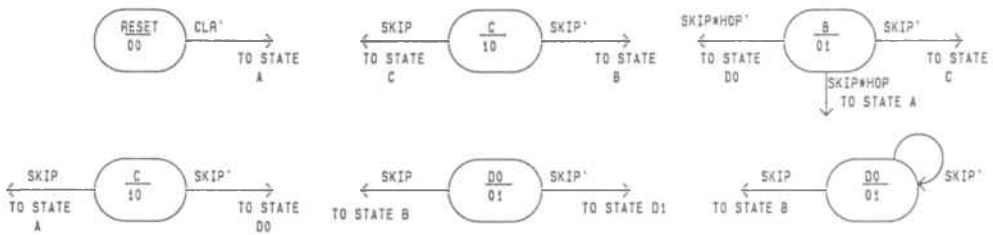


Figure 11. Converted Moore States—*All Mealy states are graphically converted to their Moore equivalents.*

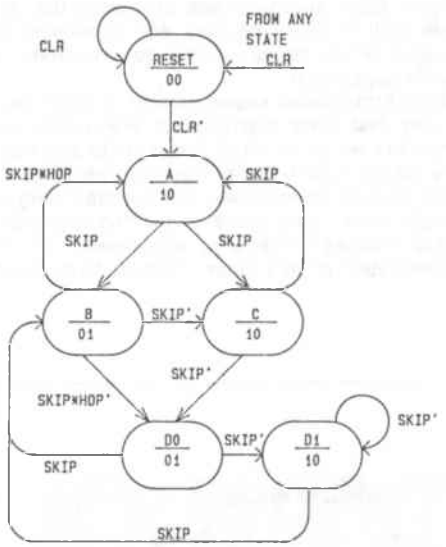


4) Construct the Final Moore State Diagram

The Moore state diagram is constructed by "cutting and pasting" the Moore states (Figure 11) over the Mealy state diagram (Figure 6). Start with the RESET state, which requires a connection to state A, which in turn requires connection to states B and C. Figure 12 shows the final Moore state diagram which results by satisfying all such connections.

Figure 12. Moore State Machine

The final converted Moore machine.



CONVERSION IN ALTERA STATE MACHINE FILE FORMAT

Figure 13 is an Altera State Machine Format implementation of the Mealy machine of Figure 6. Each IF...THEN statement represents a conditional branch from the current Mealy state to the next state. On power up, the Mealy machine resides in state RESET. As shown, the Mealy machine of Figure 6 can be represented by the Moore machine of Figure 12. The Moore machine is described using the state machine format in Figure 14. The code in Figure 14 can be processed by Altera's SAMPLUS development software, which creates a JEDEC file which can be used to program SAM devices.

Figure 13. EP600 Mealy Design

```
Altera
Mealy State Machine Design
PART: EP600
INPUTS: CLK, SKIP, HOP, CLR
OUTPUTS: S2, S1, S0, EVENp, ODDp
NETWORK:

EVENp = CONF(EVEN,)
ODDp = CONF(ODD,)

MACHINE: MEALY
CLOCK: CLK
CLEAR: CLR
STATES: [S2 S1 S0]
  RESET  [0 0 0]
      A  [0 0 1]
      B  [0 1 0]
      C  [0 1 1]
      D  [1 0 0]


RESET:
        A

        OUTPUTS: ODD

A:  IF SKIP THEN C
    B
    OUTPUTS: IF SKIP THEN ODD
             IF /SKIP THEN EVEN
B:
    IF SKIP * /HOP THEN D
    IF SKIP * HOP THEN A
    C
    OUTPUTS: IF SKIP * /HOP THEN EVEN
             IF /SKIP * (SKIP * HOP) THEN ODD
C:
    IF SKIP THEN A
    D
    OUTPUTS: IF SKIP THEN ODD
             IF /SKIP THEN EVEN
D:
    IF SKIP THEN B
    D
    OUTPUTS: IF SKIP THEN EVEN
             IF /SKIP THEN ODD

END;
```

Figure 14. EPS444 Moore Design

```
Altera
Moore State Machine Design
PART: EP3444
INPUTS: CLK, SKIP, HOP, CLR
OUTPUTS: S2, S1, S0, EVEN, ODD

MACHINE: MOORE
CLOCK: CLK
STATES: [S2 S1 S0 ODD EVEN]
  RESET  [0 0 0 0 0]
      A  [0 0 1 1 0]
      B  [0 1 0 0 1]
      C  [0 1 1 1 0]
     D0  [1 0 0 0 1]
     D1  [1 0 1 1 0]


RESET:
        IF CLR THEN RESET
        A

A:  IF CLR THEN RESET
    IF SKIP THEN C
    B

B:
    IF CLR THEN RESET
    IF SKIP * /HOP THEN D0
    IF SKIP * HOP THEN A
    C

C:
    IF CLR THEN RESET
    IF SKIP THEN A
    D0

D0:
    IF CLR THEN RESET
    IF SKIP THEN B
    D1

D1:
    IF CLR THEN RESET
    IF SKIP THEN B
    D1

END;
```

INTRODUCTION

This Application Note is intended to acquaint the user with ASMILE (Altera State Machine Input Language) state machine language syntax, as used for entering designs into the SAM family of devices. Basic functionality and syntax is reviewed as well as its use of SAM internal resources. An application utilizing ASMILE input in the form of a 68020 Microprocessor Bus Arbiter is presented. This Application Note provides illustrations of all basic concepts needed to execute a SAM design with ASMILE. For information on microassembler-based entry of SAM designs, please refer to "High End SAM Applications Using Microassembler Design Entry," later in this Handbook.

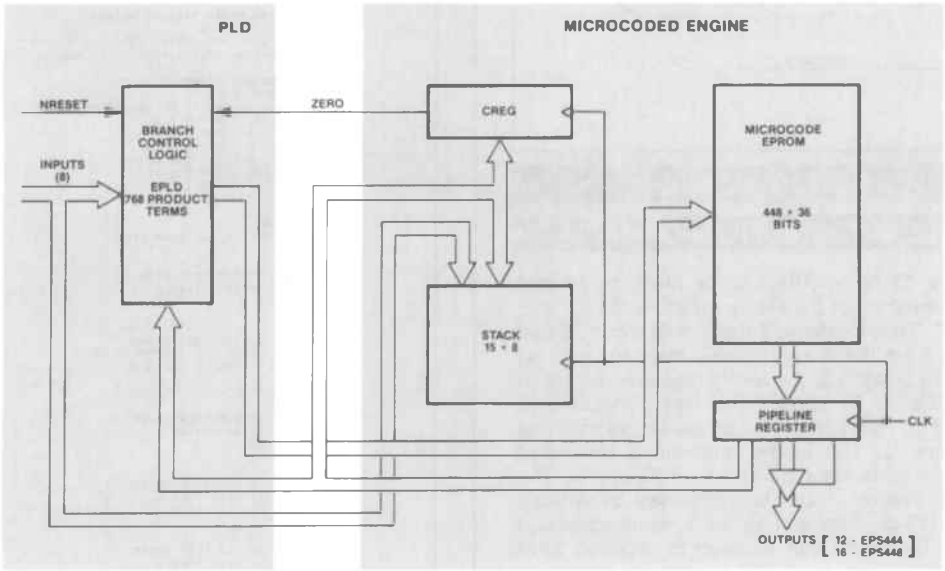
The reader is referred to Altera's SAM EPS444/448 Data Sheet for details concerning device architecture and performance. A general knowledge of SAM device architecture is assumed as background for this Application Note.

THE SAM SOLUTION

Altera's SAM (Stand-Alone Microsequencer) User-Configurable Sequencer Architecture provides a solution for high-performance control functions found in typical digital systems designed today. There have previously been two main approaches used in the design of high performance state machine/control functions in digital systems: Logic Array-based sequencers, and microcoded designs. Each approach has presented the designer with a set of benefits and drawbacks to consider when deciding how to implement a specific application.

Logic Array-based sequencers have been used for very fast state machines of low-to-medium complexity which required few outputs and relatively simple state flows or machine "algorithms". Ability to perform multi-way control branching in a single clock cycle is a plus for this approach. Devices such as conventional registered PLDs are representative of this class. Product term count

Figure 1. SAM Block Diagram



limitations, resulting in the inability to generate complex output waveforms or state transitions, limits the utility of this approach when addressing larger control problems.

Microcoded approaches have been used for the implementation of complex control functions, requiring high control output counts. Until recently, however, the only mechanism for implementing this approach has been to glue together an assortment of bit-slice component building-blocks. In addition, the approach also did not lend itself to rapid multi-way branching (a strength of Logic Arrays), instead being relegated to a serial test-and-binary-branch mechanism.

An enhanced vehicle for state machine implementation really requires a marriage of these two architectures, to obtain the high performance, multi-way branching based on real-time inputs characteristic of Logic Array-based sequencers, while having the ability to manage complex algorithms and generate high output counts characteristic of microcoded approaches. Altera's SAM family does exactly this.

SAM+PLUS SYSTEM OVERVIEW

The versatility of the SAM architecture, and its applicability to both State Machine and complex Controller functions, has necessitated the need for multiple design input formats. Altera's SAM+PLUS PC-based Design Software allows the designer to enter his design in either a high-level state machine description using Altera's ASMILE language, or in an efficient microcode assembler format known as ASM. A block diagram of this system is shown in Figure 2. Given these options, the user can employ the design description most appropriate for his particular problem, or with which he is personally most comfortable.

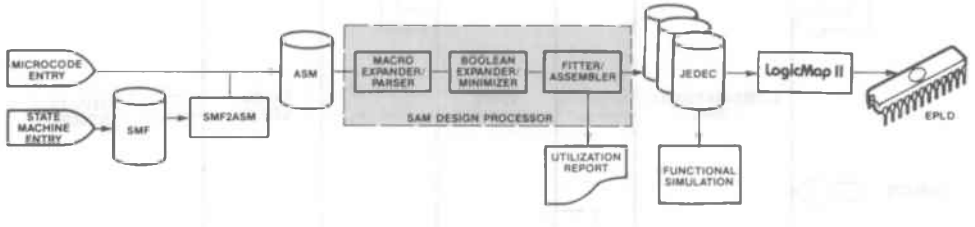
The SAM Design Processor (SDP) takes the input file and automatically minimizes the transition specification logic and fits the resultant resource requests into the SAM architecture. A Utilization Report is generated which reports total resources consumed, any unfittable requests, and assigned pinouts. Upon successful fitting, a standard JEDEC file is generated to allow programming of the device using a hardware programming card installed in the PC.

In addition, this JEDEC file, which represents the actual template of the specific application implemented, may be used as input to the SAMSIM (SAM SIMulator) program which provides functional simulation capability integrated into the total design environment. Hard-copy output of simulation results may be obtained, as well as on-line "logic analyzer" viewing capability. The result is a design entry, compilation and verification system which can be iterated rapidly until the desired functionality is obtained.

SIZING-UP A POTENTIAL SAM DESIGN

There are two broad categories of state machines. Mealy and Moore machines (see Figure 3). Given the SAM architecture, one can see that Moore machines may be directly implemented into a SAM component: SAM's outputs are a function of the currently addressed microcode location (state). Mealy machines specify outputs as functions of state and inputs. However, Mealy machines can frequently be converted to equivalent Moore machines. The general rule for this conversion is that for each transition into a state in the Mealy machine with a unique set of outputs, insert a state into the Moore machine with that output

Figure 2. SAM+PLUS System Diagram



combination. Figure 4 illustrates this concept.

ASMILE supports the resources available on SAM for state machine design. Additional features, such as the stack and counter, are supported in the microassembler format which lends itself to their efficient use.

In order to determine whether a given application is suitable for SAM, a few brief "rules-of-thumb" derived from the device architecture and specifications can prove helpful:

- Operating frequency less than or equal to specified SAM device's Fmax.
- Synchronous, Moore machine operation.
- Up to eight state machine Inputs (not including CLOCK or RESET).
- Up to sixteen state machine Outputs.
- Up to 64 Multi-Way (conditional) state branches.
- Transition expressions reduceable to 4 product terms per IF ... THEN expression.
- 192 or fewer unconditional state transitions.

An application which meets the above list of requirements will probably fit into a SAM device.

Figure 3. Types of Synchronous State Machines

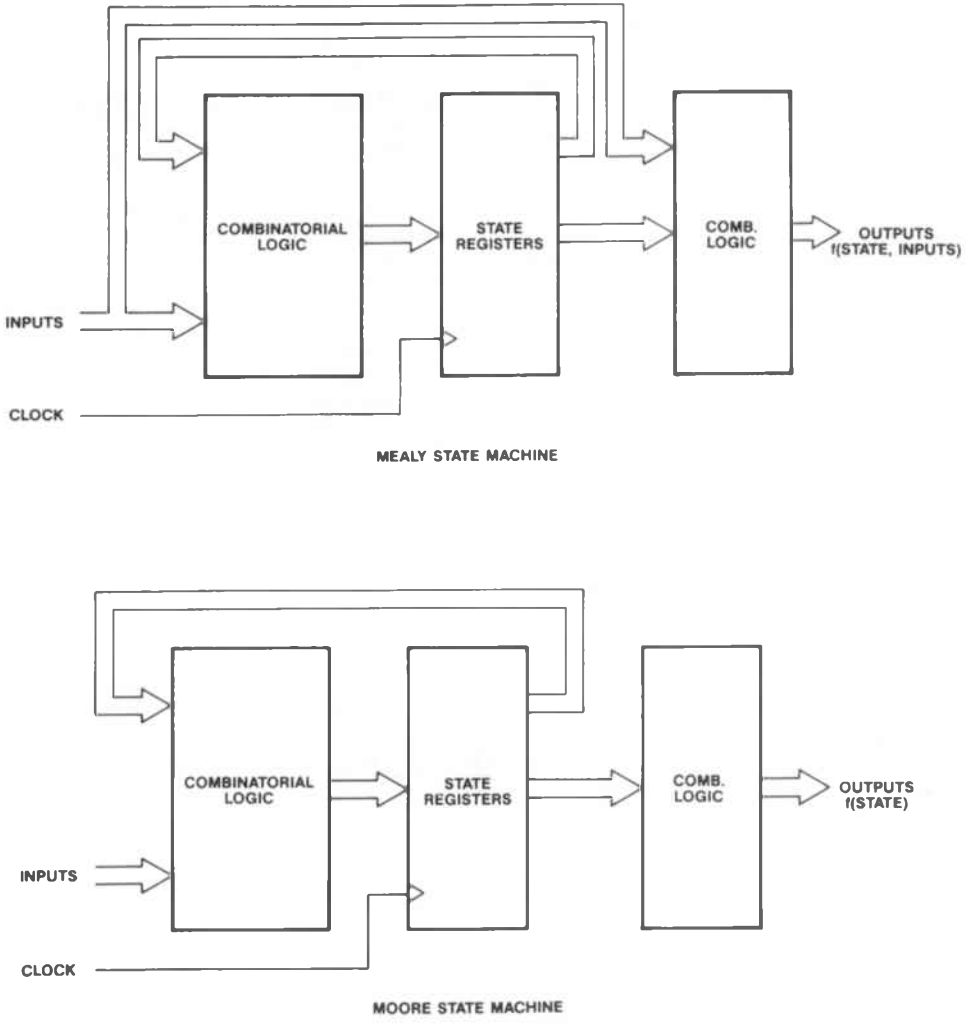
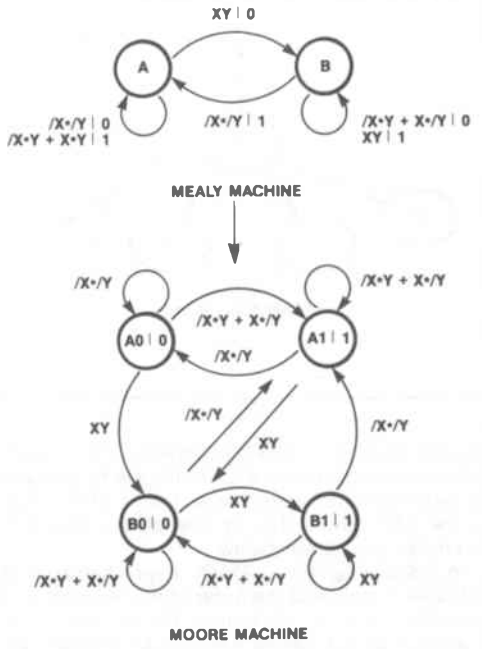


Figure 4. MEALY/MOORE Transformation



ASMILE ENTRY OVERVIEW

The basic format of a SAM ASMILE file consists of the following sections:

[Header]
PART
INPUTS
OUTPUTS
[EQUATIONS]
MACHINE
CLOCK
STATES
Transition Specifications
END$

Those sections surrounded by [] are optional and may be deleted if their use is not required in a given application.

ASMILE files may be constructed utilizing any standard text editor in non-document mode. Using an editor in document mode may inject spurious format control characters which will be detected as syntax error by the ASMILE parser at compile time. Other than this constraint, input is essentially free-form and may be structured for readability and overall clarity.

The case of characters inserted into the ASMILE file is significant, so it is important to insure that character case is maintained as text is entered. For example, the names "RWB" and "rwb" are not the same.

Comments may be inserted freely into the source code, delimited by leading and trailing percent signs, for example,

% This is a Comment %

HEADER

The header contains user-specified design identifier information. Typical information includes:

Designer's Name
Company
Date
Design Number
Revision
SAM Part Number
Other Comments

PART:

The PART section of the ASMILE file specifies the target SAM device the application is intended for.

INPUTS:

The single INPUTS section of the ASMILE file defines all external inputs into the design, as well as any required user pin assignments. Pin assignments are optional and will be assigned by SAM+PLUS if not specified. Pin assignments are specified by the format:

Input_name @ pin_number

OUTPUTS:

The OUTPUTS section of the ASMILE file contains a list of all outputs from the design as well as any pin assignments. Pin assignment syntax is similar to input pin assignments.

EQUATIONS:

The EQUATIONS section of the ASMILE file is available for the definition of intermediate equations to be used later in the design. Entry of transition specifications may be eased by defining intermediate variables initially, and then invoking them during the design. For example:

EventClk = I1*/I4 + I3*I6*/I7

might be defined in the EQUATIONS section, and then utilized later in an IF ... THEN statement.

MACHINE:

The format for the MACHINE declaration is:

MACHINE: machine_name

The MACHINE section of the ASMILE file actually specifies the state machine's state, output, and transition definitions required from the SAM device. There are three subsections which are to be included: CLOCK, STATES and Transition Specifications.

CLOCK:

The CLOCK subsection specifies the clock signal which will act as the synchronous clock source for the state machine and the resulting SAM device.

STATES:

The STATES section specifies all states in the target machine, as well as outputs corresponding to these states. The general form of this statement, when used in a SAM design, is

STATES: [output_name_1 ... output_name_n]

state_name [output_value_list]

In the above, the output_names are a list of all SAM output names used in the design, separated by whitespace. Following this initial declaration, a list of all state_names appears, each followed by a binary string in brackets which specifies all output values to be provided when the machine is in that state.

For example,

```
STATES:   [A B C D]
   S0     [0 0 0 0]
   S1     [0 1 1 0]
   S2     [1 0 0 0]
   S3     [0 0 0 1]
          •
          •
```

Specifies a machine with four outputs A through D, state S0 has all outputs low, S1 takes B and C to logic one, S2 has only output A high, etc.

TRANSITION SPECIFICATION

The form of the Transition Specification in a SAM ASMILE design is:

state_name : transition_specification

Every state in the machine must have a transition_specification which will specify successor states, either unconditionally:

S0: S2

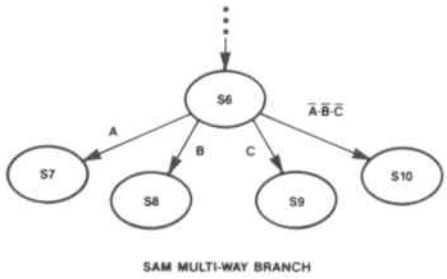
or conditionally using IF ... THEN statements.

The first state_name encountered in the Transition Specification section will be defined as the initial state of the machine coming out of Reset. As such, it has special significance. Typically, this might be defined as an "inactive" or passive machine state. Other Transition Specifications have no positional significance.

IF...THEN STATEMENTS

The SAM architecture implements in silicon the state transition specifications defined by a user in the chip's Branch Control logic block. This block allows, by its structure, the specification of up to 64 complex branching expressions in a single machine. [As noted above, up to 192 unconditional state transitions may be specified for a single SAM device]. Each IF ... THEN expression may specify a direct branch from the current state to as many as four other successor states, based upon inputs to the SAM device. This is illustrated in Figure 5. Examples are shown below.

Figure 5. SAM Multi-way Branch



In specifying IF ... THEN expressions, it is valuable to note that the order of the expression is important and can determine the machine flow. Transition specifications need not be mutually exclusive in such expressions. For example, the expression

```
S0: IF I1*I2 + I5 THEN S1
    IF I5*I6 + I4*/I3 THEN S2
    IF I4 THEN S3
    S4
```

might appear ambiguous under the condition that inputs I5 and I6 to the SAM device become true during S0. Is S1 or S2 the next state? At this point SAM's priority logic comes into play. Since the S1 transition is specified before the S2 in the design definition, it will be the next state entered. Similarly, if I4*/I3 become valid, S2 will be the next state entered in preference over S3. This precedence-resolving capability is provided in the SAM silicon architecture which employs a hardware priority encoder in selecting the next state transition. This capability resolves conflicts, and may be exploited in the design to prioritize transitions.

DEFAULT TRANSITIONS

One other benefit of this approach is the implicit "default" transition to be made. In the example above, S4 will be the next state entered if S1, S2 and S3 are not selected by the appropriate conditions being true. This feature can reduce design effort and resource requirements substantially, since default transitions are frequently defined as the negation of non-default transitions and such

inverted expressions have a tendency to consume logic product terms or resources quickly. For example,

```
S0: IF I1*I2 + I5*/I7 + I0 THEN S1
    IF I3 + /I6*I4 THEN S2
    IF I2*I3*I4*I5*/I7 THEN S3
    S4
```

is a valid ASMILE SAM transition specification. If the notion of a default transition (S4) was not in the ASMILE syntax, and had to be explicitly defined, we might have to specify the last transition as (unminimized):

```
IF /(I1*I2 + I5*/I7 + I0) * /(I3 + /I6*I4)
    * /(I2*I3*I4*I5*/I7) THEN S4
```

Each expression (IF ... THEN) may be a function of any of the eight SAM external inputs, and may contain up to 4 product terms after logic minimization. For most designs, this should prove ample.

A trade-off between number of branch destinations and product terms per destination can be made, as multiple IF ... THEN expressions can point to the same destination. For example, the expression

```
S0: IF (cond1) THEN S1
    IF (cond2) THEN S1
    IF (cond3) THEN S2
    S3
```

provides a three-way branch, with up to 8 product terms available for the specification of transitions to state S1.

END$

Every SAM ASMILE source file must terminate with the END$ terminator.

SAM ASMILE DESIGN EXAMPLE

To illustrate SAM ASMILE input syntax in a real example, a 68020 Microprocessor Bus Arbiter state machine will be examined. This machine, while not overly complex, illustrates most of the concepts of ASMILE entry.

Shown in Figure 7 is a state machine diagram for the Bus Arbiter. The 68020-based system runs at 25 MegaHertz, and therefore the Bus Arbiter machine must also run with a 40 nanoSecond clock period. To understand its operation, a review of the bus exchange protocol used on the 68020 bus is useful.

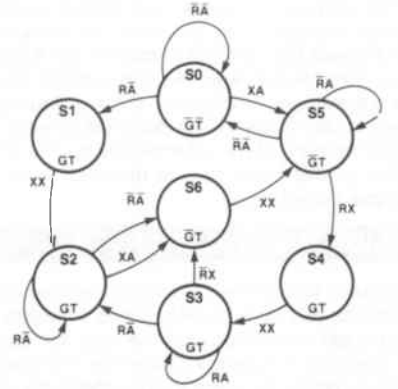
Three signal lines on a 68020 bus define the handshake required to arbitrate bus exchanges between multiple bus masters: Request, Grant, and Acknowledge. Given a bus master which desires access to the bus, the procedure is as follows (illustrated in Figure 6):

In the above flow description, the state labels S0-S6 designate correspondence between the operations shown and the state machine diagram above.

Figure 6. 68020 Bus Arbiter Operation

```
**************************************************

        Processor          Requesting Bus Master

S0
                          * Assert Request

S1 & S2
* Assert Grant

                          * External Arbitration (if
                              required) among multiple
                              Requests
                          * Wait for Completion of
                              Current Cycle
S6 & S5

                          * Next Bus Master Asserts
                              Acknowledge (ACK)
                          * Next Bus Master Deasserts
                              Request
* Deasserts Grant
  [ Wait for
    ACK to be
      deasserted ]

                          * Perform Bus Operations
                          *Deassert ACK
S0

* Resume Operation or

S4 & S3

    Rearbitrate

**************************************************
```

Figure 7. Arbiter State Flow



5

Relating this sequence to the state diagram, S0 represents the "normal", active state of the processor, S1 and S2 correspond to the Grant phase, S5 and S6 the Acknowledge phase, and S3 and S4 the rearbitration phase, if Requests are pending at the end of the current bus exchange.

DESIGN

The file shown in Figure 8 is the actual ASMILE file generated for the machine from the state diagram. It conforms to the general file outline as described above. ASMILE source files are given the extension .SMF (for state machine file) when generated. In this case, the file would be 68020ARB.SMF. Note that in the OUTPUTS and STATES sections, output variables OS0-OS6 have been defined which are each valid only during a unique state. As the design is simulated, these will give an indication of which state the machine is at any given point in time.

To compile this design, the SAM+PLUS software is invoked, specifying that ASMILE (and not microassembler) input format is being used. For a detailed description of the SAM+PLUS user interface and options, the SAM+PLUS User's Manual should be consulted. Compilation then proceeds automatically. Transition equations are automatically minimized, and "object code" generated for the EPLD and EPROM blocks. As a result, a JEDEC programming file (.JED) is generated, as well as a Utilization Report file (.RPT) reporting the results of the compilation process. Functional simulation of the design can be performed using the .JED file as a design template as described below. The .JED file is not intended to be user-readable. The .RPT file contains valuable information such as design pin assignments and resource utilization. Figure 9 shows key portions of this file. All ASMILE input is transformed into microassembler format before subsequent processing, and the equivalent microassembler code for the design is given in the .RPT file as well. More information on the interpretation of this code can be obtained from the references shown below.

DESIGN SIMULATION

Integral to the SAM+PLUS design system is the SAMSIM functional simulator. Once a design has been successfully processed, the user can specify input stimulus in a variety of formats and observe the device response quickly and effectively using this unit-delay simulator. As mentioned above, SAMSIM supports both hard-copy and virtual logic analyzer output formats. Split-window, multiple zoom-levels, and delta time display are a few of the capabilities of this interactive display mode.

SAMSIM supports both interactive and command file input. Shown in Figure 11 is a simple input stimulus command file for our design. Typically, command files are given the design name with the extension .CMD (for example, 68020ARB.CMD). The first line specifies the source design JEDEC (or .JED) file. The next two lines illustrate logic sequences for the two machine inputs. The PATTERN CREATE command allows the user to specify a sequence of input logic levels to be applied to the indicated node or nodes. The notation ( )*n, where n is an integer, signifies hold the indicated logic value on the associated input for n clocks. SIMULATE 41 instructs SAMSIM to run the simulation for 41 clocks, and interactive display is invoked with the VIEW command.

Some other representative SAMSIM commands, while not used in the example, include:

TRACE—Dumps entire state of machine (inputs, outputs, internal registers, etc.) for each clock executed.

GROUP—Specifies logical grouping of signals for easy observation or input vector specification.

Figure 8. 68020 Bus Arbiter State Machine Input File (68020ARB.SMF)

```
STAN KOPEC
ALTERA CORP.
3/10/87
68020 Bus Arbiter for SAM

% This description uses IF...THEN Transition Specifications%

PART: EPS444

% Pin Assignments (an option) are made by the designer %

INPUTS: REQUEST@1 ACK@2

OUTPUTS: GRANT@23 TRISTATE@22 OS0 OS1 OS2 OS3 OS4 OS5 OS6

MACHINE: BUSARBITER

CLOCK: CLK

% STATES gives the output value mapping %

STATES: [GRANT TRISTATE OS0 OS1 OS2 OS3 OS4 OS5 OS6]

  S0 [0 0 1 0 0 0 0 0 0]
  S1 [1 1 0 1 0 0 0 0 0]
  S2 [1 1 0 0 1 0 0 0 0]
  S3 [1 1 0 0 0 1 0 0 0]
  S4 [1 1 0 0 0 0 1 0 0]
  S5 [0 1 0 0 0 0 0 1 0]
  S6 [0 1 0 0 0 0 0 0 1]

% Transition Specifications follow%

S0:
  IF REQUEST*/ACK THEN S1
  IF ACK THEN S5
  S0

S1:
  S2

S2:
  IF /REQUEST*/ACK  + ACK THEN S6
  S2

S3:
  IF /REQUEST THEN S6
  IF REQUEST*/ACK THEN S2
  S3

S4:
  S3

S5:
  IF REQUEST THEN S4
  IF /REQUEST*/ACK THEN S0
  S5

S6:
  S5

END$
```

Figure 9. 68020 Bus Arbiter Design Report File (68020ARB.RPT)

```
STAN XOPEC
ALTERA CORP.
3/10/87
68020 Bus Arbiter for SAM

@(#) SAM Version 0.4 4/7/87 18:26:09 36.5

                                EPS444
                            ,--------_--------.
                REQUEST :   1          24 : NC
                    ACK :   2          23 : GRANT
                     NC :   3          22 : TRISTATE
                     NC :   4          21 : NC
                  CLOCK :   5          20 : NC
                    VCC :   6          19 : OS0
                  RESET :   7          18 : GND
                     NC :   8          17 : OS1
                     NC :   9          16 : OS2
                     NC :  10          15 : OS3
                     NC :  11          14 : OS4
                    OS6 :  12          13 : OS5
                            '----------------'

PART:
     EPS444

INPUTS:
     REQUEST@1, ACK@2

OUTPUTS:
     GRANT@23, TRISTATE@22, OS0@19, OS1@17,
OS2@16, OS3@15, OS4@14, OS5 OS6@12

PINS:

DEFAULT:
     [000000000]

PROGRAM:
     0D:
          [001000000] JUMP S0;
     192D:
     S0:
          IF REQUEST * ACK' THEN
               [110100000] JUMP S1;
          ELSEIF ACK THEN
               [010000010] JUMP S5;
          ELSE
               [001000000] JUMP S0;
     1D:
     S1:
          [110010000] JUMP S2;
     193D:
     S2:
          IF REQUEST' * ACK' *
             ACK THEN
                  [010000001] JUMP S6;
          ELSE
                  [110010000] JUMP S2;
     194D:
     S3:
          IF REQUEST' THEN
                  [010000001] JUMP S6;
          ELSEIF REQUEST * ACK' THEN
                  [110010000] JUMP S2;
          ELSE
                  [110001000] JUMP S3;
     2D:
     S4:
          [110001000] JUMP S3;
     195D:
     S5:
          IF REQUEST THEN
                  [110000100] JUMP S4;
          ELSEIF REQUEST' * ACK' THEN
                  [001000000] JUMP S0;
          ELSE
                  [010000010] JUMP S5;
     3D:
     S6:
          [010000010] JUMP S5;

END$

Statistical report:
        Number of label definitions      :    8
        Number of unconditional branches:     4
        Number of conditional branches   :    4
        Number of fatal errors           :    0
        Number of warnings               :    0
        Percent unconditional used       :    2.08%
        Percent conditional used         :    6.25%
```

SET—Modifies values of internal counter, stack, etc.

LINK—Logically links device pins for simulation purposes.

RADIX—Defines default radix for all SAMSIM commands. Options are binary, hex and decimal.

Running the SAMSIM simulator with this command file produces the results shown in Figure 10. The PC screen displays the input stimulus to the SAM arbiter design, and the resulting state machine operation.

The initial input stimulus applied to the SAM design shows a straightforward bus exchange between the 68020 and another bus master. This corresponds to the first REQUEST/GRANT/ACK sequence. Upon detecting a REQUEST, the 68020 asserts its TRISTATE line, and issues a GRANT pulse, allowing the new bus master to assume control. The alternate bus master asserts ACK when it detects the fact that the bus has been granted. When ACK finally drops, the 68020 knows it can resume control. The second sequence

Figure 10. SAMSIM Interactive Output

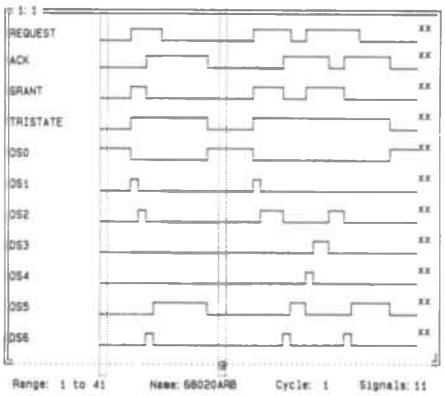


Figure 11. SAMSIM Command File (68020ARB.CMD)

```
JEDEC 68020ARB
PATTERN CREATE REQUEST = (0)*3 1 1 1 1 (0)*12 1 1 1 1 1 0 0 (1)*7 (0)*5
PATTERN CREATE ACK =     (0)*5 (1)*8 (0)*10 (1)*6 (0)*2 (1)*6 (0)*4
SIMULATE 41
VIEW
```

involves not just a single initial REQUEST (bus master #1), but a second REQUEST from another bus master (#2) during the time bus master #1 has control. As a result, the 68020 must generate a new GRANT pulse (during S4-S2), and hand-over bus control to bus master #2 when bus master #1 is finished (ACK is dropped). When bus master #2 is finished, and no requests are pending, the 68020 finally retakes control of the bus (TRISTATE goes low).

CONCLUSION

State machine design is a straightforward process using the ASMILE input language in conjunction with the SAM device. Design entry and debug, using functional simulation, can be readily accomplished at the user's PC. When the design is debugged and complete, the SAM component may be programmed using PC-based hardware and software in seconds. Should design errors be detected after in-system test, a windowed SAM device may be erased, a design change compiled, and the device reprogrammed in minutes.

AN10 Rev 2.0

INTRODUCTION

This Application Note describes the SAM microsequencer design entry process utilizing ASM microassembler input syntax and provides illustrations of all basic concepts needed to execute a SAM microassembler design. Basic microassembler functionality is reviewed, its utilization of SAM internal resources, as well as user convenience features. Cascading of multiple SAM devices to address large design problems is also covered. To illustrate a practical application of SAM, a graphics controller application is presented in detail along with annotated ASM source code.

The reader is referred to Altera's SAM EPS444/448 Data Sheet for details concerning device architecture and performance. A general knowledge of SAM device architecture is assumed as background for this Application Note.

SAM+PLUS SYSTEM OVERVIEW

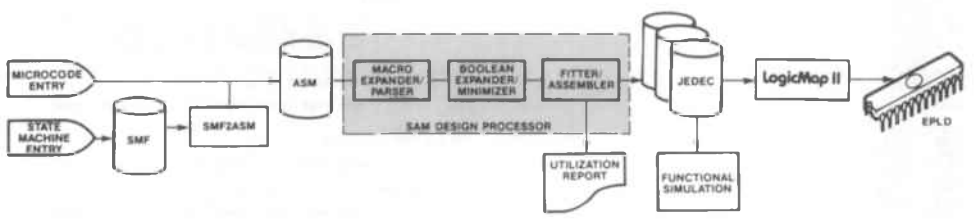
The SAM+PLUS PC-based design development system provides an efficient mechanism for entry and automatic compilation of SAM designs. Interactive functional simulation is provided in SAM+PLUS to enable rapid verification of design flows and operation. PC-compatible programming hardware is also available to allow device programming right at the designer's desk. Given the fact that control logic is frequently difficult to design, and particularly prone to design alterations, the ability to enter, compile, simulate and test a design in rapid fashion results in an effective design system.

SAM+PLUS actually supports two design entry methods, one using ASMILE state machine input language, the other ASM microassembler format, described in "SAM Applications Using State Machine Entry." This Application Note will focus on microassembler input.

Microassembler design entry begins with the creation of a design file on the PC using any standard text editor. Next, the SAM Design Processor (SDP) takes the ASM input file, automatically minimizes transition equations and generates the device programming code. A Utilization Report is generated which reports total resources consumed, absolute memory assignments of microassembler instructions and and compiler-assigned pinouts. A standard JEDEC file is generated to allow programming of the device right on the PC.

For larger designs, multiple SAM devices may be horizontally cascaded to increase the number of available control outputs. The microassembler supports the specification of a single source file for a multiple-SAM application, and automatically generates the separate JEDEC files for the programming of each of the devices at compile time. The JEDEC file, which represents the actual template of the specific application implemented, may be used as input to the SAMSIM (SAM SIMulator) program which provides functional simulation capability. Hard-copy output of simulation results

Figure 1. SAM+PLUS Block Diagram



5

may be obtained, as well as on-line "logic analyzer" viewing capability. Multi-chip applications using horizontal cascading is also supported by the functional simulator.

CHOOSING APPROPRIATE SAM APPLICATIONS

The SAM architecture supports high-performance synchronous control applications. It is important to realize that all outputs from SAM are asserted synchronously with respect to the device clock, thus SAM implements a classic Moore machine architecture. Also, as can be seen in the SAM Data Sheet, all inputs must obey a required set-up time (Tsu) relative to the Clock input.

In order to obtain greater than 16 outputs in a SAM design, the concept of horizontal cascading may be used. Similarly, if greater control store (microcode) depth is required, multiple SAM devices may be vertically cascaded, sharing a common control output bus. Both cascading approaches may be simultaneously used for problems requiring increased capacity in both dimensions.

In order to determine whether a given application will be suitable for SAM, the following "rules-of-thumb" derived from the device architecture and specifications are useful. These guidelines are for single SAM implementations. Cascaded SAM configurations may expand output count and memory depth substantially. For example, SAM+PLUS supports horizontal cascading of up to 8 EPS448 devices, for a total output count of 128 lines.

Applications which satisfy the following constraints will most likely fit into a single EPS448 SAM device:

1) Operating frequency up to specified SAM's Fmax.

2) Synchronous operation.

3) Up to eight control inputs (exclusive of Clock and nRESET).

4) Up to sixteen control outputs (single device).

Figure 2. Circle Drawing Routine

```
This is the Circle Drawing Design

% Circle Drawing Routine for SAM %

PART: EPS448 EPS448

% SAM Control Output Lines              Inputs          %
%  A & B Fields (2901)   - 8        C0-2          - 3    %
%  I0-I8 (2901)          - 9        CmdAtt        - 1    %
%  OE (2901)             - 1        Sign          - 1    %
%  Done                  - 1                             %
%  Cn (2901)             - 1                             %
%  Wr                    - 1                             %
%  ALE                   - 1                             %
%  Rd                    - 1                             %
%  ReqRd                 - 1                             %

INPUTS: C0,C1,C2,CmdAtt,Sign

OUTPUTS: A0,A1,A2,A3,B0,B1,B2,B3,I2,I1,I0,I5,I4,I3,I8,I7
OUTPUTS: I6,Rd,Wr,ALE,ReqRd,OE,Cn,Done

DEFAULT: [0000 0000 0000 0000 1110 0100]

MACROS:

  CONT = "CONTINUE"

% A & B Fields %

  RadiusReg = "0001"
  Reg1 = "0001"
  Reg2 = "0010"
  Reg3 = "0011"
  Reg4 = "0100"
  Reg5 = "0101"
  Reg6 = "0110"
  Reg7 = "0111"
  Reg8 = "1000"
  Reg9 = "1001"
  Reg10 = "1010"
  Reg11 = "1011"
  Reg12 = "1100"

% Source Control %

  AQ = "000"
  AB = "001"
  ZQ = "010"
  ZB = "011"
  ZA = "100"
  DA = "101"
  DQ = "110"
  DZ = "111"

% Function %

  ADD = "000"
  SUBR = "001"
  SUBS = "010"
  OR = "011"
  AND = "100"
  NOTRS = "101"
  EXOR = "110"
  EXNOR = "111"

% Destination  Control %

  QREG = "000"
  NOP = "001"
  RAMA = "010"
  RAMF = "011"
  RAMQD = "100"
  RAMD = "101"
  RAMQU = "110"
  RAMU = "111"

% Bus Cycle  %

  MemWr = "10001"
  RegWr = "10011"
  ALEcyc = "11100"
  NoCyc = "11000"

% Misc %

  Cn = "1"
  nCn = "0"
  Done = "1"
  nDone = "0"


EQUATIONS:

PROGRAM:

% Processor Initializes: %

% o Load Coloreg, Radius, X0, Y0 %
% o Issues DrawCirc Command      %

WAIT:       IF CmdAtt*C0'*C1'*C2' THEN [] JUMP DOIT ;
              ELSE [] JUMP WAIT ;

% Move parameters from buffer to 2901 internal registers %
% Radius -> Reg1 (Y) %
DOIT:       [ Reg1 Reg1 AQ ADD NOP RegWr nCn nDone ] CONT ;
            [ Reg1 Reg1 AQ ADD NOP RegWr nCn nDone ] CONT ;

% X0 -> Reg2 %

            [ Reg2 Reg2 AQ ADD NOP NoCyc nCn nDone ] CONT ;
            [ Reg2 Reg2 AQ ADD NOP RegWr nCn nDone ] CONT ;
            [ Reg2 Reg2 AQ ADD NOP RegWr nCn nDone ] CONT ;

% Y0 -> Reg3 %

            [ Reg2 Reg2 AQ ADD NOP NoCyc nCn nDone ] CONT ;
            [ Reg3 Reg3 AQ ADD NOP RegWr nCn nDone ] CONT ;
            [ Reg3 Reg3 AQ ADD NOP RegWr nCn nDone ] CONT ;

% Load constants to 2901 registers %
% 0 -> Reg4 (X) (AND 0 & anything gives 0) %

            [ Reg4 Reg4 ZB AND RAMF NoCyc nCn nDone ] CONT ;

% 1 -> Reg5 (d) %
% Put "1" in Reg5 %

            [ Reg4 Reg5 ZA ADD RAMF NoCyc Cn nDone ] CONT ;

% Shift Reg5 Up one to give 2 %

            [ Reg5 Reg5 ZB ADD RAMU NoCyc nCn nDone ] CONT ;

% While we have it, preload 2 into Reg9 %

            [ Reg5 Reg9 ZA ADD RAMF NoCyc nCn nDone ] CONT ;
```

Figure 2. Circle Drawing Routine (Continued)

```
% Increment Reg5 to get 3  (whew!!) %

          [ Reg5 Reg5 ZA ADD RAMF NoCyc Cn nDone ] CONT ;

% 6 -> Reg8 (const) - just shift 3 up one! %
% Load 1 in CREG to set-up for next instruction %

          [ Reg5 Reg8 ZA ADD RAMU NoCyc nCn nDone ] LOADC 1D ;

% 10 -> Reg9 (const) %
% Start by shifting Reg5 (now contains 2) up twice to get 8 %
% Reg6 (Temp register)  %

SHIFTR9:   [ Reg9 Reg9 ZA ADD RAMU NoCyc nCn nDone ]
             LOOPNZ SHIFTR9 ;

% Increment Reg9 twice to get 10 %

          [ Reg9 Reg9 ZA ADD RAMF NoCyc Cn nDone ] CONT ;
          [ Reg9 Reg9 ZA ADD RAMF NoCyc Cn nDone ] CONT ;

% Initializing done ! - Begin algorithm %
% d = 3 - 2*radius initially %

          [ Reg1 Reg6 ZA ADD RAMU NoCyc nCn nDone ] CONT ;
          [ Reg5 Reg6 AB SUBS RAMF NoCyc nCn nDone ] CONT ;

% If x >= y branch to finish up %

OUTERLOOP: [ Reg4 Reg1 AB SUBS RAMF NoCyc Cn nDone ] CONT ;
          IF Sign THEN [] JUMP DrawEnd ;

% Write pixels, translate origin & reflect to all octants %

            ELSE [] CALL CircPix ;

% Test d sign, If >= 0, use POS %

          [ Reg5 Reg5 ZA ADD RAMF NoCyc nCn nDone ] CONT ;
          IF Sign THEN [] JUMP POS ;

% Compute d = d + 4*x + 6 %
% First 4*x %

            ELSE [ Reg4 Reg6 ZA ADD RAMU NoCyc nCn nDone ] CONT ;
          [ Reg6 Reg6 ZA ADD RAMU NoCyc nCn nDone ] CONT ;

% Add 6 %

          [ Reg8 Reg6 AB ADD RAMF NoCyc nCn nDone ] CONT ;
          [ Reg6 Reg5 AB ADD RAMF NoCyc nCn nDone ] JUMP IncX ;

% Compute d = d + 4*(x-y) + 10 %
% First x-y %

POS:       [ Reg1 Reg6 ZA ADD RAMF NoCyc nCn nDone ] CONT ;
          [ Reg4 Reg6 AB SUBS RAMF NoCyc Cn nDone ] LOADC 1D ;

% Then 4*(x-y) %

SHIFTR6:   [ Reg6 Reg6 ZA ADD RAMU NoCyc nCn nDone ]
             LOOPNZ SHIFTR6 ;

% Add 10 %

          [ Reg9 Reg6 AB ADD RAMF NoCyc nCn nDone ] CONT ;
          [ Reg6 Reg5 AB ADD RAMF NoCyc nCn nDone ] CONT ;

% Decrement y %

          [ Reg1 Reg1 ZA SUBR RAMF NoCyc nCn nDone ] CONT ;

% Increment x  and repeat til x = y %

IncX:      [ Reg4 Reg4 ZA ADD RAMF NoCyc Cn nDone ] JUMP OUTERLOOP ;

% Last pixel write / ends octant with x = y (45 degrees) %

DrawEnd:   [] Call CircPix ;
           [] LOADC 16D ;

% Issue Done to processor for 16 clocks %

DoDone:    [ Reg1 Reg1 ZA ADD RAMF NoCyc nCn Done ]
             LOOPNZ DoDone ONZERO WAIT ;

% End Main Routine %

%  This  routine reflects the pixel into all octants and calls  a
   routine which translates the pixel relative to x0,y0,
   calculates the pixel address as addr = x + y*1023 and runs the
   memory cycle. %

CircPix:   [ Reg4 Reg6 ZA ADD RAMF NoCyc nCn nDone ] CONT ;
           [ Reg1 Reg11 ZA ADD RAMF NoCyc nCn nDone ] CALL TRANS ;

% Reflect X to -X %

           [ Reg4 Reg6 ZA SUBS RAMF NoCyc Cn nDone ] CONT ;
           [ Reg1 Reg11 ZA ADD RAMF NoCyc nCn nDone ] CALL TRANS ;

% Swap X & Y %

           [ Reg1 Reg6 ZA ADD RAMF NoCyc nCn nDone ] CONT ;
           [ Reg4 Reg11 ZA ADD RAMF NoCyc nCn nDone ] CALL TRANS ;

% Swap -X & Y %

           [ Reg4 Reg11 ZA SUBS RAMF NoCyc Cn nDone ] CONT ;
           [ Reg1 Reg6 ZA ADD RAMF NoCyc nCn nDone ] CALL TRANS ;

% Reflect Y %

           [ Reg1 Reg11 ZA SUBS RAMF NoCyc Cn nDone ] CONT ;
           [ Reg4 Reg6 ZA ADD RAMF NoCyc nCn nDone ] CALL TRANS ;

% Swap -Y & X %

           [ Reg1 Reg6 ZA SUBS RAMF NoCyc Cn nDone ] CONT ;
           [ Reg4 Reg11 ZA ADD RAMF NoCyc nCn nDone ] CALL TRANS ;

% Reflect -X, -Y  %

           [ Reg4 Reg6 ZA SUBS RAMF NoCyc Cn nDone ] CONT ;
           [ Reg1 Reg11 ZA SUBS RAMF NoCyc Cn nDone ] CALL TRANS ;

% Swap -X & -Y   %

           [ Reg1 Reg6 ZA SUBS RAMF NoCyc Cn nDone ] CONT ;
           [ Reg4 Reg11 ZA SUBS RAMF NoCyc Cn nDone ] CALL TRANS ;
           [] RETURN ;

% This routine Translates relative to x0,y0 and runs the memory
  update cycle %

TRANS:     [ Reg1 Reg11 AB ADD RAMF NoCyc nCn nDone ] CONT ;
           [ Reg2 Reg6 AB ADD RAMF NoCyc nCn nDone ] LOADC 10D ;
           [ Reg11 Reg12 ZA ADD RAMF NoCyc nCn nDone ] CONT ;

% Multiply y by 1024 %

MULT1024:  [ Reg11 Reg11 ZA ADD RAMU NoCyc nCn nDone ]
             LOOPNZ  MULT1024 ;

% Subtract y to get effective multiply by 1023 %

DONE1024:  [ Reg12 Reg11 AB SUBR RAMF NoCyc Cn nDone ] CONT ;

% Calculate address %

           [ Reg6 Reg11 AB ADD RAMF NoCyc nCn nDone ] CONT ;

% Write pixel in buffer RAM %

RUNBUS:    [ Reg11 Reg11 ZA ADD RAMF ALEcyc nCn nDone ] CONT ;
           [ Reg11 Reg11 ZA ADD RAMF MemWr nCn nDone ] RETURN ;

ENDS
```

5) Up to 256 primary microcode locations.

6) Up to 64 of 256 primary microcode locations may be multi-way (external conditional) branches (single device).

7) Transition expressions reduceable to 4 product terms per IF ... THEN expression.

MICROASSEMBLER INPUT

Shown in Figure 2 is an example of the structure of a SAM ASM input file. This file may be created using any standard text editor. It is important that the text editor is used in non-document mode in order to prevent the insertion of any spurious format control characters which may be detected by the ASM microassembler parser at compile time as input errors. Other than this constraint, input is essentially free-form and may be structured for readability and overall clarity.

The case of characters inserted into the ASM file is significant, so be sure that case significance is maintained. For example, the names "RWB" and "rwb" are not the same.

Comments may be inserted freely into the source code, delimited by leading and trailing percent signs (%).

The basic format of a SAM ASM file consists of the following sections:

[HEADER]
PART
INPUTS
OUTPUTS
[PINS]
[DEFAULT]
[MACROS]

[EQUATIONS]
PROGRAM
END$

Those sections noted within brackets are optional and may be ommitted if not required.

HEADER:

The header contains user-specified design identifier information. It may include design title, designer's name, date, revision information, etc.

PART:

The PART section of the ASM file specifies the target SAM device or devices the application is intended for. By specifying AUTO, the user permits the SAM+PLUS software to pick the optimal device or set of devices for the application based upon minimal pin count. Multiple devices may be invoked for designs requiring a larger number of total outputs than a single SAM device can supply, i.e., the SAM+PLUS software supports horizontal cascading (see SAM Datasheet) of devices at a source code level. This cascading capability may be invoked by utilizing AUTO with a design requiring high output count as noted, or may be explicitly defined by supplying a list of devices after PART: which the design is to be fitted into. As shown in the example below, two EPS448 devices are going to be used in this application, and have been explicitly entered. Devices may be cascaded horizontally up to a width of 128 outputs in a single source code listing and simulated as one large virtual SAM. Separate JEDEC files are generated for each device to support programming devices when design is complete.

INPUTS:

The single INPUTS section of the ASM file defines all external inputs into the design, as well as any required user pin assignments. Pin assignments are specified by the format input_name @ pin_number. Note that since in a horizontally cascaded design all design inputs must be common, there will never be more inputs specified in a source file than are available in a single SAM device.

Only user-defined inputs should appear in the INPUTS section: the CLOCK and nRESET inputs to SAM, being fixed-function pins, should not be included.

OUTPUTS:

The OUTPUTS section(s) of the ASM file contains a list of all outputs from the design as well as any pin assignments. Pin assignment syntax is similar to input pin assignments. If multiple SAMs are specified in the PART: section of the design file (horizontal cascading), there will be multiple OUTPUTS sections in the ASM file, one for each SAM component. If AUTO parts selection is used for a cascaded design, a single OUTPUTS declaration may be used to specify all required outputs. At compile time, outputs will be assigned to the various devices automatically.

Output names must be unique across all OUTPUTS section declarations.

AUTO parts selection may not be used in conjunction with user-defined pin assignments.

PINS:

The PINS section allows mapping of external variable names onto internal variable names for convenience. For example, a user may have an active-low signal in his system he has called /WR which enters into his transition specifications in his SAM design. To keep the logical sense of such specifications clear, it is wise to transform all active-low external signals into equivalent active-high names internally, e.g., /WR = WRint.

DEFAULT:

The DEFAULT section allows the specification of a default output combination to be used whenever the output string is not explicitly defined in an instruction. In a single SAM device specification the syntax is simply DEFAULT: [O0.....On], where O0 through On represents a binary string corresponding to the n outputs specified for the SAM design. Default output values are matched to output pins in the order they appear in the OUTPUTS declaration. If multiple OUTPUTS sections appear in a cascaded SAM application, the DEFAULT specifier is increased in width to accommodate this change as shown in the example. Only one DEFAULT section may appear per ASM file.

MACROS:

The MACROS section allows the user to define string equivalences to be substituted universally throughout his ASM source code listing. For example, the user may wish to redefine instruction mnemonics for efficiency or clarity, or may wish to redefine binary output strings with alphanumeric labels. For example,

REG1TOALU = "0101111001100000"

The left hand side of this expression is undoubtedly easier to remember and type repeatedly into a listing than the right.

Imbedded strings are not macro substituted. Macro instances must be delimited by white space to be recognized. For example, if a macro substitution is defined as

REG = "0110"

the string 0110 would be substituted into

[REG ALU OP] CONTINUE;

but not into

[BREG4 AL OP] CONTINUE;

EQUATIONS:

The EQUATIONS section of the ASM file is

available for the definition of intermediate equations to be used later in the design. Entry of transition sepcifications may be eased by defining intermediate variables initially, and then invoking them during the design. For example,

EventClk = I1*/I4 + I3*I6*/I7

might be defined in the EQUATIONS section, and then utilized later in an IF ... THEN ... ELSE statement or statements, such as

IF EventClk THEN JUMP START:

PROGRAM:

The PROGRAM section of the ASM file actually specifies the sequence of instructions to be executed and associated outputs required from the SAM device. The format of a basic instruction specification in the PROGRAM section is

label: [output-spec] opcode;

label is an optional alphanumeric string which may be used to identify the instruction in branching expressions, etc. [output-spec] represents an actual numeric string of the correct length (in either binary, hexadecimal or decimal notation), a Macro substitution with numeric equivalence (as defined above), or the special character Z which signifies tristate output pins. Hexadecimal and decimal strings are defined by a string of valid digits of correct length, followed by H or D respectively. In horizontally cascaded applications, all outputs are specified in the single output-spec within brackets. The output-spec defined in the DEFAULT statement will be utilized whenever the output-spec has length zero, i.e., [] implies default output-spec.

END$

Every SAM ASM source file must terminate with the END$ terminator.

MULTI-WAY BRANCH SYNTAX

The syntax for multi-way branching within the SAM ASM source file is by way of a complex expression of the form:

IF (expression1) THEN (instruction1)
ELSEIF (expression2) THEN (instruction2)
ELSEIF (expression3) THEN (instruction3)
ELSE (instruction4)

For example, a complex instruction of this type might look like:

IF I0*I1*I5*/I7 + I3*I4 +I6*/I0 + /I3*/I1 THEN [1111001110010000] CALL label1 RETURNTO label2;
ELSEIF I3*/I2 + I5*I6 + /I0*I4*I1 THEN [1011000011100011] LOADC 255 GOTO label3;
ELSEIF I4*I6*I0 THEN PUSH 15 GOTO label4;
ELSE [1111111100000001] PUSHI GOTO label5;

Each expression may be a function of any of the eight SAM external inputs containing up to 4 product terms.

If more than 4 product terms are needed to define a transition from one state to another, it is possible to trade-off product term counts for number of multi-way branch destinations. For example, it is perfectly valid to enter:

IF (expression1) THEN [ ] JUMP START;
ELSEIF (expression2) THEN [ ] JUMP START;
ELSEIF (expression3) THEN [ ] JUMP NEXT1;
ELSE JUMP NEXT2;

Here, expression1 and expression2 could each be 4 product term expressions, resulting in 8 product terms which can be used to specify the transition to START.

Note the inherent priority scheme in the above statements. The SAM architecture physically implements such a priority scheme in the Branch Control Block: the first occurrence of a valid expression results in the execution of the corresponding instruction. If the first three expressions are all false, then instruction4 will subsequently executed.

Up to 64 such IF ... THEN ... ELSE constructs may be implemented in a single SAM program, along with 192 conventional instructions without IF ... THEN ... ELSE. The result is a total microcode memory capacity of (64 x 4) + 192 = 448 words.

SAM MICROASSEMBLER OPCODES

The basic SAM device instruction set accessible by the user through the microassembler consists of:

CONTINUE

Execute next sequential instruction

JUMP (label1)

Jump to instruction specified @ label1

LOOPNZ (label1) ONZERO (label2)

If Count Register (CREG) is zero, execute instruction @ label2, else decrement CREG and execute instruction @ label1. Useful for one-instruction timing and delay loops.

DECNZ GOOT (label1)

Decrement the CREG if non-zero; execute instruction @ label1.

POPC GOTO (label1)

Top-of-Stack is popped into CREG and the instruction @ label1 is executed.

POPXORC (constant1) GOTO (label1)

Top-of-Stack is popped, bitwise XORed with (constant1) and loaded to CREG. Instruction @ label1 is next executed. Useful for comparing Top-of-Stack to a value by subsequently testing CREG zero-flag using a LOOPNZ instruction.

LOADC (constant1) GOTO (label1)

CREG is loaded with the value constant1, and instruction @ label1 is next executed.

RETURN

Address of the next instruction is popped from Top-of-Stack and subsequently executed. Used to terminate subroutines.

PUSHLOADC (constant1) GOTO (label1)

CREG value is pushed onto the Stack and CREG is reloaded with constant1.

PUSHI GOTO (label1)

The eight input lines are pushed onto the Top-of-Stack and the instruction at label1 is subsequently executed. May be used to implement a "dispatch" function in conjunction with a subsequent RETURN instruction: external inputs provide address of next SAM instruction.

ANDPUSHI (constant1) GOTO (label1)

The eight input lines are bitwise ANDed with constant1, the result is pushed onto the Stack and the instruction @ label1 is subsequently executed. May be used to mask inputs before loading to CREG or next address.

CALL (label1) RETURNTO (label2)

Label2 is pushed onto the Stack, and the instruction @ label1 is executed next. Used for subroutines.

PUSH (constant1) GOTO (label1)

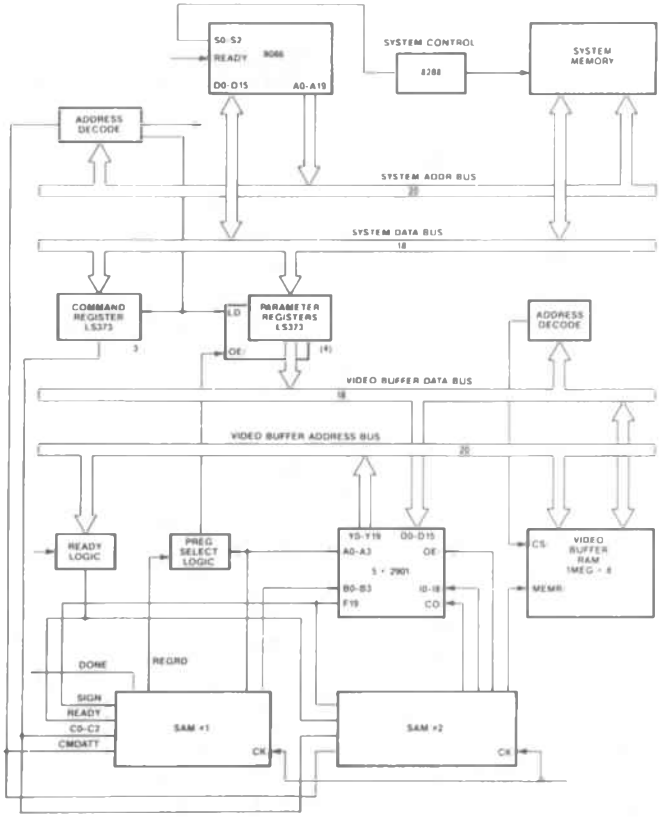
Constant1 is pushed onto the Stack and the instruction @ label1 is next executed.

The Branch Control Block of SAM is invoked automatically by use of IF ... THEN ... ELSE constructs in conjunction with the above instructions. This allows program flow control based upon external inputs ala conventional state machines and multi-way branching in a single clock.

DESIGN EXAMPLE

Now that the basic syntax and elements of a

Figure 3. SAM Graphics Engine



SAM ASM file have been covered, a detailed example of a SAM application will be presented: a high-performance Graphics Controller. In this particular application, two SAM devices will be horizontally cascaded to generate the control outputs for a graphics subsystem. This subsystem provides graphics primitive drawing capability for a larger microprocessor-based system.

Figure 3 shows a typical 8086 microprocessor-based system. Beneath the Address/Data Buses is the graphics subsystem to be controlled by the SAM devices, the primary elements of which are a 1 Megabyte high-speed static RAM video frame buffer (giving individual pixel addressing capability), five 2901 bit-slice elements used to construct a 20 bit ALU/data path engine, and two SAM devices as previously mentioned to provide overall control within the subsystem.

This basic graphics engine represents a user-microcodeable arrangement which can potentially support many primitive graphic drawing operations such as line drawing, polygon filling, drawing of conic sections and others. For the purposes of this example, a single primitive drawing operation which draws circles of arbitrary radius and origin into the frame buffer will be discussed. The basic concept behind this algorithm will be discussed below.

In order to execute its role of controller for this subsystem, the pair of SAM devices must be able to execute the following subfunctions:

- Read Commands issued by main microprocessor
- Transfer Parameters associated with commands to Register File in 2901's
- Initialize Constant Registers in 2901's to specified values for algorithm
- Compute values for pixels on circle as function of specified Radius for first octant [Assume circle origin = (0,0)]
- Translate x,y coordinates into RAM addresses
- Reflect circle pixel coordinates into remaining seven octants
- Translate pixel coordinates relative to actual origin
- Perform Video Buffer write to all pixel addresses specified
- Issue DONE interrupt to main processor

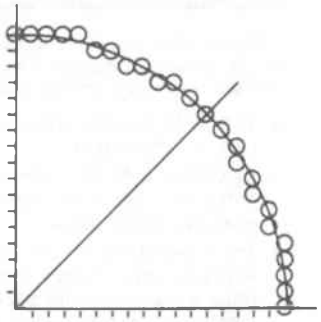
This activity is done independently of the main microprocessor and frees it up to do other tasks while the operation is performed.

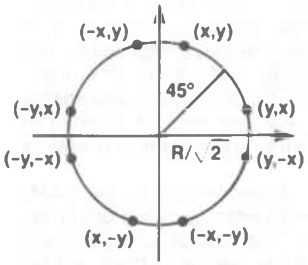
These operations fall into two general categories of controlling bus transfers between various elements (Registers, ALU, RAM, etc.) and sequencing computations performed by the 2901 ALU in generating the pixel addresses to be set to draw the required circle. The structure of the SAM microassembler code shown above generally follows this flow.

CIRCLE DRAWING ALGORITHM

The sample algorithm to be implemented in the SAM code to draw the circle is one based upon a methodology developed by Bresenham. In order to speed computation, it exploits the fundamental symmetry of a circle, by calculating the circle points in the first octant (see Figure 4), and then reflecting those coordinates into the other seven

Figure 4. Circle Symmetry Exploited by Bresenham





octants. For a given pixel location (x,y), reflection involves drawing points (-x,y), (x,-y), (-x,-y), as well as those points with x and y swapped. In drawing the points for a circle in the first octant, one can easily see that, having just calculated one of the pixel locations, there are only two possible choices for the next pixel location: increment x (horizontal move) and increment x and y (diagonal move). The trick is how to decide, based upon current location, which of the two to pick next.

The entire derivation of the algorithm will not be presented here. However, a complete discussion of the algorithm may be found in Foley and Van Dam (1981), referenced below. Suffice it to say, it is obvious that the best match between actual pixel coordinates and the ideal circle points can be obtained by checking an error term equal to the difference in distance from the circle's center to each of the two potential next pixel choices: the sign of the term will indicate which point to pick to obtain the best fit. The basic algorithm implemented is shown in Figure 5.

Figure 5. Circle Drawing Algorithm

```
procedure circle (radius, value : integer) ;

   var x,y,d : integer ;

begin

   x := 0 ;
   y := radius ;
   d := 3 - 2 * radius ;

  while x < y do begin

      CircleDraw (x,y,value);

      if d < 0
         then d := d + 4 * x + 6
         else begin
              d := d + 4 * (x - y) + 10 ;
              y := y - 1
         end

      x := x + 1

  end

  if x = y then CircleDraw (x,y,value) ;

end
```

TIMING CONSIDERATIONS

SAM timing analysis is straightforward, as all times are relative to the synchronous clock input. Tsu specifies minimum set-up time for inputs to gain recognition at the next clock edge, while Tco specifies clock-to-output delays for the user-configured output pins. Output tristate and enable times are speced as Tcz, but are not relevent in this particular application as outputs are always enabled.

For this particular design example, the SAM-controlled graphics subsystem is being driven by a 20 MegaHertz clock. This implies a clock period of 50 nanoSeconds. SAM control outputs will reflect a Tco of approximately 15 nanoSeconds, while inputs must obey a 15 nanoSecond set-up time (Tsu) relative to the clock edge.

High-speed Static RAM will be used for the video frame buffer for two reasons: one is raw speed. The memory must be fast enough to keep-up with SAM's high-speed bus cycles. The second is that SRAM requires no refresh cycles, unlike DRAM. Thus more time is available to perform buffer drawing functions: no time is lost for refresh cycles.

Memory consists of CMOS SRAM components organized 8K × 8 with an access time of 45 nanoSeconds, and a minimum Write Pulse width of 30 nanoSeconds. The CMOS 2901 bit-slices require a 30 nanoSecond propagation delay from A and B Register Address inputs to valid Y output, and a 10 nanoSecond set-up time prior to the Clock high-to-low transition on A and B inputs. A timing diagram is shown in Figure 6.

The bus cycle uses a two clock approach. During the first cycle, the 2901 will generate a pixel address to be set, and during the second cycle, the actual write pulse will be generated by SAM to write the frame buffer.

Operations performed entirely within the 2901 slices (register transfers, ALU operations, etc.) are all executed in a single clock cycle. Note that Carry Lookahead circuitry is employed with the 2901 slices to improve arithmetic computation times, but is not explicitly shown in the block diagram.

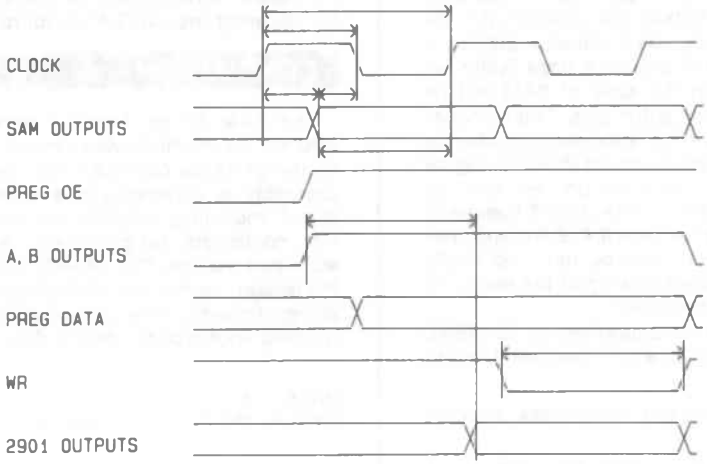
The algorithm below uses many of the 2901's operations, as well as many of the internal addressing modes. In the following listings, standard mnemonics have been used for the various Source, Destination and Operation specifiers. These control lines for the 2901's are all generated by the SAM devices. These mnemonics, and resulting 2901 functions, may be found in any standard 2901 Data Sheet, available from multiple vendors.

EXAMPLE PROGRAM LISTING

Figure 2 above is a source listing of the basic circle drawing process. The following comments are worth noting before going further:

- Two SAM devices are used in a horizontal cascade configuration.
- Extensive MACRO definitions to ease design entry and allow the use of user- and 2901-specified mnemonics.
- Two subroutines, CircPix and Trans, are invoked multiple times to draw the circle pixels. CircPix reflects the pixels into all octants of the circle as mentioned above, while Trans translates the pixels relative to the actual circle origin and runs the memory update cycle. These functions utilize the Stack and subroutining resources on SAM.
- Since the display is assumed to be 1024 × 1024

Figure 6. Primary SAM—2901 Graphics Controlled Timing



pixels, x and y pixel coordinates must be converted to SRAM address locations by multiplying the y coordinate by 1023 and adding the x coordinate.

- The signal CmdAtt is an input to the SAMs from the main processor, signaling that all parameters are loaded to the Parameter Registers, and that a circle drawing operation should be executed. DoneInt is a signal from SAM to the processor, asserted when the drawing operation is complete.

COMPILING THE DESIGN

By convention, microassembler source files are given the extension .ASM. This file is called CIRC.ASM. Compilation of this design involves invoking the SAM+PLUS software and specifying ASM microassembler input format. A variety of runtime options for SAM+PLUS are available, which provide special reporting modes and logging simulation input and output to a special file. For detailed descriptions of the SAM+PLUS user interface and options, see the SAM+PLUS User's Manual. Compilation is an automatic process resulting in the generation of programming "object code" for the EPLD and EPROM blocks on SAM. In this case, two programming files will be generated, since two devices are required to implement the design. These two files are given the extensions .JD1 and .JD2 to distinguish them. These JEDEC files are not intended to be user readable (as with any object code). Functional simulation uses these programming files for its modelling of SAM operation. An additional product of the compilation is a single Report file (extension -.RPT) which describes the resources which have been used in the SAM devices, pin assignments which have been selected and absolute locations within SAM's microcode assigned to the instructions entered. Figure 7 shows key portions of the CIRC.RPT report file. Notice the assigned pinouts for the two devices, as well as the substitution of absolute addresses for logical labels.

DESIGN SIMULATION

The SAMSIM functional simulator allows simulation of single-, as well as horizontally-cascaded SAM designs. Once a design has been successfully compiled, the user can specify input stimulus in a variety of formats and observe the device response. SAMSIM supports both hard-copy waveform and tabular output, as well as interactive "virtual logic analyzer" viewing on the PC monitor. Split-window, multiple zoom levels, and delta time display are a few of the capabilities available for analyzing the simulation results in this fashion.

SAMSIM supports both interactive and command file input. Shown in Figure 8 is a sample input stimulus command file for this design. Command files are typically given the design name with extension .CMD. In this example, CIRC.CMD is the name of the command file. The first line specifies the source JEDEC files. Note only the primary file name is given and not the extensions. GROUP CREATE creates a group called CF containing 3 signals (C0-C2). By creating this group, the input pattern for the group can be specified in the PATTERN CREATE CF statement immediately following, rather than having to enter each signal's

stimulus separately. The PATTERN CREATE statement shows the sequential values the given input (or group of inputs) is to take beginning at the start of the simulation and continuing onward. Hex format (as shown) can be used to streamline group pattern entry further. The notation ()*n indicates repeat the enclosed stimulus pattern n times. TRACE CREATE creates a trace buffer file CIRC.TRC into which the state of SAM will be dumped after each simulation step. This information includes internal information such as value on top-of-stack, counter value, etc., as shown in Figure 9. TRACE ON turns the trace process on and may be discontinued with a TRACE OFF command later in the command file. SIMULATE 200 specifies a 200 clock simulation is to be run, and finally VIEW enables interactive viewing of the results of the simulation when complete.

Other useful commands supported by SAMSIM, but not used in our example include (among others):

SET — Modifies values of internal stack, counter, etc.

RADIX — Defines default radix for all SAMSIM input. Options are decimal, binary and hex.

UNASSEMBLE — Converts a micro-word back into its original source code.

Running SAMSIM with the above command file gives the output shown in Figure 10.

In reviewing the simulation output figure, a few words of explanation are required. It is immediately apparent that there are two types of output displayed, two examples of which are CmdAtt and AF. CmdAtt is an example of a single signal waveform, in this case corresponding to a device input. AF correseponds to a group of four signals (note the (4) after the name AF) which includes A0-A3. For AF, the values in the group are displayed in a vertical hex notation each time any signal in the group changes. (If an explicit value is not displayed, it is the same as the previous time step's value). By grouping common signals, much more information can be displayed in a single screen than might otherwise be visisble. In our example, A (AF=A3-A0), B (BF=B3-B0), and I outputs (IL=I2-I0, IM=I5-I3, IH=I8-I6) are viewing groups which have been formed.

The virtual logic analyzer supports commands which allow the order of waveforms to be changed interactively, arbitrary signal groups to be constructed, among others. An on-line HELP command gives instant explanations for all commands. An extremely flexible interactive analysis tool is the result.

The simulation results shown in Figure 11 correspond to the first 40 or so clocks after the graphics controller receives a CmdAtt signalling the beginning of a circle drawing operation. The three RegRd pulses correpond to reading the circles's radius and x-y origin from the parameter register. The single OE pulse two-thirds of the way across the display is the point where the CircPix routine is first entered. It is left as an exercise to the reader to verify the intermediate output values by following the CIRC.ASM source file.

CONCLUSION

The SAM device family provides an efficient solution for sophisticated control problems such as the graphics controller just described. SAM's capability is applicable to a wide range of problems, including industrial control, graphics and disk controllers, programmable sequence generators and the like. The SAM+PLUS tool set makes the design, verification and debug of such designs straightforward. The combination represents a winning approach to control design.

AN11 Rev 2.0

Figure 7. Report File for Circle Drawing Routine

```
This is the Circle Drawing Design


@(#) SAM Version 0.4 4/7/87 18:26:09 36.5

                        EPS448
                    ,-------_-------.
                AO :  1         28 : A1
                NC :  2         27 : A2
                NC :  3         26 : A3
                NC :  4         25 : B0
                C0 :  5         24 : B1
             CLOCK :  6         23 : B2
               VCC :  7         22 : B3
             RESET :  8         21 : GND
                C1 :  9         20 : I2
                C2 : 10         19 : I1
            CmdAtt : 11         18 : I0
              Sign : 12         17 : I5
                I7 : 13         16 : I4
                I6 : 14         15 : I3
                    '---------------'

                        EPS448
                    ,-------_-------.
                NC :  1         28 : NC
                NC :  2         27 : NC
                NC :  3         26 : NC
                NC :  4         25 : I6
                C0 :  5         24 : Rd
             CLOCK :  6         23 : Wr
               VCC :  7         22 : ALE
             RESET :  8         21 : GND
                C1 :  9         20 : ReqRd
                C2 : 10         19 : OE
            CmdAtt : 11         18 : Cn
              Sign : 12         17 : Done
                NC : 13         16 : NC
                NC : 14         15 : NC
                    '---------------'

PART:
     EPS448, EPS448

INPUTS:
     C0, C1, C2, CmdAtt, Sign

OUTPUTS:
     A0, A1, A2, A3, B0, B1, B2, B3, I2, I1, I0, I5, I4, I3, I6, I7

OUTPUTS:
     I6, Rd, Wr, ALE, ReqRd, OE, Cn, Done

PINS:

DEFAULT:
     [00000000000000000111001001

PROGRAM:
     0D:
          [00000000000000000111001001 JUMP WAIT;
     192D:
     WAIT:
          IF CmdAtt * C0' * C1' * C2' THEN
               [00000000000000000111001001 JUMP DOIT;
          ELSE
               [00000000000000000111001001 JUMP WAIT;
     1D:
     DOIT:
          [00010001000000000110011001 JUMP 2D;
     2D:
          [00010001000000000110011001 JUMP 3D;
     3D:
          [00100010000000000111000000] JUMP 4D;
     4D:
          [00100010000000000110011001 JUMP 5D;
     5D:
          [00100010000000000110011001 JUMP 6D;
     6D:
          [00100010000000000111000000] JUMP 7D;
     7D:
          [00110011000000000110011001 JUMP 8D;
     8D:
          [00110011000000000110011001 JUMP 9D;
     9D:
          [01000100011100001111000000] JUMP 10D;
     10D:
          [01000101100000001111000010] JUMP 11D;
     11D:
          [01010101011000011111000000] JUMP 12D;
     12D:
          [01011001100000001111000000] JUMP 13D;
     13D:
          [01010101100000001111000010] JUMP 14D;
     14D:
          [01011000100000011111000000] LOADC 1D GOTO SHIFTR9;
     15D:
     SHIFTR9:
          [10011001100000011111000000] LOOPNZ SHIFTR9 ONZERO 16D;
     16D:
          [10011001100000001111000010] JUMP 17D;
     17D:
          [10011001100000001111000010] JUMP 18D;
     18D:
          [00010110100000011111000000] JUMP 19D;
     19D:
          [01010110001010001111000010] JUMP OUTERLOOP;
     20D:
     OUTERLOOP:
          [01000000100101001111000010] JUMP 193D;
     193D:
          IF Sign THEN
               [00000000000000000011100100] JUMP DrawEnd;
          ELSE
               [00000000000000000011100100] CALL CircPix RETURNTO 21D;
     21D:
          [01010101100000001111000000] JUMP 194D;
     194D:
          IF Sign THEN
               [00000000000000000011100100] JUMP POS;
          ELSE
               [01000110100000011111000000] JUMP 22D;
     22D:
          [01100110100000011111000000] JUMP 23D;
     23D:
          [10000110001000011111000000] JUMP 24D;
     24D:
          [01100101001000011111000000] JUMP IncX;
     25D:
     POS:
          [00010110100000001111000000] JUMP 26D;
     26D:
          [01000110001010011111000010] LOADC 1D GOTO SHIFTR6;
     27D:
     SHIFTR6:
          [01100110100000011111000000] LOOPNZ SHIFTR6 ONZERO 28D;
     28D:
          [10010110001000011111000000] JUMP 29D;
     29D:
          [01100101001000011111000000] JUMP 30D;
     30D:
          [00010001100000101111000000] JUMP IncX;
     31D:
     IncX:
          [01000100100000001111000010] JUMP OUTERLOOP;
     32D:
     DrawEnd:
          [00000000000000000011100100] CALL CircPix RETURNTO 33D;
     33D:
          [00000000000000000011100100] LOADC 16D GOTO DoDone;
     34D:
     DoDone:
          [00010001100000001111000001] LOOPNZ DoDone ONZERO WAIT;
     35D:
     CircPix:
          [01000110100000001111000000] JUMP 36D;
     36D:
          [00011011100000001111000000] CALL TRANS RETURNTO 37D;
     37D:
          [01000110100001001111000010] JUMP 38D;
     38D:
          [00011011100000001111000000] CALL TRANS RETURNTO 39D;
     39D:
          [00010110100000001111000000] JUMP 40D;
     40D:
          [01000101100000001111000000] CALL TRANS RETURNTO 41D;
     41D:
          [01000101100001001111000010] JUMP 42D;
     42D:
          [00010110100000001111000000] CALL TRANS RETURNTO 43D;
     43D:
          [00011011100001001111000010] JUMP 44D;
     44D:
          [01000110100000001111000000] CALL TRANS RETURNTO 45D;
     45D:
          [00010110100001001111000010] JUMP 46D;
     46D:
          [01001011100000001111000000] CALL TRANS RETURNTO 47D;
     47D:
          [01000110100001001111000010] JUMP 48D;
     48D:
          [00011011100001001111000010] CALL TRANS RETURNTO 49D;
     49D:
          [00010110100001001111000010] JUMP 50D;
     50D:
          [01001011100001001111000010] CALL TRANS RETURNTO 51D;
     51D:
          [00000000000000000111001001 RETURN;
     52D:
     TRANS:
          [00111011100100001111000000] JUMP 53D;
     53D:
          [00100110001000011111000000] LOADC 10D GOTO 54D;
     54D:
          [10111100100000001111000000] JUMP MULT1024;
     55D:
     MULT1024:
          [10111011100000011111000000] LOOPNZ MULT1024 ONZERO DONE1024;
     56D:
     DONE1024:
          [11001011001001011111000010] JUMP 57D;
     57D:
          [01101011001000011111000000] JUMP RUNBUS;
     58D:
     RUNBUS:
          [10111011110000001111110000] JUMP 59D;
     59D:
          [10111011110000001111001001 RETURN;

END$

Statistical report:
        Number of label definitions      :   62
        Number of unconditional branches:   59
        Number of conditional branches   :    3
        Number of fatal errors           :    0
        Number of warnings               :    3
        Percent unconditional used       :   30.73%
        Percent conditional used         :    4.69%
```

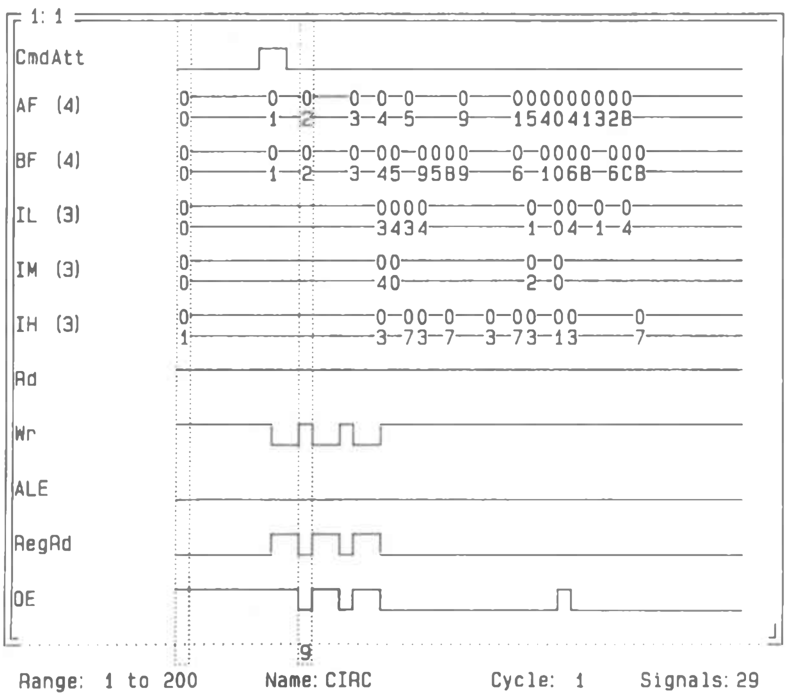
Figure 8. Command File

```
JEDEC CIRC
GROUP CREATE CF = C0 C1 C2
PATTERN CREATE CF = (0H)*200
PATTERN CREATE CmdAtt = (0)*5 (1)*2 (0)*193
PATTERN CREATE Sign =      (0)*200
TRACE CREATE CIRC.TRC
TRACE ON
SIMULATE 200
VIEW
```

Figure 9. Trace File Output

```
Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
MULT1024: 55D: [12288992D] LOOPNZ MULT1024 ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
DONE1024: 56D: [13313506D] CONTINUE ;

Sign=0 CmdAtt=0 C2=0 C1=0 C0=0
57D: [7021024D] CONTINUE ;
```

Figure 10. SAMSIM Interactive Output



FEATURES

- **How to perform multi-way branching.**
- **How to access more than 4-product terms per branch.**

INTRODUCTION

The EPS448/444 SAM device can directly perform a 4-way branch in a single clock cycle. This Application Brief describes how to perform multi-way branching and how to use more than 4 product terms for a branch condition. It assumes a working understanding of the EPS448 architecture and the assembly language or state machine syntax.

BEYOND 4-WAY BRANCHING

When required, the EPS448 can perform multi-way braching in one of three ways.

LINKED BRANCHING

The the first method (Figure 1) is called linked branching and simply involves using 2 clock cycles to perform the branch. On the first clock cycle the machine does a 4-way branch to 4 intermediate states. On the second clock cycle it finishes with a 4-way branch out of each of these intermediate states. The result is up to a 16-way branch in 2 clock cycles. This method can be used with either state machine or assembly language design entry.

In applications running at a low frequency, it may be possible to double SAM's clock frequency, use linked branching, and give the appearance of a multi-way branch in a single system clock.

DISPATCH ROUTINE

The second method of doing an N-way branch is called a dispatch routine. With this method the inputs are pushed onto the stack in the first clock cycle and are used as the next state address in the second clock cycle. The assembly language commands PUSHI or ANDPUSHI are used to push the inputs onto the stack and the RETURN command causes a branch to that address.

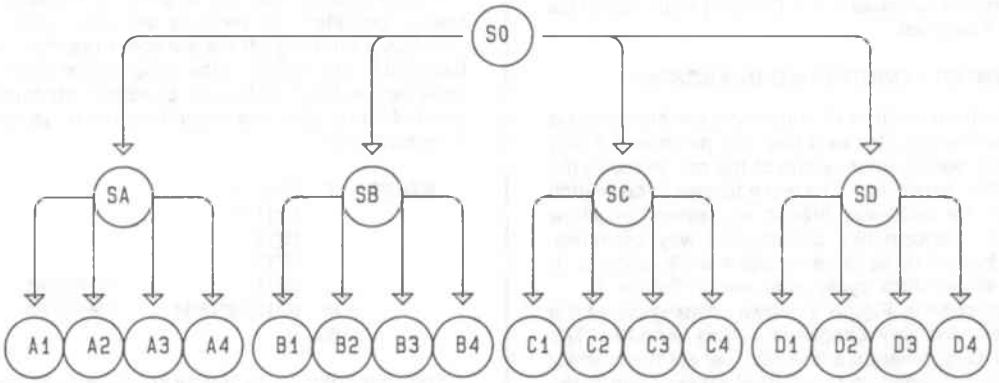
In the sample code shown in Figure 2, the ANDPUSHI command pushes the inputs onto the stack after first masking them with the binary number "00001110B" where input I7 is the MSB and input I0 is the LSB. The result is that an even number between 0 and 14 decimal is pushed onto the stack, matching the number "I3 I2 I1 0" binary.

The RETURN command in Figure 2 causes the top of stack to become the next address. Since this value is an even number between 0 and 14 decimal, an eight-way branch to one of these addresses will be performed.

To complete the branch, the 8 even memory addresses between 0 and 14 must contain the 8 potential next states. Instructions can be placed in a particular memory location by using an absolute label instead of a relative label. Absolute labels must represent a legal and unique memory location

Figure 1. Linked Branching

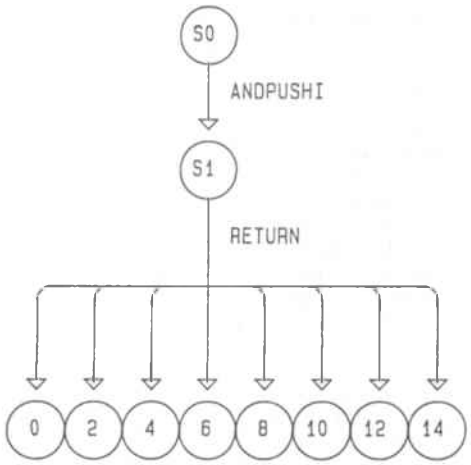
A 16 way branch is obtained in 2 clock cycles by using intermediate states. In the first clock cycle, the machine branches to 1 of 4 intermediate states (SA-SD). On the second clock edge, the machine performs another 4-way branch to 1 of 16 final states (A1-A4, B1-B4, C1-C4, D1-D4).



5

Figure 2. Dispath Routine 8-Way Branch

The dispatch routine above performs an 8-way branch in 2 clock cycles. On the first clock edge, the number represented by the top of stack is used as the next address. Since I3, I2, and I1 can be used to represent an even number between 0 and 14, each of these addresses is a potential next state.




```
S0:  [STATE0] ANDPUSHI 00001110B;
% Push I3, I2, and I1 onto the stack%

S1:  [STATE1] RETURN;
% Jump to the top of stack %

0D:  [OUT0] JUMP NEXT0;
2D:  [OUT2] JUMP NEXT2;
4D:  [OUT4] JUMP NEXT4;
6D:  [OUT6] JUMP NEXT6;
8D:  [OUT8] JUMP NEXT8;
10D:  [OUT10] JUMP NEXT10;
12D:  [OUT12] JUMP NEXT12;
14D:  [OUT14] JUMP NEXT14;
```

within SAM. All addresses and numbers within SAM+PLUS must start with a digit and end with a "H", "D", or "B", for Hexadecimal, Decimal, or Binary.

1) **These are legal absolute labels: 10001B, 0F2H, 44D.**
2) **These would be interpreted as relative labels: F2H, 44D.**
3) **These are illegal labels: 44, 0F2D, 10001H.**

For example, the "4D: [OUT4] JUMP NEXT4" instruction in Figure 2 will be placed in address 4 decimal of the microcode. It will be executed if /I3*I2*/I1 was true at the end of the ANDPUSHI command because input I2 being high translates to a 4 decimal.

COUNTER CONDITIONED BRANCHING

The third method of enhancing the branching is to use the counter as a flag and perform a 2-way branch based on the value of the counter with the LOOPNZ command. The two addresses you branch to can be multi-way branch addresses that allow you to perform two additional 4-way branches. This branching scheme results in an 8-way branch in a single clock cycle as shown in Figure 3.

The code in Figure 3 shows an example of the actual Assembly Language syntax required. The LOADC commands at the S0 label set the counter to a 1 or 0 based on the current value of one of the inputs (I1).

The next instruction, at label S1, is performing the branch with a LOOPNZ command. Based on the value in the counter, the next state will either come from label ABCD or from label EFGH. Since both of these labels are in the multi-way branch block, the actual next instruction depends on the input values. If the counter has a 1, and input condition /I0*I5 is true, then the next command will be JUMP NEXTF.

EXCEEDING 4-PRODUCT TERMS PER CONDITION

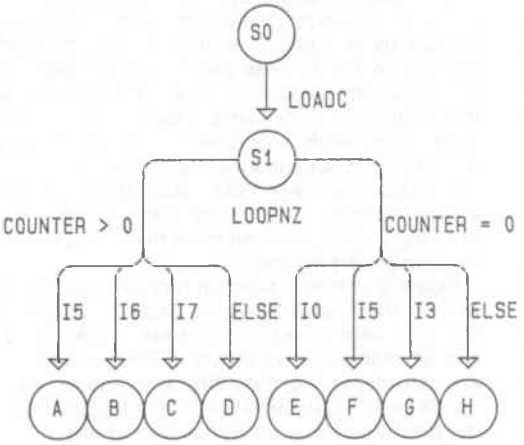
If your design runs out of product terms for a branch condition, or receives the error message "Predicate too long", there are some tips that may help you fit the design. In the sample state machine code below, the first branch condition contains 5 product terms while the second condition contains 1 product term.

```
START:  IF   I5*I0' =
             I5*I1' =
             I5*I2' =
             I5*I3' =
             I5*I4'                THEN S1
        IF   I0*I1*I2*I3*I4        THEN S0
        S2
```

The first solution to this problem is to make a trade off between the number of branches and the

Figure 3. Counter Conditioned 8-Way Branch

An 8-way branch can be performed in a single clock cycle if the counter has previously been set as a flag. In the above code, the LOADC command sets the counter flag based on an input condition (I1). The LOOPNZ command performs a 2-way branch based on the flag to label ABCD or EFGH which are both 4-way branch locations.




```
S0:  IF I1 THEN [OUTS0] LOADC 1D;
     ELSE [OUTS0] LOADC 0D;

S1:  LOOPNZ ABCD ONZERO EFGH;

ABCD: IF I5 THEN [OUTA] JUMP NEXTA;
      ELSEIF I6 THEN [OUTB] JUMP NEXTB;
      ELSEIF I7 THEN [OUTC] JUMP NEXTC;
      ELSE [OUTD] JUMP NEXTD;

EFGH: IF I0 THEN [OUTE] JUMP NEXTE;
      ELSEIF I5 THEN [OUTF] JUMP NEXTF;
      ELSEIF I3 THEN [OUTG] JUMP NEXTG;
      ELSE [OUTH] JUMP NEXTH;
```

number of product terms. The software will automatically partition the first branch so that there are actually two branches to S1 with one branch having 4 product terms and the other having 1 product term. The result is shown below.

```
START:  IF  I5*I0' =
            I5*I1' =
            I5*I2' =
            I5*I3'              THEN S1
        IF  I5*I4'              THEN S1
        IF  I0*I1*I2*I3*I4      THEN S0
        S2
```

The above approach reduces the number of possible branches to 3. In cases where all 4 branches are needed, re-ordering the branches may reduce the number of product terms. Re-ordering may make better use of the built-in prioritization of the EPS448 architecture. Remember that the second branch condition can assume that the first condition has failed.

In the above example you could factor the first condition into "I5*/(I0*I1*I2*I3*I4)" and rearrange the branch order so that the S0 branch is considered first. Once this is done the S1 branch condition can assume that the expression "I0*I1*I2*I3*I4" failed and need not test for it. Thus, the following syntax is equivalent and has 1 product term per condition. A fourth branch could easily be added.

```
START:  IF  I0*I1*I2*I3*I4      THEN S0
        IF  I5                  THEN S1
        S2
```

FEATURES

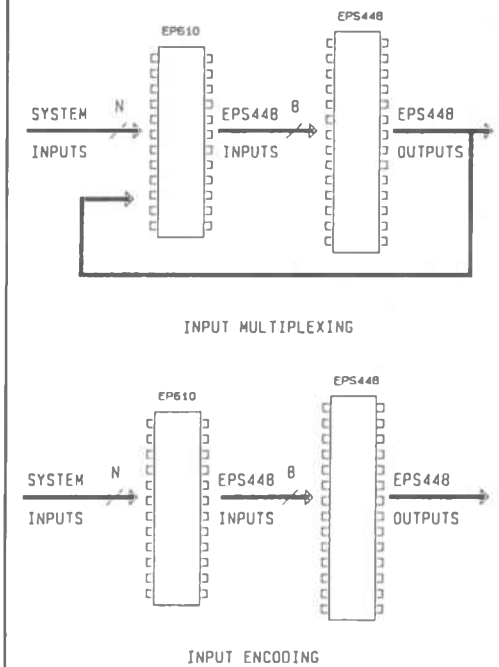
- **Increasing the number of states.**
- **Techniques for Vertically Cascading SAMs.**

INTRODUCTION

The EPS448 provides 8 dedicated inputs. For applications requiring more than 8, it is possible to use external logic to reduce the number of signals, while still retaining the same sequencing function. This Application Brief describes 2 approaches to reduce the number of inputs. It assumes a basic knowledge of the SAM architecture and state machine syntax.

Both input reduction methods, called "input multiplexing" and "input encoding" (Figure 1), use a general purpose EPLD in front of the EPS448 to reduce N inputs down to the 8 allowed. The input multiplexing method reduces the inputs based on the current state of the machine, while the input encoding method does not use the current state.

Figure 1. Input Reduction Alternatives—*There are two alternatives for input reduction. Multiplexing is based on the current state, while encoding is not.*



INPUT MULTIPLEXING

Input multiplexing is the most flexible approach to input reduction. N system signals are transformed by a multiplexer into 8 outputs which connect to the EPS448 inputs and are used for branch control decisions. The EPS448 controls the multiplexer select lines based on the current state. Thus, each state selects the set of inputs required to make the subsequent branch.

Consider the SAM+PLUS state machine code shown in Figure 2. This state machine has a total of 12 inputs (A-L), but no more than 8 are needed for a single 4-way branch.

Figure 3 indicates which inputs are required by each of the branching states. State S1 requires 4 inputs to determine the next state. States S2, S3, and S4 each require 7 inputs. If States S1 and S2 are considered together, they still require only 8 inputs (A-H). State S0 does not require any inputs because it performs an unconditional branch.

The input requirements from the chart divide into 3 sets. Set 1, required by states S1 and S2, consists of inputs A, B, C, D, E, F, G, and H. Set 2, required by state S3, consists of inputs A, B, H, I, J, K, and L. Set 3, required by state S3, consists of inputs A, B, C, D, H, I, and J. The current state determines which set of inputs is routed to the EPS448's inputs.

Figure 4 shows how the input multiplexing is handled. Since inputs A, B, and H, are included in all three of the input sets, they run directly into the EPS448 (I5-I7). The other 9 inputs are routed through an EP610 and multiplexed down to the 5 remaining input pins (I4-I0). Two outputs from the EPS448 (SELECT0 and SELECT1), which depend on the current state, feed the multiplexer select inputs on the EP610 (Figure 5). They direct the correct set of system signals to the EPS448 inputs.

The EP610 multiplexing circuitry is easily entered using Altera's Logicaps schematic capture program and the TTL MacroFunctions. The 9 system signals (C, D, E, F, G, I, J, K, and L) enter from the left and the 5 inputs for SAM (I0-I4) leave on the right.

The resulting truth table for the inputs to the EPS448 is shown in Table 1.

The state machine file for the EPS448 (Figure 6) must contain references that map the 12 system signals to the appropriate EPS448 input pin. The Equations section does this by setting each system signal equal to a pin name. For example, L and D are both equal to input pin I1 because the multiplexer will place one of these 2 signals at pin I1 depending on the current state.

The select lines for the EP610 multiplexer, SELECT0 and SELECT1, are defined as outputs in the States section along with any number of gene-

Figure 2. Input Multiplexing Example—*This state machine requires a total of 12 inputs, but never more than 8 for a single 4-way branch. Input multiplexing performs a 12 to 8 multiplexing to achieve the input reduction.*

```
S0:
    S1

S1:
    IF A */B THEN S1
    IF B */C THEN S2
    IF C */D THEN S3
    S4

S2:
    IF A * E */F THEN S4
    IF B * F */G THEN S1
    IF C * G */H THEN S0
    S2

S3:
    IF H * I * J * K * L THEN S4
    IF A * H * I * J * K THEN S2
    IF A * B * H * I * J THEN S0
    S3

S4:
    IF C * B + /H * J  THEN S0
    IF A * D + /I * K  THEN S2
    IF A + C + H + I   THEN S3
    S4
```

Figure 3. Inputs Required for Each State—*The above chart maps each state against the system inputs required to determine the next state. This input requirements are dived into 3 sets; each set containing less than 8 inputs. One set at a time will be routed to the input pins of the EPS448.*

	STATE	SYSTEM INPUTS										
		A	B	C	D	E	F	G	H	I	J	K	L
SET1	S1	X	X	X	X								
	S2	X	X	X		X	X	X	X				
SET2	S3	X	X						X	X	X	X	X
SET3	S4	X	X	X	X				X	X	X		

Figure 4. Input Multiplexing—*This input reduction method involves multiplexing the many system signals (A-L) down to the 8 input pins of the EPS448 (I0-I7). Inputs A, B, and H run directly into the EPS448. An EP610 takes the remaining 9 inputs and multiplexes them down to I0-I4.*

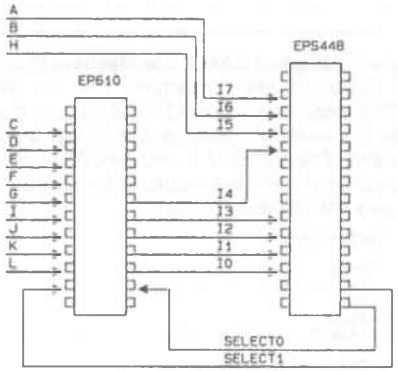
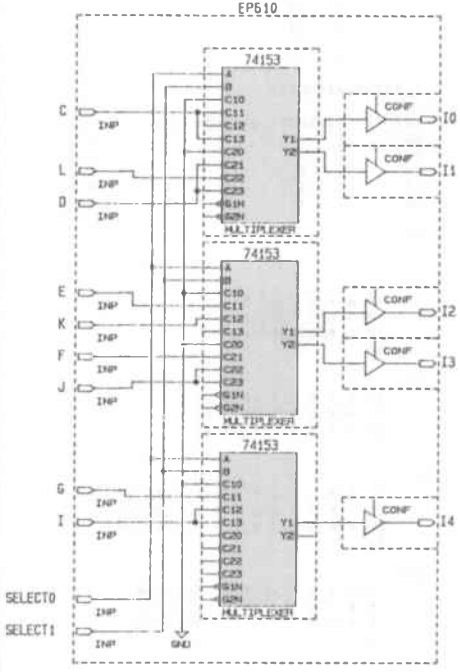


Figure 5. Muliplexing Schematic for the EP610—*The multiplexers within the EP610 are entered using the Logicaps schematic capture package. SELECT0 and SELECT1 come from the SAM device and select the inputs needed for the next transition.*



5

Table 1. Input Multiplexing Truth Table

Select1	Select0	I0	I1	I2	I3	I4	I5	I6	I7
0	0	0	0	0	0	0	H	B	A
0	1	C	D	E	F	G	H	B	A
1	0	0	L	K	J	I	H	B	A
1	1	C	D	0	J	I	H	B	A

Figure 6. Modified SAM State Machine File— *The modified state machine file for the EPS448 includes an EQUATIONS section that maps the system inputs to the actual input pin used. The SELECT 0 and SELECT1 outputs cause the proper inputs to be routed to the pins for the next branch.*

```
PART:EPS448

INPUTS:
  I0, I1, I2, I3, I4, A, B, H

OUTPUTS:
  SELECT1, SELECT0, F02 . . . FXX

% The SELECT lines go to   %
% the EP610.                %

EQUATIONS:

% The equation section is  %
% used to map the system   %
% inputs to the actual SAM %
% input pins.              %

        C = I0;
        D = I1;
        L = I1;
        E = I2;
        K = I2;
        F = I3;
        J = I3;
        G = I4;
        I = I4;

MACHINE:EXAMPLE2

STATES: [SELECT1 SELECT0 F02 ... F15]
S0      [   0       0     X       X ]
S1      [   0       1     X       X ]
S2      [   0       1     X       X ]
S3      [   1       0     X       X ]
S4      [   1       1     X       X ]

S0:
   S1

S1:
    IF A */B THEN S1
    IF B */C THEN S2
    IF C */D THEN S3
    S4

S2:
    IF A * B */F THEN S4
    IF B * F */G THEN S1
    IF C * G */H THEN S0
    S2

S3:
    IF H * I * J * K * L THEN S4
    IF A * H * I * J * K THEN S2
    IF A * B * H * I * J THEN S0
    S3

S4:
    IF C * B + /H * J  THEN S0
    IF A * D + /I * K  THEN S2
    IF A + C + H + I   THEN S3
    S4

END$
```

ral purpose outputs. For example, S3 has outputs SELECT1 = 1 and SELECT0 = 0, which corresponds to input set 2.

INPUT ENCODING

Input encoding, a second method of input reduction, also uses a general-purpose EPLD to map N system signals to the 8 available with the EPS448. Yet, unlike the multiplexing method, input encoding completes the input reduction without using the current state of the machine.

The most common example of input encoding is address decoding. It can be used when a sequencer needs to make a branch decision based on the current value of an address. For example, the EPS448 may need to sit in a state called "IDLE" until an 8-bit address reaches 0F Hexadecimal.

If all 8 address bits go into the EPS448, no other input pins would be left for additional branch conditions. A more efficient approach is to run the 8 address bits into an EP610 which encodes the address down to a single bit notifying the EPS448 when the address equals 0F Hexadecimal. Only one EPS448 input is consumed, leaving 7 input pins for other uses.

Input encoding can be generalized to cover much more than address decoding. Anytime a group of inputs represents a single piece of information, such as a board level command, interrupt lines, or carry-out signals, input encoding should be considered. In particular, if there is an input combination in this group that is considered illegal, or an input combination where several values are don't cares, input encoding is possible.

Consider an example where a system needs to issue a series of commands to the EPS448. The EPS448 then executes a given sequence of states for each command. Assume there are 19 commands in this system and each command is represented by a unique combination of 13 signals labeled A through M. Figure 7 shows a truth table with the 13 signals and the corresponding commands. If A is 0, for example, then the command to the EPS448 is called IDLE, regardless of the other 12 input values.

Since there are only 19 rows in the truth table, 5 bits can uniquely define the current command, instead of the 13 signals shown in the table. These 5 bits (I0, I1, I2, I3, and I4), are the only inputs required by the SAM device to make the branch decisions. With the resulting input encoding, 5

inputs—instead of 13—are required and the design easily fits into the EPS448 with 3 inputs to spare.

An EP610 accomplishes the input encoding as shown in Figure 8. The 13 inputs (A-M) go into the EP610 and are converted to 5 outputs (I0-I4) that connect to the EPS448.

Altera's truth table entry mechanism provided with the state machine package (PLSME) proves to be the most convenient means of entering the EP610 truth table. Figure 9 shows the state machine file that describes the truth table. The NETWORK section is used to indicate that the outputs (I0-I4) are Combinatorial Outputs with No Feedback (CONF). The truth table section of the file lists the 13 inputs (A-M) and the corresponding outputs (I0-I4) for each command. The A+PLUS software automatically processes the state machine file into a JEDEC file (programming code) for the EP610.

To simplify design entry of the SAM device, the EQUATIONS section of the SAM+PLUS design file (.ASM or .SMF) can define each of the commands in terms of the inputs I0-I4. Three sample equations are shown below.

```
ERROR = /I4 */I3 */I2 */I1 * I0;
BACK1 = /I4 */I3 */I2 * I1 * I0;
FORWD1 = /I4 */I3 *I2 * I1 */I0;
```

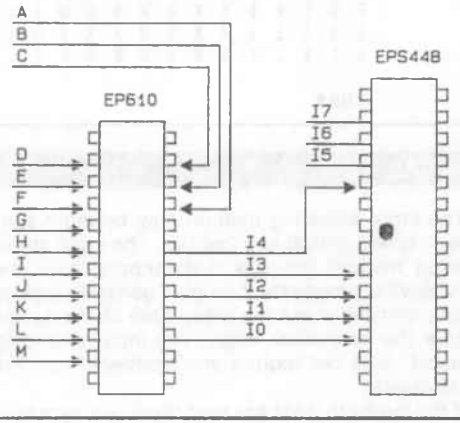
With the above equations entered, the branching within the SAM
device can be expressed in terms of the commands such as:

```
S0: IF ERROR THEN S1      % This is State %
    IF BACK1 THEN S2      % Machine Syntax %
    IF FORWARD1 THEN S3
    S6
```

Figure 7. Input Truth Table—*Input encoding can be used to reduce inputs anytime don't cares (X) appear in an input truth table. In this example, 13 inputs (A-M) are compressed to a 5-bit number (I4-I0).*

A	B	C	D	E	F	G	H	I	J	K	L	M	SYSTEM COMMAND	ROW NUMBER (I4-I0)
0	X	X	X	X	X	X	X	X	X	X	X	X	IDLE	0
1	0	X	X	X	X	X	X	X	X	X	X	X	ERROR	1
1	1	0	X	X	X	X	X	X	X	X	X	X	RESET	2
1	1	1	0	0	0	0	X	X	X	X	X	X	BACK1	3
1	1	1	0	0	0	1	X	X	X	X	X	X	RIGHT1	4
1	1	1	0	0	1	0	X	X	X	X	X	X	LEFT1	5
1	1	1	0	0	1	1	X	X	X	X	X	X	FORWD1	6
1	1	1	0	1	X	X	0	0	X	X	X	X	BACK2	7
1	1	1	0	1	X	X	0	1	X	X	X	X	RIGHT2	8
1	1	1	0	1	X	X	1	0	X	X	X	X	LEFT2	9
1	1	1	0	1	X	X	1	1	X	X	X	X	FORWD2	10
1	1	1	1	0	X	X	X	X	0	0	X	X	BACK3	11
1	1	1	1	0	X	X	X	X	0	1	X	X	RIGHT3	12
1	1	1	1	0	X	X	X	X	1	0	X	X	LEFT3	13
1	1	1	1	0	X	X	X	X	1	1	X	X	FORWD3	14
1	1	1	1	1	X	X	X	X	X	X	0	0	BACK4	15
1	1	1	1	1	X	X	X	X	X	X	0	1	RIGHT4	16
1	1	1	1	1	X	X	X	X	X	X	1	0	LEFT4	17
1	1	1	1	1	X	X	X	X	X	X	1	1	FORWD4	18

Figure 8. Input Reduction through Encoding—*With input encoding, an EP610 compresses 13 system inputs (A-M) down to 5 inputs to the EPS448 (I0-I4). 3 input pins (I5-I7) are still available for general purpose use.*



CAUTIONS WHEN USING TRUTH TABLES

Truth tables used for input encoding must insure that there are no output value conflicts. Such conflicts occur when an input combination satisfies the input conditions for two rows in the table, each of which specifies different output values. Output conflicts are usually introduced by injudicious use of "Don't Care" (X) values in Truth table definition.

Figure 10 shows an instance where "Don't Care"s create contradictory rows. Suppose the input combination (/A*B*C) were applied to the Truth table. This input combination satisfies the first row, which claims OUT1 should go high, as well as the second row, which claims OUT1 should go low. Because OUT1 has conflicting values for this input combination, the Truth table is invalid.

If an invalid Truth table is used in an input encoded SAM design, undefined state transitions may occur. The state machine entry option treats "Don't Care" entries as true "Don't Care"s and will not check that rows are free from output conflicts. Insure that the truth table entered into the PLSME software package is valid and contains only mutually exclusive input conditions, otherwise inadequately defined outputs may cause incorrect state branching.

Figure 9. Truth Table Entry for the EP610—*The truth table entry mechanism for the EP610 eases the entry of the desired truth table. The 12 inputs (A-M) are listed on the left, and the 5 outputs (I0-I4) are listed on the right.*

```
PART:EP610
INPUTS:                          % 13 SYSTEM INPUTS %
A,B,C,D,E,F,G,H,I,J,K,L,M

OUTPUTS:                         % I0-I4 GO TO THE EPS448 %
I0,I1,I2,I3,I4

NETWORK:                         % THR OUTPUTS ARE COMBINATORITAL  %
I0 = CONF(I0,)                   % OUTPUTS WITH NO FEEDBACK (CONF) %
I1 = CONF(I1,)
I2 = CONF(I2,)
I3 = CONF(I3,)
I4 = CONF(I4,)

T_TAB:
% SYSTEM INPUTS          : OUTPUTS     ;SYSTEM COMMAND %

A B C D E F G H I J K L M : I4 I3 I2 I1 I0 ;
0 X X X X X X X X X X X X : 0  0  0  0  0  ; % IDLE %
1 0 X X X X X X X X X X X : 0  0  0  0  1  ; % ERROR %
1 1 0 X X X X X X X X X X : 0  0  0  1  0  ; % RESET %
1 1 1 0 0 0 0 X X X X X X : 0  0  0  1  1  ; % BACK1 %
1 1 1 0 0 0 1 X X X X X X : 0  0  1  0  0  ; % RIGHT1 %
1 1 1 0 0 1 0 X X X X X X : 0  0  1  0  1  ; % LEFT1 %
1 1 1 0 0 1 1 X X X X X X : 0  0  1  1  0  ; % FORWD1 %
1 1 1 0 1 X X 0 0 X X X X : 0  0  1  1  1  ; % BACK2 %
1 1 1 0 1 X X 0 1 X X X X : 0  1  0  0  0  ; % RIGHT2 %
1 1 1 0 1 X X 1 0 X X X X : 0  1  0  0  1  ; % LEFT2 %
1 1 1 0 1 X X 1 1 X X X X : 0  1  0  1  0  ; % FORWD2 %
1 1 1 1 0 X X X X 0 0 X X : 0  1  0  1  1  ; % BACK3 %
1 1 1 1 0 X X X X 0 1 X X : 0  1  1  0  0  ; % RIGHT3 %
1 1 1 1 0 X X X X 1 0 X X : 0  1  1  0  1  ; % LEFT3 %
1 1 1 1 0 X X X X 1 1 X X : 0  1  1  1  0  ; % FORWD3 %
1 1 1 1 1 X X X X X X 0 0 : 0  1  1  1  1  ; % BACK4 %
1 1 1 1 1 X X X X X X 0 1 : 1  0  0  0  0  ; % RIGHT4 %
1 1 1 1 1 X X X X X X 1 0 : 1  0  0  0  1  ; % LEFT4 %
1 1 1 1 1 X X X X X X 1 1 : 1  0  0  1  0  ; % FORWD4 %

END$
```

FINAL TIPS

The input encoding method may be more suitable to speed critical applications. The input multiplexing method requires that outputs leave the SAM device, propagate through a general-purpose EPLD, and still make the setup time of the device before the next clock edge. The input encoding method, does not require any feedback from the SAM device.

If the inputs to SAM are asynchronous, synchronize them by placing registers in front of the inputs. Since input reduction typically places a general-purpose EPLD in front of SAM, synchronization is easily accomplished by using registered instead of combinatorial outputs from the EPLD.

Some applications may need to combine the input encoding and input multiplexing methods. For example, an address decode signal can be used as one of the inputs to the multiplexers in Figure 5. When the proper state arrives, the decode signal is routed to the input of the SAM device.

Figure 10. Invalid Truth Table—*This Truth table is invalid because output values are not uniquely defined over all input conditions.*

A	B	C	OUT1	OUT2
0	X	X	1	0
0	1	X	0	1
X	0	1	1	1

AB66 Rev 1.0

FEATURES

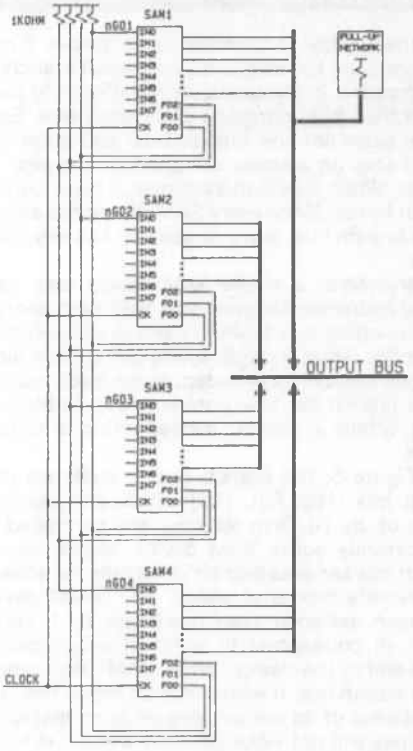
- Increasing the number of states
- Techniques for Vertically Cascading SAMs

INTRODUCTION

When an application requires more states or more microcode depth than a single EPS448 can provide (more than 448 states or words), multiple devices can be cascaded vertically to accommodate the additional states. This Application Brief describes architectural approaches for vertically cascading SAM devices by presenting a few of many ways that control passing can be handled. The mechanism that best suits your needs depends directly on how your sequencer can be most naturally partitioned. The Application Brief assumes an understanding of the SAM device and a working knowledge of the assembly language syntax.

When vertically cascading SAM devices, the outputs of all the devices are tied together forming a tri-state output bus (Figure 1). One device at a time will have control of the output bus, while all other devices are tri-state disabled. The controlling device performs sequencing until it is time to branch to a sequence found in another EPS448.

Figure 1. Vertically Cascaded SAMs—*When vertically cascading SAM devices, the outputs are tied together. While one device has control of the output bus, the others are tri-state disabled. Each device has one input (e.g. nGO1) that enables it onto the bus.*

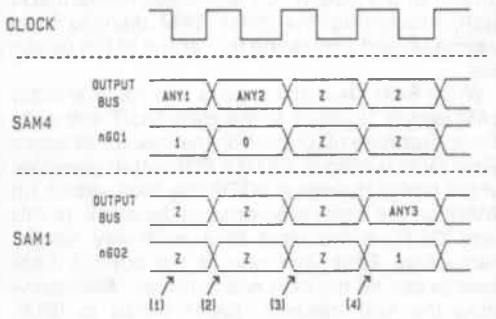


SIMPLE VERTICAL CASCADING

The simplest method of vertically cascading SAM devices is to use one input pin to signal a device to take control of the output bus (Figure 1). A control line (e.g. nGO1) to each device is pulled up through a 1K Ohm resistor. When the active EPS448 wants to pass control to another device, it pulls down the appropriate nGO signal, and jumps to an idle state where its outputs are tri-stated. This configuration essentially allows conditional JUMP instructions between SAM devices.

Figure 2 shows the timing associated with this simple vertical cascading. Initially SAM4 has control of the output bus and SAM1 is idle. SAM1's outputs are tristated during this period. To pass control, SAM4 brings the nGO1 signal low to activate SAM1. On the next clock cycle, called the transition clock period, both SAM1's and SAM4's outputs tristate. SAM1 becomes active, thereby taking control of the output bus. SAM4 becomes idle on the following clock.

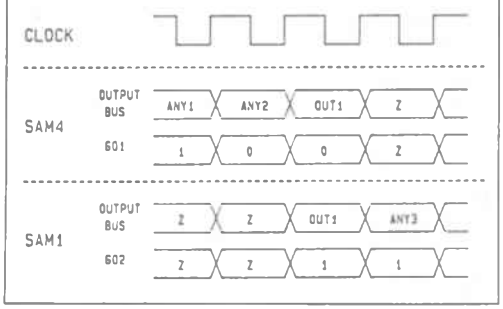
Figure 2. Passing Control Between Devices—*To pass control between 2 devices (SAM4 and SAM1). SAM4 starts active(1), then brings nGO1 low (2) to pass control. On the next clock period, both devices are disabled (3) before SAM1 takes control of the output bus (4).*



5

Notice that both devices are tri-state disabled during the transition clock period. This idle period prevents potential glitches during transitions from high-Z to a valid output or from a valid output to high-Z, from causing bus contention with the other devices. During this clock period, the control lines (nGO1-4) float unless pulled up by the resistors shown.

Figure 3. Alternative for Control Passing—*To avoid the clock cycle where the output bus is left floating, both devices drive the same value onto the bus for a single clock cycle.*



In an alternative method, shown in Figure 3, both devices drive the same value (OUT1) on the output bus for the transition clock period. Though removing the need for pull up resistors, this method may introduce temporary bus contention, which may cause a current surge into the SAM devices, and hence induce noise or ground bounce elsewhere on the board. Many systems design procedures prohibit contention due to potential emmission problems. The bus contention would not functionally impair SAM device operation.

Each of the SAM devices must have its own source file (.ASM or .SMF). A segment of the code used in the first device, SAM1, is shown in Figure 4. While the machine is inactive it remains in the state called IDLE with the outputs disabled. When it receives an active low nGO1 signal (pin IN0), it wakes up, leaves its outputs high-Z, and JUMPs to START. When it gets to the label START it takes control of the output bus and holds nGO2-nGO3 high, preventing the other SAM devices from waking up and competing for control of the output bus.

When SAM1 is ready to pass control to another SAM device, it jumps to the state QUIT and pulls the appropriate nGO control line low; for example when GO2 is output, SAM1's F00 output goes low, which brings the signal nGO2 low, thus waking up SAM2 on the next rising edge of the clock. In this case QUIT is the label of a multi-way branch instruction. Thus, any one of the other 3 SAM devices can be the next one activated. After activating the next machine, SAM1 jumps to IDLE, where it waits to be given control again.

Figure 4. Vertical Cascading Assembly Code—*The ASM syntax above is for the SAM1 device. SAM1 will sit in the IDLE state until it sees the nGO1 signal from pin IN0 go low. Then it will jump to START and take control of the output bus. When SAM1 is ready to give up control of the outputs, it jumps to QUIT where it brings one of three nGO lines low to activate the next machine. Finally, it jumps back to the IDLE state.*

```
MACROS:   %F F F F F F              F %
          %0 1 2 3 4 5 .......      15%
   HOLD = "1 1 1"
   GO2  = "0 1 1"
   GO3  = "1 0 1"
   GO4  = "1 1 0"
   OUT1 =        "1 0 1 ...........0"  % INSERT DESIRED     %
   ANY3 =        "1 0 0 ...........1"  % OUTPUT VALUES     %
   ANY2 =        "0 0 1 ...........1"  % FOR THESE STATES  %

PROGRAM:

IDLE: IF /IN0 THEN [Z] JUMP START;
      ELSE [Z] JUMP IDLE;

START: [HOLD ANY3] CONTINUE;

                .
                .
                .

QUIT: IF IN3 & IN4 THEN [GO2 ANY2] JUMP IDLE;
      ELSEIF IN3   THEN [GO3 ANY2] JUMP IDLE;
      ELSE  [GO4 ANY2] JUMP IDLE;
```

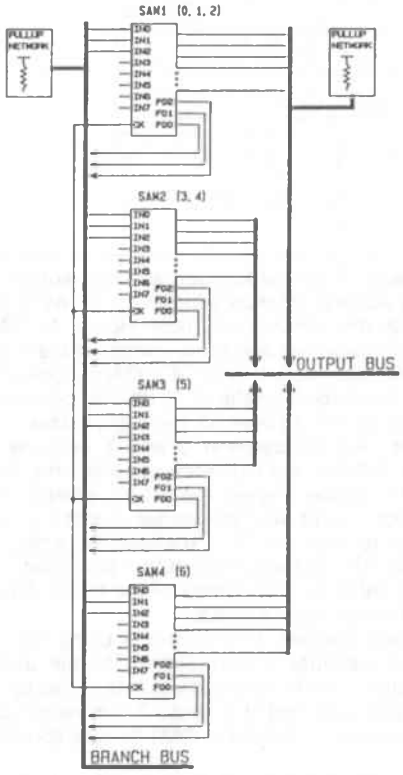
BRANCH BUS

As the number of SAM devices increases, it may be convenient to move to an addressed branching configuration. In this configuration (Figure 5) there is a branch bus alongside the output bus. Each device examines the branch bus, and stays idle until it sees an address assigned to it appear on the bus. When it sees this address, it takes control of both buses. Since every SAM device has access to the branch bus, every device can call any other device.

Furthermore, a single SAM device may have several addresses assigned to it, with each address corresponding to a different starting location in its microcode. Thus, a single device can contain more than one block of independent states. Each address on the branch bus then corresponds to a block of states within a device, instead of to a unique device.

In Figure 5, the branch bus is made up of 3 output bits [F00, F01, F02] which are passively pulled up by 1K Ohm resistors and controlled by the currently active SAM device. Values on the branch bus serve as branch addresses for accessing discrete blocks of states. The SAM1 device has been assigned branch address 0, 1, and 2 which all correspond to different sequences of states within the device. While SAM1 has control of the output bus, it will drive the branch bus with the address of its current sequence so that other machines will not wake up. Only when it is ready

Figure 5. Addressed Branch Cascading—*In addressed cascading, each SAM device wakes up when it sees one of its assigned addresses on the branch bus. Each SAM can recognize more than one address. For example, SAM1 will take control if it sees addresses 0, 1, or 2 on the branch bus.*



to pass control will it drive a new value onto the branch bus. For example, when SAM1 outputs the branch address 3, the SAM2 device will assume control.

Figure 6 shows a portion of the ASM code required for SAM1. It is similar to that for simple cascading. While SAM1 is sitting IDLE, it must look for any of its three possible branch addresses (BA0, BA1, BA2) to come valid. If one of these values appear on the branch bus, it jumps to the start of the corresponding sequence (START0, START1, and START2). If it sees none of these, it will stay IDLE.

When SAM1 is ready to give up control, it jumps to the label QUIT where it decides which address to call next. Each possible branch calls a different routine by specifying a different value on the branch bus. Notice that the GOx macros correspond to binary branch addresses (i.e. GO3 = "011").

Figure 6. Addresed Cascading Code—*The addressed cascading above uses 3 input lines (IN0-IN2) to address different routines within a single device. By specifying different addresses at the QUIT label, any routine in any other device can be called.*

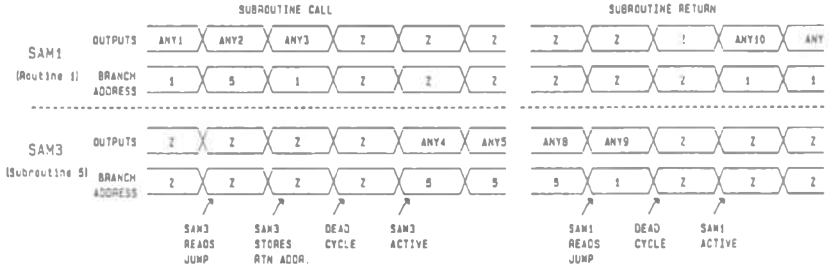
```
MACROS:         %F F F F ..... F %
                %0 1 2 3        15%

          G00 = "0 0 0"
          G01 = "0 0 1"
          G03 = "0 1 0"
          G03 = "0 1 1"
          G04 = "1 0 0"
          G05 = "1 0 1"
          G07 = "1 1 1"
         ANY2 =        "1 ......0"
         ANY3 =        "0 ......1"

EQUATIONS:
  %INPUT ADDRESS DECODES%
          BA0 = /IN2*/IN1*/IN0;     %BRANCH ADDRESS 0%
          BA1 = /IN2*/IN1*IN0 ;     %BRANCH ADDRESS 1%
          BA2 = /IN2*IN1*/IN0 ;     %BRANCH ADDRESS 2%

PROGRAM:

IDLE: IF BA0  THEN [Z] JUMP START0;
    ELSEIF BA1 THEN [Z] JUMP START1;
    ELSEIF BA2 THEN [Z] JUMP START2;
    ELSE [Z] JUMP IDLE;


START0: [G00 ANY3] CONTINUE;

START1: [G01 ANY3] CONTINUE;

START2: [G02 ANY3] CONTINUE;
                .
                .
                .


QUIT: IF IN5 * IN6 THEN [G03 ANY2] JUMP IDLE;
      ELSEIF  IN5  THEN [G04 ANY2] JUMP IDLE;
      ELSEIF  IN7  THEN [G05 ANY2] JUMP IDLE;
      ELSE [G07 ANY2] JUMP IDLE;
```

VERTICAL SUBROUTINE CALLS

The configurations discussed so far have only allowed a JUMP function between SAM devices. It is possible, however, to perform subroutine calls between devices. Subroutines are most easily implemented using the branch bus configuration of Figure 5. The difference between a subroutine call and a jump is that a subroutine must eventually jump back to the machine that called it. This feature requires the subroutine to recall the return address of the calling device. The return address can be stored on the called SAM's internal stack.

The timing involved in this control pass is shown in Figure 7. Before the subroutine call, SAM1 is in control of the bus and is executing the block of states starting with 1 (branch address 1). It drives the branch bus with the value for 1 until it is ready to call a subroutine (5 - branch address 5) within another device (SAM3). To start the subroutine call, SAM1 specifies the address of the subroutine (5) on the branch bus, which wakes up the SAM3 device. SAM1 then applies the return address (1) so that SAM3 can store the return value on its internal stack. SAM1 tristates both the branch and output buses on the next clock cycle. A "dead"

Figure 7. Timing of Vertically Cascaded Subroutines—*In the above example of a cascaded subroutine call, SAM1 calls subroutine number 5 from SAM3 by placing 5 on the branch bus. Next SAM1 places its own address (1) on the branch bus to tell the subroutine where to return to. Finally, SAM1 tristates, and on the next clock period, SAM3 takes control of the output bus. When SAM3 (address 5) is ready to return, it puts the return value (1) on the branch bus and then idles.*



clock cycle follows, after which SAM3 takes control of the output bus.

The syntax required for the SAM3 device is shown in Figure 8. Notice that if SAM3 reads its address, it still leaves its outputs tri-stated for two more clock cycles. The ANDPUSHI opcode is used to store the return address in the stack by masking off inputs IN3-IN7. (See the Application Brief on "Multiway Branching" for information on using ANDPUSHI with dispatch routines).

Figure 8. Vertically Cascaded Subroutine Code— *this example, SAM3 is used as a subroutine at the branch address 5. After it is called, it stores the return address on the stack with the ANDPUSHI command. The RETURN command jumps to an absolute address which puts the correct return address onto the branch bus.*

```
MACROS:
      %OUTPUT VALUES FOR BRANCH BUS%
        000 = "000"   001 = "001"   002 = "010"
        003 = "011"   004 = "100"   005 = "101"
        006 = "110"   007 = "111"

      %OUTPUT BUS VALUES%
        ANY3 = "10...1", ANY8 = "00...1", ANY9 ="01...0"

EQUATIONS:
      %BRANCH ADDRESS INPUTS%
        BA5 = IN2#/IN1#IN0;     %BRANCH ADDRESS 5%

PROGRAM:

%IDLE OR START THE SUBROUTINE %
IDLE: IF BA5 THEN [Z] ANDPUSHI 7H GOTO START5;
      ELSE [Z] JUMP IDLE;


START5: [Z] CONTINUE;              % DEAD CYCLE %
        [005 ANY3] CONTINUE;       % TAKE CONTROL %
                                   % OF OUTPUT BUS %

              .
              .
              .

QUIT5: [005 ANY8] RETURN;          %SUBROUTINE RETURN %

% JUMP TABLE (RESERVED LOCATIONS)%
0D:[000 ANY9] JUMP IDLE;    % ALSO THE POWER-UP STATE %
1D:[001 ANY9] JUMP IDLE;
2D:[002 ANY9] JUMP IDLE;
3D:[003 ANY9] JUMP IDLE;
4D:[004 ANY9] JUMP IDLE;
5D:[005 ANY9] JUMP IDLE;
6D:[006 ANY9] JUMP IDLE;
7D:[007 ANY9] JUMP IDLE;    % THIS IS AN ERROR CONDITION %
```

To return from the subroutine, SAM3 applies the return address (branch address 1) on the branch bus for one clock cycle (see Figure 7). SAM1 reads this address and takes control of the output bus (becomes active). The RETURN opcode at the QUIT label accomplishes this task by performing a jump to the address at the top-of-stack. This address will be between 0 and 7 because the inputs IN3-IN7 were masked off when the ANDPUSHI opcode pushed the return address onto the stack. In this example the top of stack is 1 and a jump to address 1D (1 decimal) is performed. Address 1D has the correct output specification to signal SAM1 to take control of the buses. Finally, SAM3 jumps back to IDLE.

Branch address 7 is unusable using this approach, because it corresponds to the default condition of all branch address bits pulled up. An error has occurred if address 7D is reached. In this instance all cascaded SAM devices should be reset.

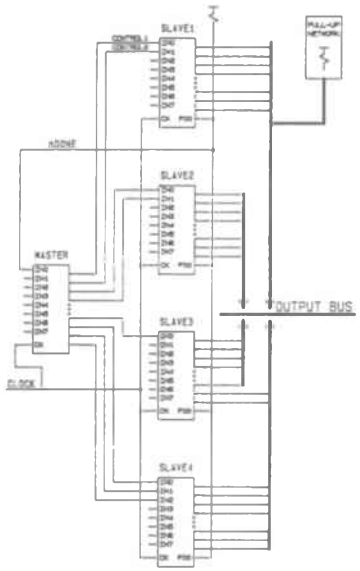
During power-up, the device will start in state 0D. Using the above approach, all machines will correctly jump to their idle state on the succeeding clock cycle. The SAM device that initiates control after power-up should jump to its correct starting state.

After making the subroutine call, SAM1 must go into a high-Z state and wait for the routine to finish. It can do this in one of two ways: it could return to its actual IDLE state, which is its starting state, or it could stay in a separate loop. In the first case, when SAM1 regains control (by recognizing its address), it will effectively be restarting. In the second case, SAM1 will return to where it was when the call was made.

MASTER/SLAVE CASCADING

Another vertical cascading method is the Master/Slave configuration (Figure 9), in which a single SAM device (the master) controls the overall

Figure 9. Master/Slave Cascading—*In a Master/Slave configuration one SAM device (the Master) has control of the sequencing. It calls routines from any number of slaves. The slave signify that they are finished by pulling the nDONE line low.*

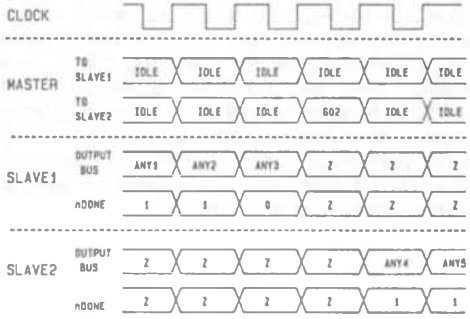


sequencing by calling routines within other SAM devices (the slaves). The slaves are cascaded together with all but one of the outputs connected to the output bus. The single output (F00) called nDone in Figure 9, is passively pulled up through the resistor shown. When an active Slave device has finished executing its routine, it pulls the nDONE line low. This action alerts the Master that it should jump to the next routine.

Any number of control lines can run from the master to a given slave depending on the number of routines within the slave: for n control lines the slave can have 2^n-1 routines. The case where all control lines are high indicates to the slave that it should be idle. In this example, SLAVE1 has 2 inputs which allow it to contain up to three routines. When both CONTROL0 and CONTROL1 are high, SLAVE1 is idle. There is no limit to the number of slaves that can be added to this system. Furthermore, unlike the branch bus approach, as more routines are added, the number of inputs to the other devices does not increase.

The timing associated with the master-slave configuration is shown in Figure 10. SLAVE1 originally has control of the output bus. It signifies that it is done by pulling nDONE low. The master responds by initiating the SLAVE2 machine which then takes over the output bus. Notice that there is still a single clock cycle where the output bus is undefined.

Figure 10. Timing of Master/Slave Configuration—*In the above example of a Master/Slave cascaded configuration, SLAVE1 starts with control of the output bus. It signifies it is done by bringing nDONE low. The Master then initiates SLAVE2 by sending it an active starting address.*



FINAL TIPS ON VERTICAL CASCADING

Each technique for vertically cascading SAM devices consumes output and input pins from each EPS448 to accommodate control passing. If needed, additional outputs can be created by horizontally cascading devices as described in the software manual (SAM+ Reference Guide P. 2-48). If additional inputs are needed, they can be generated by multiplexing several signals down to the 8 inputs available with the EPS448 as described in "Increasing Inputs to SAM".

Partitioning is the most important step of creating a vertically cascaded sequencer. Partitioning is the process of dividing the code or state segments among the multiple SAM devices. Most large sequential applications have natural divisions that help dictate where partition borders should fall. The general goal in partitioning is to create blocks of sequences that fit within a single EPS448 device, and to minimize control transfers from one device to another.

All clock pins should be tied together; all reset pins should be tied together. Inputs to different SAM devices need not be tied together, allowing different sections or routines of the overall sequencer to branch based on different input signals.

When doing any vertical cascading, the output lines from the resulting sequencer come off of a tri-state bus (output bus). Do not connect output bus signals to system clock or latch enable inputs. As in any tri-statable bus, the output bus of cascaded EPS448 devices is not guaranteed to be glitch-free during transitions from high-Z to output valid or from output valid to high-Z.

Insure that only one device becomes active on power-up. All other devices should jump to IDLE from power-up and thus should have the following line in their source file: "0D: [Z] JUMP IDLE;".

If every device goes IDLE without first calling another device (or bringing nDONE low), the system will be deadlocked in an idle state. This is easily avoided with proper coding, or could be prevented autonomously with a watch-dog timer implemented in one of the SAM devices. The watchdog timer counts up toward a final count value that corresponds to a fixed period of time. During normal operation, the other SAM devices will periodically clear the watchdog counter, preventing it from reaching its final count value. If the watchdog reaches its final count value, it indicates a system error has occurred, since it wasn't reset in the prescribed period of time. The watchdog timer should then reset the system or generate appropriate system error status.

AB65 Rev 1.0

BUS INTERFACE INTEGRATION APPLICATIONS **PAGE NO.**

6

FEATURES

- **Familiar control signals.**
- **Easily implemented.**
- **Simulation considerations.**

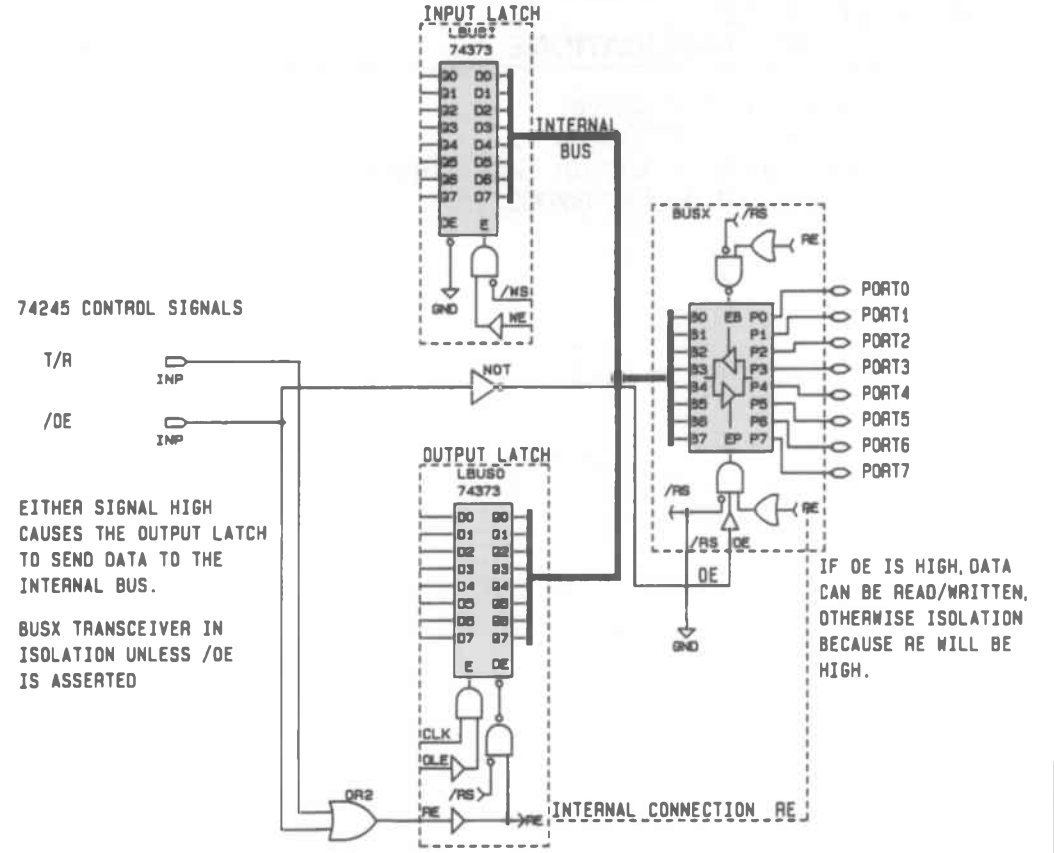
INTRODUCTION

The 74245 Octal Bus Transceiver is one of the most popular ways to buffer devices attached to a bus. The EPB1400 is designed to interface directly to most bus structures, having its own internal bus transceiver capable of 24mA drive. This does not mean that one has to give up the familiarity of the 245's control signals. Due to the flexibility of the EPB1400 architecture and dedicated control macrocells, it is a simple task to emulate the 245's function with the EPB1400.

CONTROL SIGNALS

The 245 has two control signals: Output Enable ($\overline{OE}$) and Transmit/Receive (T/R).The control signals for the Bus Transceiver in the EPB1400 (BUSX) are the Port Read Enable (RE), the Read Strobe signal ($\overline{RS}$), and the Output Enable signal (OE). Since the 74245 does not utilize a data strobe scheme, we will not utilize the $\overline{RS}$ signal, which is used by the EPB1400 as a fast pin strobing option.

Figure 1.
This schematic shows the EPB1400 configured so the Microprocessor Interface section emulates a 74245.



Note that the RE signal of the EPB1400's BUSX Bus Transceiver is derived by the logical 'OR' of the RE signals for the two LBUSO latches. Therefore, we must consider these latches.

Table 1. Function Table Comparison.

74245			EPB1400 PINS		
/OE	T/R	OPERATION	/OE	REp	OPERATION
0	0	B TO A	0	0	PORT TO IB
0	1	A TO B	0	1	IB TO PORT
1	X	ISOLATION	1	X	ISOLATION

Examining the truth table for the BUSX Bus Transceiver in the EPB1400 data sheet, we see that there are four possible input combinations (the RS signal defaults to ground) that yield three possible states. The truth table for the 245 shows that when its OE signal is high, the device must be in the high impedance state, regardless of the level of the T/R signal. In this state, the OE signal of the Bus Transceiver (BUSX) must be low and the RE signal must be high at all times, regardless of the state of the T/R signal. Refering to the truth table shown in Table 1, we can then emulate a 74245 in schematic primitive logic symbols using the LogiCaps schematic capture software. Refer to Figure 1 for exact implementation.

SIMULATION RESULTS

By adding some logic and inputs to make a complete design, we can then test the design by using the Altera Functional Simulator. Following the simulation methodology outlined in the Simulation section of this Handbook to develop input vectors and command files, we get the the anticipated results shown in Figure 4. Some of the signals may only be viewed by calling out buried node extensions or special names reserved for the EPB1400. For example, in order to observe the signals of the internal bus, we must use .B0 to .B7. The inputs to the BUSX macro must have the

Figure 2. Simulator Command File.

```
PLOT ON;
GROUP HEXADECIMAL BUS = .B0 .B1 .B2 .B3 .B4 .B5 .B6 .B7;
GROUP HEXADECIMAL INPUT = I0 I1 I2 I3 I4 I5 I6 I7;
GROUP HEXADECIMAL PORT = PORT0 PORT1 PORT2 PORT3 PORT4
PORT5 PORT6 PORT7;
VECTOR @245;
INIT PORT0.OE = 1;
INIT .OL24.RE = 0;
PLOT CLK /OE TR .OL24.RE PORT0.OE PORT0 PORT1;
SIM 30;
VIEW;
QUIT;
```

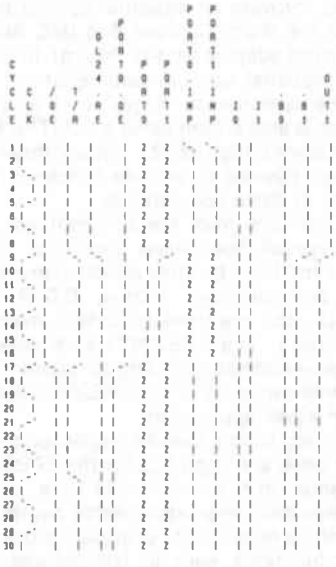
Figure 3. Input Vector File.

```
PATTERN:

I0 = (1)*;
I1 = (0)*;
I2 = (1)*;
I3 = (1)*;
I4 = (1)*;
I5 = (1)*;
I6 = (1)*;
I7 = (1)*;

CLK = (0011)*;
/OE = (00)*8 (11)*8;
TR = (00)*4 (11)*4 (00)*4 (11)*4;

PORT0 = (11)*4 (ZZ)*4 (1)*;
PORT1 = (11)*4 (ZZ)*4 (1)*;
PORT2 = (11)*4 (ZZ)*4 (1)*;
PORT3 = (11)*4 (ZZ)*4 (1)*;
PORT4 = (11)*4 (ZZ)*4 (1)*;
PORT5 = (11)*4 (ZZ)*4 (1)*;
PORT6 = (11)*4 (ZZ)*4 (1)*;
PORT7 = (11)*4 (ZZ)*4 (1)*;
```

extension .INP, and to see the Output Enable signal of the BUSX, use the extension .OE. For further information on simulating the EPB1400, please refer to the Simulation section of this Handbook, which describes simulation techniques for this device in detail.

Figure 4.

The Waveform File created by the Functional Simulator shows that the logic is correct and the design functions as a 74245.



AB58 Rev 2.0

PS/2 MICRO CHANNEL ADD-ON CARD INTERFACING WITH THE EPB2001 & EPB2002

INTRODUCTION

The IBM Personal System/2 (PS/2) architecture announced in early 1987 included a new add-on card bus architecture called the Micro Channel Bus. This architecture, which supports both 16- and 32-bit processors, includes the concepts of high-performance multi-device bus arbitration, level-sensitive interrupts, and Programmable Option Select (POS) registers. The POS registers are intended to replace DIP switches and jumpers on the add-on card, and allow software-configuration of card features. Simultaneously, in order to reduce the overall size of the PS/2 system, the physical size of add-on cards has been reduced by some 40% to only 33 square inches. The net result is a higher-performance, more reliable scheme for the attachment of add-on cards to the PS/2. The cost of these benefits, however, is a more complex interface design problem for the add-on card designer requiring a highly-integrated VLSI approach.

This Application Note illustrates the use of Altera's user-configurable adapter interface chips for the IBM Personal System/2 (PS/2) Micro Channel. A typical application is described, consisting of a Multi-Function card including 32K words of static CMOS RAM, a general-purpose output port and two serial ports. The Micro Channel interface EPLDs, the EPB2001 and EPB2002, provide all essential control interfaces between the Micro Channel Bus (MC Bus) and a PS/2 feature adapter (add-on board). Implementation of optional add-on card features, such as wait-state generation, output port and bit-rate prescaler, is also shown using a BUSTER EPB1400 EPLD. User-configurability allows these highly-integrated devices to provide functions typically requiring multiple components.

POS I/O pins from the EPB2001 are used to control typical board-level functions: wait-state duration and DMA channel selection and enabling. Board control lines (-DEN, DT/-R, -IOWR, -MEMRD, etc.) are used to control memory and I/O accesses. The EPB2001's chip select block provides all address decoding for board functions. The overall versatility of the EPB2001/2002 is illustrated in a real application.

In addition to this specific application example, a set of general design tips for the EPB2001/2002 is presented at the end of this Note. These tips show ways that the programmable nature of these chips can be exploited to provide optional, board-specific functions, such as latched addresses or card data-size feedback. 32-bit system design considerations are also discussed.

This Application Note assumes the reader is familiar with the basic concepts of the IBM MC Bus. The reader is directed to the references listed at the end of this Note for further information on the basic characteristics of the Micro Channel. Comprehensive functional and parametric information on the EPB2001/EPB2002 may be found in the EPB2001/EPB2002 Data Sheet, available from Altera.

MULTI-FUNCTION CARD APPLICATION OVERVIEW

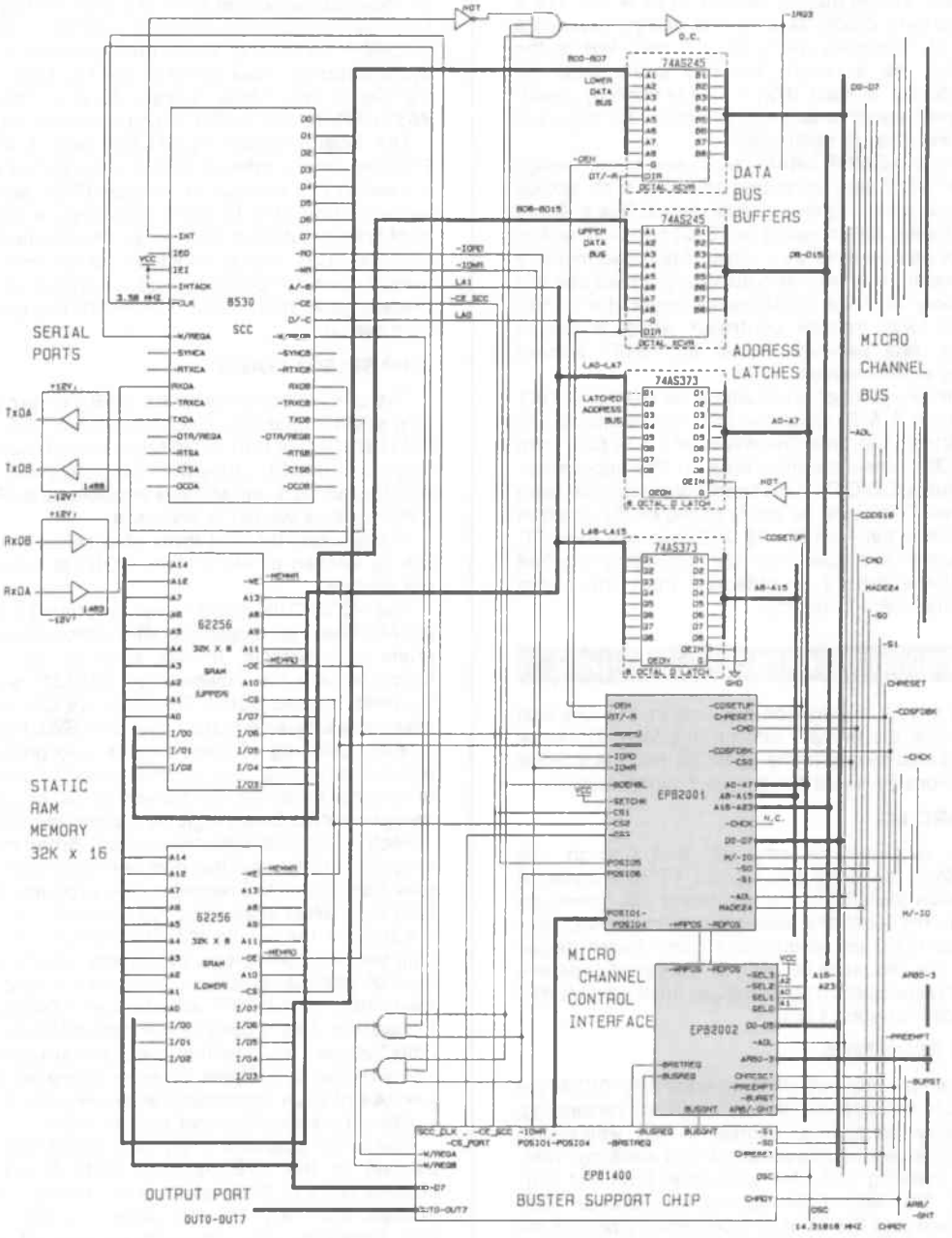
The block diagram in Figure 1 shows the overall structure of the Multi-Function card design. Shown on the right are the MC Bus lines which run to all add-on card slots. The EPB 2001 provides mandatory POS register functions required by the specification in any add-on card interface. In addition, board control logic, board I.D. storage and address decoding are integrated on the chip. The EPB2002, shown at lower right, provides DMA arbitration support. Not all add-on designs require DMA, but where it is needed, the EPB 2002 fully supports the IBM bus exchange protocols.

This board design appears to the PS/2 as either an 8-bit I/O peripheral (for accesses to the Z8530 Serial Communications Controller (SCC)), or as a 16-bit memory extension (32K words). The two 74AS245 transceivers shown at upper right buffer data transfers between the Micro Channel and I/O or memory. Memory will always be accessed as words (no byte read/write), and as a result, selective enabling of the buffers using -SBHE from the Micro Channel is not required. Two 74AS373 flow-through latches are used to latch addresses during bus transfers. The -ADL line from the MC Bus controls the latching. Only the 16 low-order address bits need be latched since memory capacity is 64K (2**16). The EPB 2001's chip select logic will handle upper-order address decoding for the various I/O and memory chips on the board.

The Z8530 SCC provides two serial channels in a single 40 pin device. This chip supports a variety of synchronous and asynchronous communication modes, on-chip data buffering, and an integral baud-rate generator. The Z8530 includes handshake lines for DMA Request/Acknowledge which interface to the EPB2002. The EPB2002 requests use of the MC Bus via the -PREEMPT line in response to these requests.

The clock for the SCC is provided by a divide-by-four counter, which is driven by the OSC line on the MC Bus. This clock divider is implemented in a small portion of an EPB1400. The OSC line provides a precision clock input of frequency 14.31818 Megahertz. The clock divider generates a

Figure 1. Multi-Function Card Block Diagram



3.58 MegaHertz output for the SCC. As a result, a local oscillator is not needed for the SCC.

Wait-state logic associated with the Z8530 is also designed into the EPB1400. In a Micro Channel default bus cycle, the width of the -CMD bus transfer strobe may be as narrow as 90 nS. The 4 MegaHertz Z8530 used in this design requires a 250 nS minimum width on -RD and -WR to the device. As a result, the bus cycle must be "stretched" at least 160nS. This is done by inserting wait-states to extend the cycle. This logic will be described in detail later.

The two CMOS SRAM chips used in this design have 100 nS address access times. As such, access to these chips requires no wait-states. This memory will always be accessed as 16-bit words. The 32K block can be used as a general-purpose memory extension. A more elaborate design could use this memory as a transmit/receive buffer if an additional local transfer controller where added to move data between buffer and SCC without external intervention.

The two pairs of serial data lines (RxD and TxD, Channel A & B) from the SCC are connected to 1488/1489 line driver/receivers which in turn form RS-232-compatible interfaces. In this application, the full DCD/CTS/RTS handshake has not been shown. This could be easily added by the addition of buffers between the RS-232 links and the SCC modem control pins. The +12V/-12V supply required by these drivers is obtained from the Micro Channel edge connector.

EPB2001 INTERFACE FUNCTIONS

In this Multi-Function application, the EPB2001 provides the primary control interface. The main functional blocks in the EPB2001 and the specific functions provided for this application include:

BOARD I.D.

As required for any MC Bus add-on, the EPB2001 provides two CMOS EPROM bytes at location 0100-0101H for the board I.D. These are read-only locations accessible from the MC Bus. These I.D.'s are unique to a given board design and are allocated by IBM to registered Independent Developers. The developer must contact IBM directly for such I.D.'s.

POS REGISTERS

The four required POS registers 0102-0105H are used in this application to control I/O remapping, memory remapping, number of I/O wait-states, DMA request source selection and masking, interrupt masking and DMA arbitration level and Fairness. The latter DMA control functions are also mapped into the satellite POS register bits on the EPB2002. The bit-mapping of these functions is illustrated in Figure 2.

POS bits which are required outside the EPB2001 are brought-out via the POS I/O pins. This mapping is shown in Figure 2.

BOARD CONTROL

The board control logic on the EPB2001 provides all required transceiver control (-DEN, DT/-R) and I/O/memory control strobes (-IOWR, -IORD, -MEMWR, -MEMRD). These are generated as bus cycle status decodes timed by the MC Bus -CMD transfer strobe. These signals directly drive the Z8530, SRAM and BUSTER read and write strobes.

The Board Enable (-BDENBL) output of the EPB2001, which reflects POS register 0102H bit 0, is used in this application to mask DMA requests and interrupts prior to enabling the card at system boot time. In addition, this bit is internally factored into the -CSx enable functions to disable chip selects when the board is disabled. This also suppresses generation of the -CDSFDBK line prior to card set-up.

CHIP SELECT LOGIC

The chip select block in the EPB2001 provides chip select outputs for the CMOS SRAM, SCC, and BUSTER Output Port. In addition, one of the -CSx outputs is used to drive the -CDDS16 (card data size 16) output to the MC Bus whenever the SRAM (organized as words) is accessed.

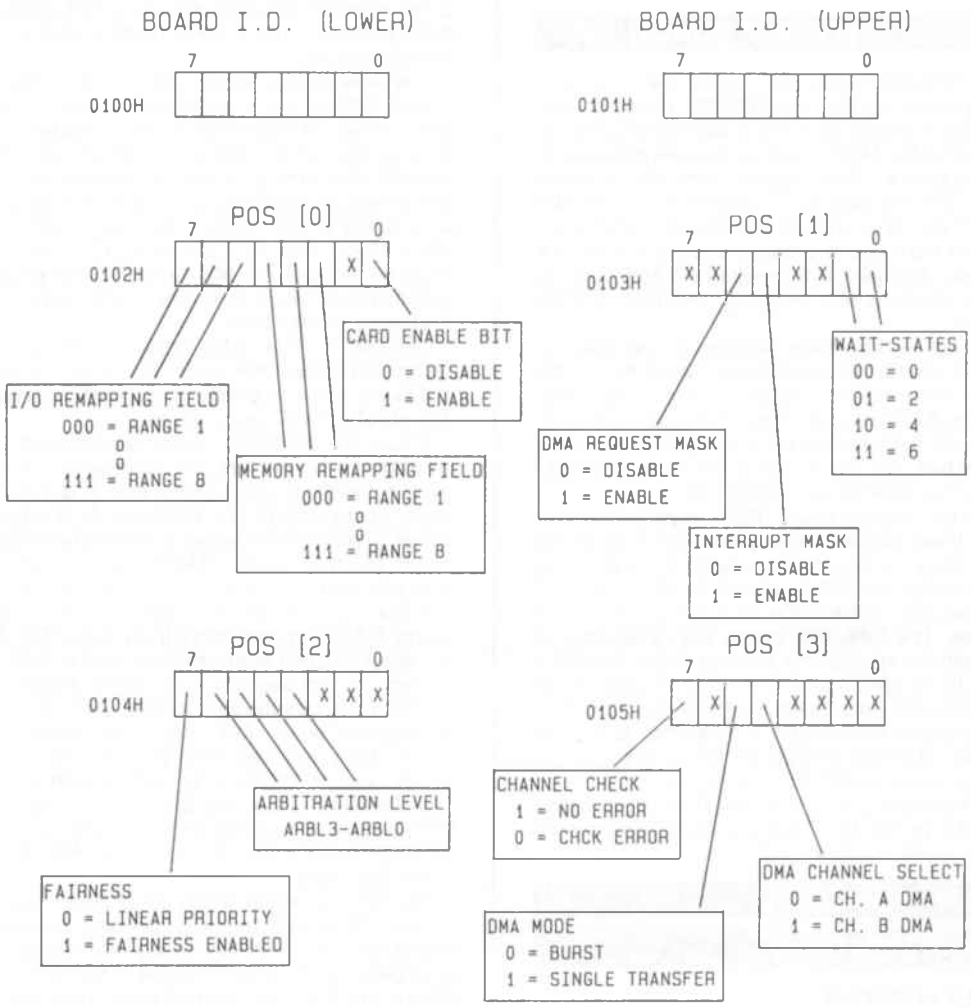
In this case, the first three chip select outputs will be latched by -ADL. The -CDDS16 output is not latched.

The -CDSFDBK (card select feedback) line to the MC Bus is an unlatched OR-function of appropriate chip selects. In this example, all -CSx functions with the exception of -CDDS16 will be factored into this signal. This selective ORing is a user-programmable feature of the EPB2001 architecture. Latching of these lines is also programmable. Remaining -CSx outputs could be used to drive other peripheral or memory chips, or supply design-specific board logic as described later.

Each of the chip selects has 8 pre-programmed ranges controlled by the address remapping fields specified in the POS registers. The programmable POS chip select enable decoder provides a unique mechanism for linking the POS registers to the chip select decode block. In this way the PS/2 can control address response ranges for the card and associated memory/I/O resources as it configures the system. This is used to eliminate address conflicts between cards without mechanical setting of DIP switches or jumpers, an error-prone process. The use of such mechanical arrangements in fact violates the Micro Channel specification.

Due to the structure of the chip select block as defined in the EPB2001/2002 Data Sheet, an address block of 2**N locations must sit on a 2**N address boundary. In other words, as the block size increases, the allowable number of base address positions decreases. For example, a 256K RAM chip select must have a base address which

Figure 2. Multi-Function POS Register Control Functions



POS I/O CONNECTIONS

REGISTER	BIT	PIN
0103	0	POSI/01
0103	1	POSI/02
0105	5	POSI/03
0105	4	POSI/04
0103	4	POSI/05
0103	5	POSI/06

is a multiple of 256K. This is not a severe restriction in most practical applications, but should be kept in mind as alternate address locations are selected.

EPB2002 INTERFACE FUNCTIONS

The EPB2002 does not require the degree of programmability that the EPB2001 contains. The EPB2002 implements the bus arbitration protocol defined for the Micro Channel in an asynchronous state machine. POS register bits are included which are readable and writeable from the MC Bus. These bits control the four-bit Arbitration Level for the board (arbitration priority) and enable Fairness (Fairness is a mechanism specified by IBM to eliminate bus "hogging" by bursting DMA devices).

The actual arbitration process is discussed in detail in the EPB2001/2002 Data Sheet and in the IBM documentation listed at the end of this note. The EPB2002 provides direct a.c. and d.c. compatibility with the interface signals required. For more information on this process the reader should consult the referenced documents.

The bit positions and POS register location which these bits will be mapped into is up to the board designer. The MC Bus specification does not define required POS locations for them. Any of POS registers 0102-0105H may be used for this purpose. The EPB2002 allows the remapping of these bits by appropriate connections to the SELx inputs to the chip. This is described in detail in the EPB2001/2002 Data Sheet. By connecting the SELx inputs as shown in Figure 1, these bits have been mapped into POS register 0104H for this application, bit locations D3-D7 on the bus.

Further details on the electrical interface of the EPB2002 to the BUSTER and SCC components will be given in a later section.

Z8530 INTERFACE CONSIDERATIONS

SYSTEM INTERFACE

Basic signals involved in interfacing the Z8530 to an MPU bus (the MC Bus in this case) are shown in Figure 3. These include -CE (Chip Enable), A/-B (Channel A/B designator), D/-C (Data/Command designator), -RD (Read Strobe), -WR (Write Strobe) and the 8-bit data bus, D0-D7. Here we are using the IBM MC Bus specification convention of a leading hyphen to indicate an active-low signal name. The Z8530 includes 9 Read-able and 15 Write-able registers to set device configuration, report status and access data. To reduce address input pin requirements, the Z8530 uses the concept of an address pointer register: any command register access (read or write) must be preceded by a write to the SCC with D/-C low and the address of the register to be accessed on the next cycle as the associated data. After the subsequent access, the pointer register is automatically reset to the address pointer register. Every command register access is thus a two bus cycle process: write address pointer, write or read selected register.

The waveforms shown are for data transfer cycles. Timing for a command cycle is similar, but involves the additional cycle mentioned above. Timing data is for the 4 MegaHertz Z8530. In general, the timing is very straightforward: the Z8530 requires that -CE, D/-C, and A/-B be set-up prior to the strobe (-RD or -WR) going active low. -CE's set-up time is 0 nS, while D/-C and A/-B must be set-up 80 nS prior to the strobe edge. All signals have a 0 nS hold time to the rising (inactive) edge of the strobe.

Data for a write cycle must be set-up 10 nS before the active -WR edge. In a read cycle, the Z8530 will place valid data on its D0-D7 pins within 250 nS of the -RD active edge.

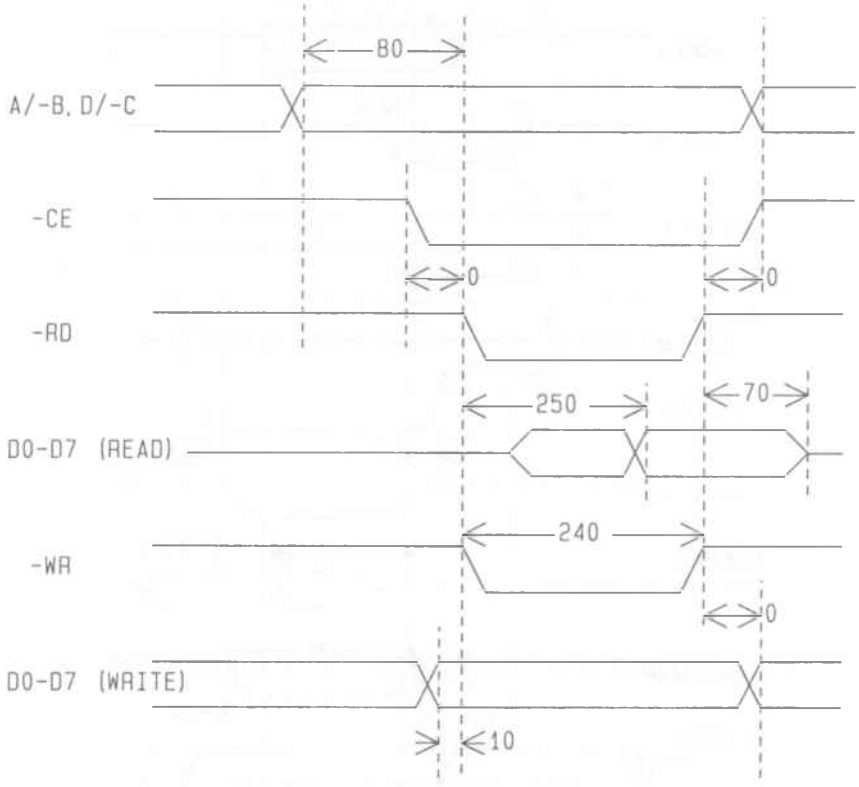
These are the SCC device requirements. The subsequent Figure 4 shows the signals provided by the EPB2001 chip select and board control logic. In this design, the D/-C and A/-B inputs to the SCC will be connected to the latched address inputs provided by the 74AS373 devices. The MC Bus provides a minimum set-up of 85 nS from address valid on the MC Bus to -CMD falling. Using 74AS373's, the delay from input (MC Bus) to output (latched board address bus) is 5 nS. The address is consequently valid at the Z8530 more than 80 nS before the strobe edge, since the strobe is triggered by the -CMD line going active.

The delay from address valid on the A0-A23 inputs to the EPB2001 -CSx outputs valid is 30 nS. The Z8530 requires only 0 nS, so over 55 nS of timing margin is available on the -CE path. The EPB2001 latches the -CE line with -ADL for the entire bus cycle.

The address ranges which the EPB2001 decodes for the -CE input to the SCC consist of blocks of 4 locations. The EPB2001 allows the designer to pre-specify up to 8 such ranges. The PS/2 POST (Power-On Self Test) routines can then relocate the Multi-Function card's resources to eliminate address conflicts with other adapters. Any I/O base address can be selected for the block which resides on a 4-byte boundary.

The EPB2001 asserts the -DEN line for the data transceivers within 20 nS of the DT/-R line falling (write cycle to board). This signal falls within 20 nS of -ADL going active. The MC Bus spec has a minimum of 40 nS from -ADL active to -CMD active. The -CMD signal triggers -IOWR, and -IOWR has a delay of up to 20 nS, which tracks parametrically with the -DEN delay. As a result, there will be a minimum of 20 nS between -DEN falling and -IOWR falling. The 74AS245's have a 9 nS delay from enable input to outputs valid. Data is therefore present at the SCC inputs 11 nS before the

Figure 3. Z8530/SCC Timing Requirements (4 MHz)



-IOWR (-WR strobe) input and supports data set-up time requirements of the Z8530.

The MC Bus requires drivers to tristate within 30 nS of -CMD going inactive. The 74AS245's 9 nS tri-state delay, coupled with the 20 nS -CMD to -DEN delay on the trailing edge, is a good match with the bus spec.

INTERRUPT INTERFACE

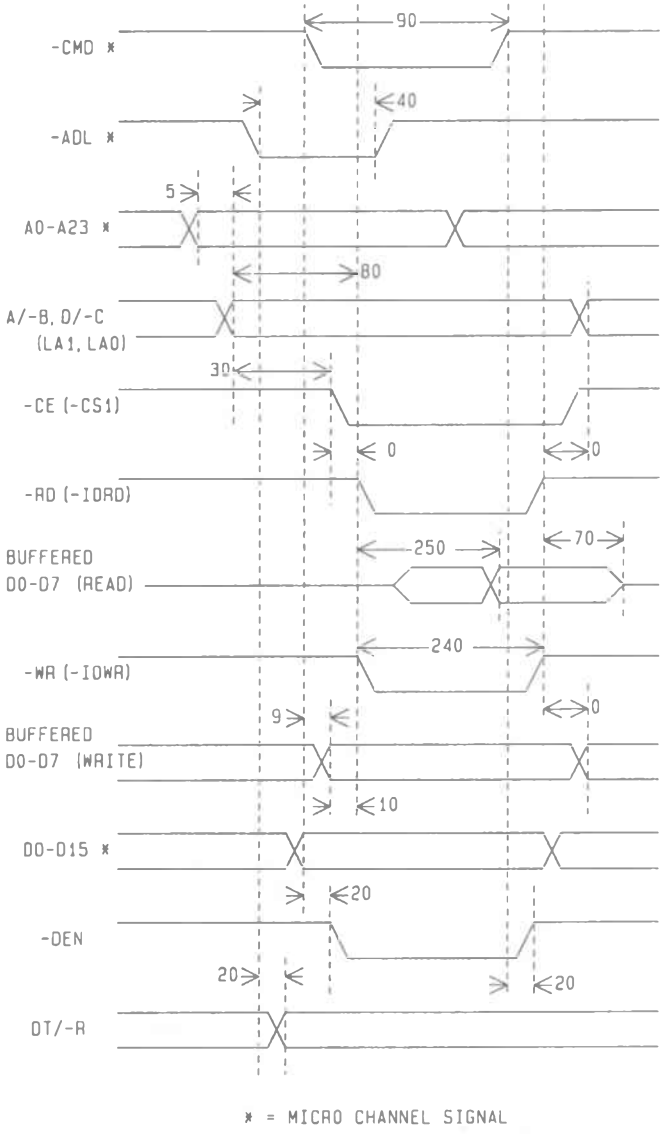
The SCC may be programmed to run in either an interrupt-driven or DMA transfer mode. On this particular board, hardware has been provided to support either mode, allowing the board software driver to select the data transfer mechanism. The Z8530 can be programmed to issue an interrupt request each time a byte of data is received, when the transmit buffer is empty, or on a variety of error conditions. The reader is referred to the Z8030/Z8530 Data Sheet for further details.

To support interrupt-driven operation, an open-collector 24 mA driver is used to connect the SCC's -INT output to the MC Bus interrupt -IRQ3. While the SCC has an open-drain output, its drive is insufficient to handle the MC Bus Iol requirements and must be re-buffered. Several prioritized -IRQx lines are available on the MC Bus, and in this case selection of the -IRQ3 line fixes the SCC's priority at a relatively low level, fourth lowest in the hierarchy. This is the alternate serial port level defined for the PS/2.

The Z8530 specification requires that the dedicated interrupt acknowledge -INTACK become valid 250 nS before -RD becomes active. This is not feasible in a single bus cycle and complicates logic design. However, the Z8530 also supports interrupt servicing by providing a register (Read Register #2) which contains an interrupt vector, and interrupt reset bits in Write Register #0. Interrupt acknowledge can therefore be treated as standard I/O read and write operations without special timing.

This Multi-Function card design does not use the channel check non-maskable interrupt protocol defined for the MC Bus. The -CHCK line, like the other interrupt lines on the MC Bus, is a level-sensitive, shared open-collector arrangement. A channel check request by any add-on card is wire ORed onto this line. Adapter logic can activate this line by an active-low pulse on the -SETCHK input to the EPB2001. This also resets POS register

Figure 4. EPB2001/Z8530 Micro Channel Timing



0105H, bit 7 (when inactive, a logic one). By interrogating this location on each adapter, the PS/2 channel check handler can determine which card has issued the channel check request.

DMA INTERFACE

In this application, the Z8530 will handshake with the EPB2002 when DMA transfers are enabled. Due to constraints in the basic DMA structure of the Micro Channel/PS/2, only one DMA channel per adapter is allowed. As a result, only one SCC channel can issue DMA requests at a time. To make this feature software configurable, a DMA request source selector has been built into the BUSTER support device. It has the two -W/REQx outputs of the SCC as inputs, and generates either a -BRSTREQ or -BUSREQ line for the EPB2002. The DMA request mode is controlled by a POS register bit as outlined earlier. A POS register bit selects which channel is the DMA requestor. DMA requests can also be permanently disabled by

another POS bit for interrupt-driven mode on both channels. Interrupt-driven operation may be used on one channel while DMA operation is employed on the other.

The Z8530 may be programmed to assert -W/REQx on a character receive or transmit buffer empty condition. This signal is routed to either the EPB2002 -BRSTREQ or -BUSREQ input, and results in the EPB2002 asserting -PREEMPT. Timing for the assertion is not critical.

Timing for the removal of the request is important, however. The Z8530 deasserts -W/REQx line 240 nS after the corresponding strobe goes active. This signal experiences a 35 nS delay in going through the DMA request selector in the EPB 1400. In the burst transfer mode, the EPB2002 will deassert -BURST approximately 50 nS after this signal is deasserted, and at the same time BUSGNT will be deasserted. To properly terminate the cycle, -BURST must be inactive 35 nS before -CMD rises. Therefore, in order to guarantee correct burst operation, it is necessary that the strobe width (and therefore -CMD active width) be

240 nS + 35 nS + 50 nS + 35 nS = 360 nS

To guarantee the absence of DMA overrun (erroneous extra transfers), the DMA transfer cycle must be extended via wait states so this requirement is satisfied.

Since -BURST is not used in single transfer mode, this timing constraint need not be considered. However, for simplicity, in this design single transfer and burst wait-states will be the same.

In single transfer mode, it is necessary to deassert the -BUSREQ line on the occurrence of BUSGNT active and the -S0, -S1 status lines going to the active state (indicating start of the DMA transfer). The logic in the lower left of the BUSTER schematic (Figure 6) performs this function.

For burst transfers, the BUSGNT signal is not used on the adapter. Since the PS/2's resident DMA channels on the motherboard are being used, BUSGNT is not needed to communicate bus ownership. As long as -BURST is active, the DMA channels will execute transfer cycles. DMA acknowledge for both cases consists of the DMA channel actually accessing (reading/writing) the SCC data buffer. No dedicated DMACK lines appear on the MC Bus. In other applications, BUSGNT migt be used to reset a request transfer flip-flop or otherwise indicate to board logic the granting of the bus.

SERIAL INTERFACE

Minimal serial interfaces for Channel A & B are shown in this design. Modem control signals (-RTS, -CTS, -DCD) are not used to control data transfer over the RS-232 links. This could easily be added by programming the SCC for such operation and adding the appropriate buffers. With the arrangement shown, only the RxD, TxD and Ground lines for each RS-232 line need be connected.

BUSTER SUPPORT LOGIC

WAIT-STATE LOGIC

Overall strobe (-RD or -WR) width required by the SCC is 250 nS. Since the default MC Bus cycle provides only a 90 nS -CMD width, the cycle must be extended by use of a wait-state generator, in this case designed into a BUSTER component. The timing source for the wait-state generator, the OSC signal from the MC Bus, has a period of about 70 nS. The wait-state generator will pull CHRDY on the MC Bus inactive (low) for a number of OSC clock edges determined by a value written into a POS register bit field.

The number of wait-states is software programmable, and can be selected by parameters in the add-on card's Adapter Description File (ADF). The ADF, required for every add-on card, describes to the PS/2 the available address response ranges and configuration options for a given card. The system reads the board I.D. for each card in its backplane when the syetem is booted. The board I.D. is used to find the associated ADF file on the system's disk, and the information there is used to configure the adapter and/or to eliminate address range conflicts with other cards.

BUSTER's 28 MegaHertz operating frequency supports the wait-state generator and clock divider functions with considerable margin given the 14 MegaHertz clock inputs. The number of wait-states can be varied from 0 to 6. A minimum number of clock edges to support the 250 nS width required by the 4 MegaHertz SCC is 4 x 70 nS = 280 nS. If DMA is used as described earlier, 6 x 70 nS = 420 nS must be used. The generator is triggered by an active -CE input to the SCC, indicating an Z8530 access to/from the MC Bus.

The actual wait-state generator consists of a loadable down-counter shown in Figure 6. The counter is automatically reloaded each time it reaches zero. While counting, CHRDY is held low.

OUTPUT PORT

As an added feature, an output port is constructed in a portion of the BUSTER EPB1400 for general-purpose use. These 8 lines may be used to control whatever output functions are required. This block is shown at the top of Figure 6. It is entered using Altera's LogiCaps schematic capture package and TTL MacroFunction elements. The integral MPU data port on the EPB2001 simplifies the MC Bus interface.

This output port uses another of the EPB2001's -CSx outputs for selection during write operations. The -WS input to the EPB1400 is connected to the -IOWR output of the EPB2001. Since the output port is the only resource on the BUSTER device mapped into the I/O address space, no address inputs are needed.

BUSTER's write control interface timing is very simple: a minimum write strobe (-WS) low width of 20 nS will insure correct writing of the output port when accompanied by valid data and chip select. To be precise, data must be valid 7 nS prior to BUSTER's -WS input falling, and the chip select 10 nS prior to -WS falling. Since MC Bus data is valid at -CMD's leading edge (prior to -IOWR leading edge), and the EPB2001's -CSx lines are valid at least 55 nS prior to -CMD falling, the minimum overlaps are easily satisfied. Minimum strobe width of 90 nS is also more than adequate.

STATIC RAM INTERFACE CONSIDERATIONS

The 32K word static RAM on the Multi-Function card has been implemented using two 32K x 8 CMOS SRAM devices. These devices are 62256LP-10 RAMs, packaged in 28 lead, 600 mil DIP packages. The devices have 15 address inputs (A0-A14), eight bidirectional data lines (I/O0-I/O7), a chip select (-CS), output enable (-OE) and write enable (-WE).

For these chips, access time is 100 nS from address or chip select. Output enable delay from -OE to valid read data is 50 nS. Minimum write enable width (-WE) is 75 nS, with a 40 nS set-up time for of valid write data to -WE trailing edge.

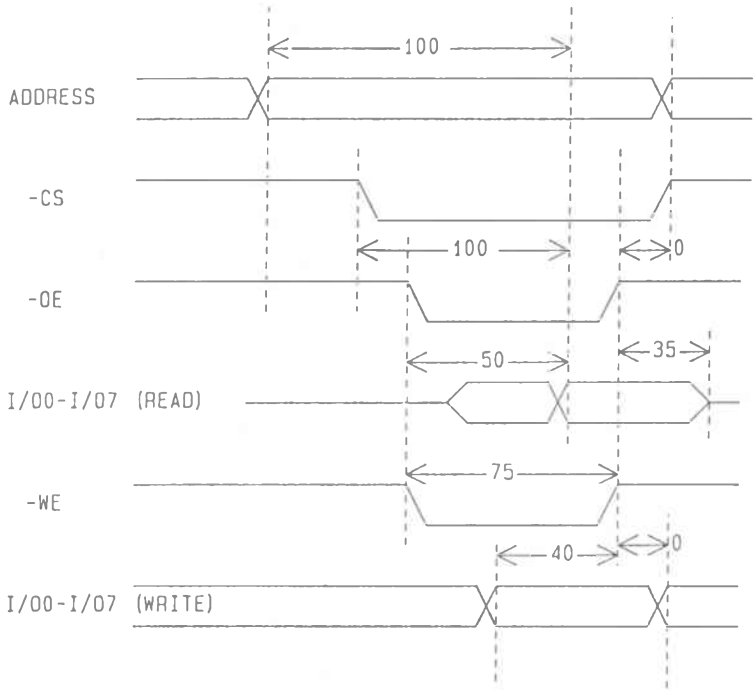
As shown in the Multi-Function board schematic (Figure 1), addresses from the 74AS373's drive the SRAM address inputs. SRAM I/O's drive the upper and lower bytes of the board data bus. The -CE inputs are driven by a -CSx output of the EPB2001, while -OE is connected to -MEMRD and -OE to -MEMWR.

As discussed earlier, the MC Bus guarantees 85 nS set-up of addresses to -CMD. 5 nS delay through the 74AS373's gives greater than 80 nS of address to -MEMRD or -MEMWR set-up at the SRAM devices. Even assuming minimum strobe width of 90 nS (minimum -CMD width on the MC Bus), this allows 170 nS of address access time at the SRAMs. Chip select access is reduced by the chip select decode delay on the EPB2001 (30 nS), but is still 140 nS.

Since the output enable delay on the SRAMs is 50 nS, even with a 9 nS delay through the 74AS245 transceivers, data is valid on the bus in time to meet the 60 nS MC Bus maximum delay for read data.

The SRAMs require 40 nS of valid write data/-WE overlap to guarantee correct writing of the selected locations. The MC Bus specifies 0 nS of write data set-up to the leading edge of -CMD, and a mini-

Figure 5. 62256LP-10 Timing Requirements



mum 90 nS -CMD width. This results in >80 nS of overlap, more than adequate even with the 9 nS buffer delay.

GENERAL EPB2001/2002

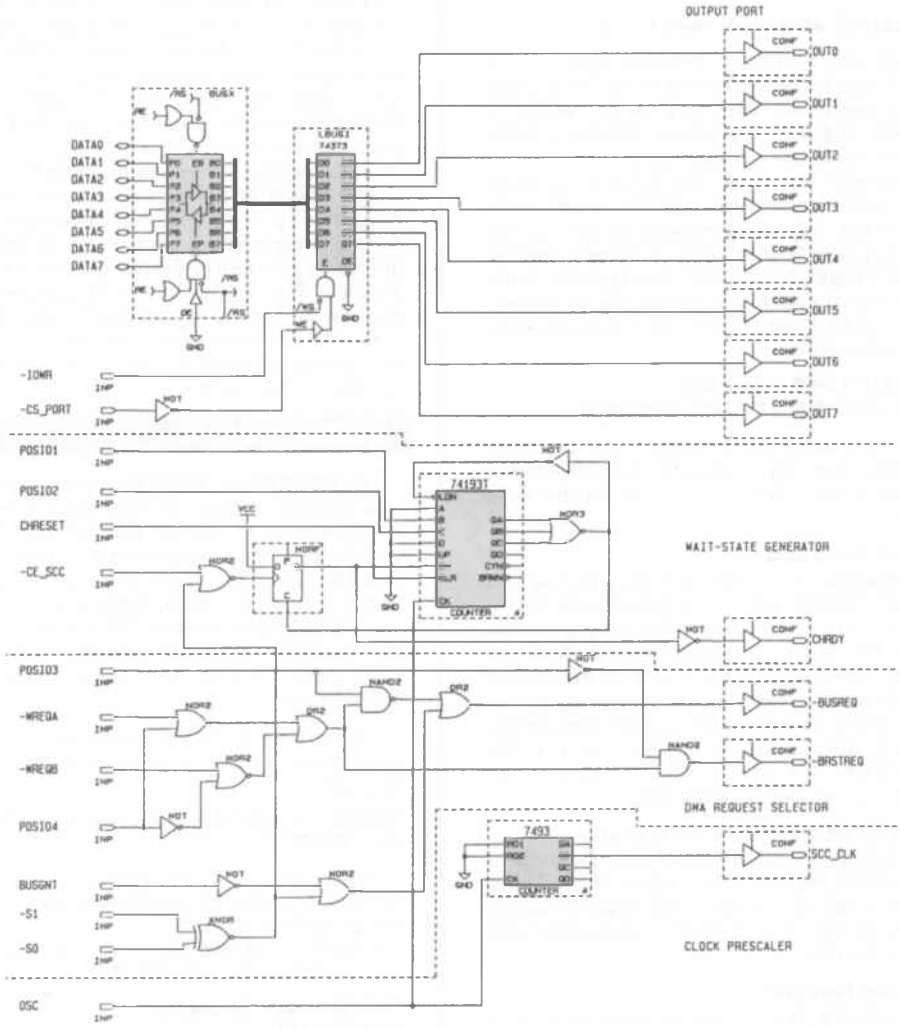
DESIGN TIPS

EPB2001 CHIP SELECT LOGIC USE IN 32-BIT SYSTEMS

The decoding inputs to the EPB2001 are labelled A0-A23. This allows full decoding of 24-bit MC Bus addresses down to the byte level. Even though 32 address inputs are not provided, the device is still applicable to 32-bit machines. This is true because the granularity required in chip select ranges is often quite coarse.

A typical adapter may be memory only, I/O only, or some mix of memory and I/O. A0-A2 on the EPB2001 must always be connected to the A0-A2 pins on the MC Bus to assure correct access of the POS registers and board I.D. If the A3-A23 inputs to the EPB2001 are connected to the A11-A31 lines on the MC Bus, the chip select logic can decode

Figure 6. BUSTER Multi-Function Support Chip



addresses down to block sizes of 2K bytes over the full 32-bit address range. Since most RAM chips in use today are much greater than 2K byte density, this is usually sufficient for memory addressing and memory-only applications.

Mixed memory/I/O adapters can also use this scheme. However, it implies that I/O peripherals also be placed on 2K byte boundaries, and is somewhat wasteful of the address space. Only 32 such blocks are available. Further decoding of A3-A10 external to the EPB2001 can reduce or eliminate this waste.

Pure I/O applications require only A0-A15. In such cases, more than enough address inputs are available on the EPB2001 for 16- or 32-bit applications.

ALTERNATIVE ADDRESS INPUT USE

If an I/O-only adapter is being designed, or one for 16-bit applications specifically, all 24 address inputs, as well as MADE24 may not be needed. In such cases, these inputs may be used to provide additional board-specific functions.

For example, it may be desireable to generate read or write strobes for specific board I.C.'s. These strobes may be generated as an address decode logically ANDed with a transfer strobe. The -CMD signal on the MC Bus typically functions as such a strobe. By connecting an unused Ax pin to -CMD, it can be factored into appropriate read or write strobes.

ALTERNATE USES FOR THE EPB2001 CHIP SELECT LOGIC BLOCK

The chip select programmable logic block on the EPB2001 is primarily used to generate adapter chip select signals. Some other uses for this logic includes:

- **Latched Addresses—**
  -CSx outputs of the EPB2001 may be used as latched address outputs for general use on the add-on card. In this case, the programmable chip select block is programmed to drive the corresponding output low whenever the desired address input is low. The output is latched by -ADL. When used for this purpose, the corresponding -CSx output is not factored into the -CDSFDBK line to the Micro Channel (an EPB2001 programmable feature).
- **Additional POS Register Bit Access—**
  -CSx outputs of the EPB2001 may be used to access POS register bits (output only). These may be used to increase POS register output pins to 24 from the 16 POSI/O lines dedicated for this purpose.
- **Data Size Feedback—**
  -CSx outputs have sufficient drive to directly drive the -CDDS32 and -CDDS16 lines on the Micro Channel. These signals indicate a devices's data bus width as 32 or 16 bits, respectively. They are derived as unlatched decodes of appropriate address ranges.

As described in the EPB2001/2002 Data Sheet, chip select address ranges may be enabled by any combination of POS register bits. Typically, a bit field is designated within a POS register to act as an Address Relocation control field. It may be desireable to factor into certain chip select enabling functions special bits such as POS 0102 bit 0 (card enable) or POS 0105 bit 7 (channel check). In this way, chip selects ively disable is disabled or during error recovery (channel check) routines.

MC MAP SOFTWARE SUPPORT

The MC Map Micro Channel Interface Assembler, available from Altera, provides an easy table-driven entry mechanism for specifying EPB2001 designs. All programmable options mentioned above are easily specified using this PC-based package. Once design entry is finished, the design is compiled in less than a minute and the resulting JEDEC file may be used to program the EPB2001. Programming of the component may be accomplished using Altera's PC-based hardware: an LP4 or LP5 programming card, PLE3-12 Master Programming Unit and PLEJ2001 programming pinout adapter. The actual device is programmed in seconds and ready for use on the board.

Further information on the MC Map software and programming hardware may be obtained from Altera's Marketing group.

SUMMARY

The EPB2001 and EPB2002 Micro Channel interface devices provide all required interface functions between an IBM PS/2 add-on board and the system bus. The programmable nature of the EPB2001 provides many possibilities for implementing adapter-specific functions. Coupled with other EPLDs such as BUSTER for unique application features, extremely efficient interfaces for PS/2 adapters can be rapidly designed and implemented.

Information on IBM's Independent Developer Assistance Program may be obtained from IBM through

IBM Independent Developer Assistance Program—
(800)426-7736

Registration in this program is a prerequisite for obtaining technical assistance and board I.D. assignments from IBM.

Specifications and technical reference manuals may be obtained from IBM through

IBM Technical Directory—
(800)426-7282

AB14 Rev 1.0

MULTIPLE ARRAY MATRIX EPLDS **PAGE NO.**

7

FEATURES

- **Erasable, User-Configurable CMOS EPLDs capable of implementing high density custom logic functions.**
- **Advanced 0.8 micron double-metal CMOS EPROM technology.**
- **Multiple Array MatriX Architecture optimized for speed and density.**
  - **—Typical clock frequency = 50MHz**
  - **—Programmable Interconnect Array (PIA) simplifies routing**
  - **—Flexible Macrocells increase utilization**
  - **—Programmable clock control**
  - **—Expander product terms implement complex logic functions**
- **MAX+PLUS development system eases design.**
  - **—Runs on IBM-AT and compatible machines**
  - **—Hierarchical schematic capture with 7400 series TTL and custom Macrofunctions**
  - **—State machine and Boolean entry**
  - **—Graphical delay path calculator**
  - **—Automatic error location**
  - **—Timing simulation**
  - **—Graphical interactive entry of waveforms**

GENERAL DESCRIPTION

The Altera Multiple Array MatriX (MAX) family of EPLDs provides a User-Configurable, High-Density solution to general purpose logic integration requirements. With the combination of innovative architecture and state of the art process, the MAX EPLDs offer LSI density, without sacrificing speed.

The MAX architecture makes it ideal for replacing large amounts of TTL SSI and MSI logic. For example, a 74161 counter utilizes only 3% of the 128 Macrocells available in the EPM5128. Similarly, a 74151 8 to 1 multiplexer consumes less than one percent of the over 1,000 product terms in the EPM5128. This allows the designer to replace 50 or more TTL packages with just one MAX EPLD. The family comes in a range of densities, shown below. By standardizing on a few MAX building blocks, the designer can replace hundreds of different 7400 series part numbers currently used in most digital systems.

The family is based on an architecture of flexible Macrocells grouped together into Logic Array Blocks (LABs). Within the LAB is a group of additional product terms called Expander Product Terms. These Expanders are used and shared by the Macrocells, allowing complex functions, up to 35 product terms, to be easily implemented in a single Macrocell. A Programmable Interconnect Array (PIA) globally routes all signals within devices containing more than one LAB. This architecture is fabricated on an advanced 0.8 micron CMOS EPROM process, yielding devices with 3 times the integration density at twice the system clock speed of the largest current generation EPLD.

MAX FAMILY MEMBERS

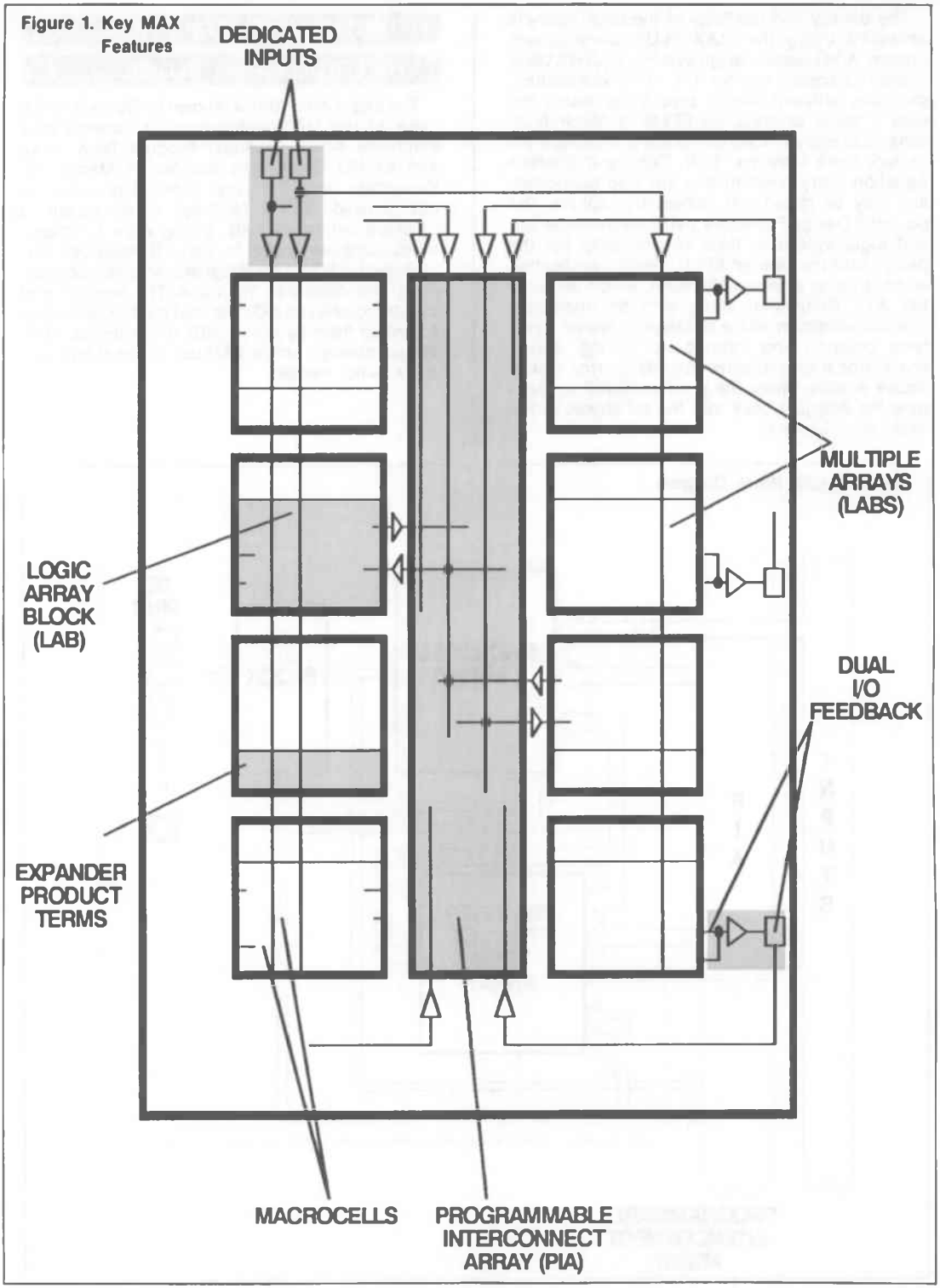
FEATURE	EPM 5016	5024	5032	5064	5127	5128
MACROCELLS	16	24	32	64	128	128
MAX FLIP FLOPS	16	24	32	64	128	128
MAX LATCHES(1)	32	48	64	128	256	256
MAX INPUTS(2)	15	19	23	35	35	59
MAX OUTPUTS	8	12	16	28	28	52
PACKAGES	20D	24D	28D	40D	40D	68J
			28J	44J	44J	68G

KEY: D - DIP J - J-LEAD CHIP CARRIER G - PIN GRID ARRAY

Notes: (1) When all Expander Product Terms are used to implement latches.

(2) With one output.

Figure 1. Key MAX Features



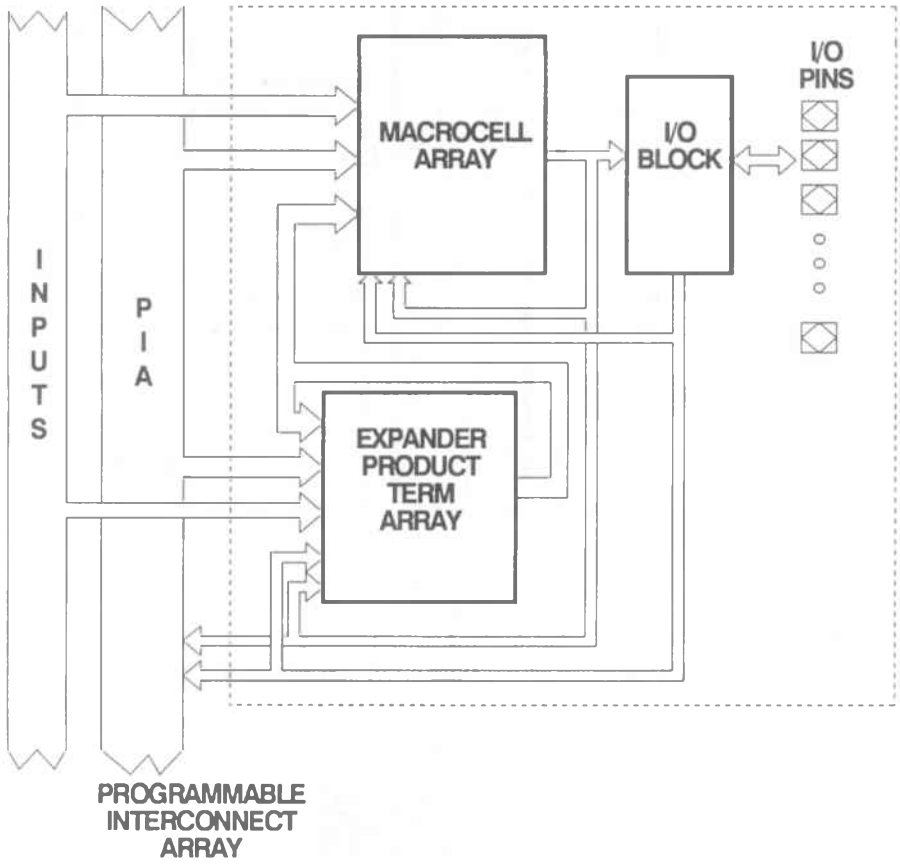
The density and flexibility of the MAX family is accessed using the MAX+PLUS development system. A PC based design system, MAX+PLUS is optimized specifically for the MAX architecture, providing efficient design processing within the time it takes to erase an EPLD. A hierarchical schematic entry mechanism is used to capture the design. State Machine, Truth Table and Boolean Equation entry mechanisms are also supported, and may be mixed with schematic capture. The powerful Design Processor performs minimization and logic synthesis, then automatically fits the design into the desired EPLD. Design verification is done using a timing simulator, which provides full A.C. simulation, along with an interactive graphic waveform editor package to speed waveform creation and debugging. During design processing a sophisticated automatic error locator shows exactly where the error occurred by popping the designer back into the schematic at the exact error location.

FUNCTIONAL DESCRIPTION

THE LOGIC ARRAY BLOCK

The Logic Array Block, shown in Figure 2, is the heart of the MAX architecture. It consists of a Macrocell Array, Expander Product Term Array, and an I/O Block. The number of Macrocells, Expanders, and I/O vary, depending upon the device used. Global feedback of all signals is provided within an LAB, giving each functional block complete access to the LAB resources. The LAB itself is fed by the Programmable Interconnect Array and dedicated input bus. The feedbacks of the Macrocells and I/O pins feed the PIA, providing access to them by other LABs in the device. MAX EPLDs having a single LAB use a global bus, and a PIA is not needed.

Figure 2. LAB Block Diagram



THE MAX MACROCELL

Traditionally, PLDs have been divided into either PLA (programmable AND, programmable OR), or PAL (programmable AND, fixed OR) architectures. PLDs of the latter type provide faster input-to-output delays, but can be inefficient due to fixed allocation of product terms. Statistical analysis of PLD logic designs has shown that 70 % of all logic functions (per Macrocell) require 3 product terms or less.

The Macrocell structure of MAX has been optimized to handle variable product term requirements. As shown in Figure 3, each Macrocell consists of a product term array and a configurable register. In the MAX Macrocell, combinatorial logic is implemented with 3 product terms OR'ed together, which then feeds an XOR gate. The second input to the XOR gate is also controlled by a product term, providing the ability to control active high or active low logic. The MAX+PLUS software will also use this gate to implement complex mutually exclusive-OR arithmetic logic functions, or to do DeMorgan's Inversion, reducing the number of product terms required to implement a function.

If more product terms are required to implement a given function, they may be added to the Macrocell from the Expander Product Term Array. These additional product terms may be added to any Macrocell, allowing the designer to build gate intensive logic, such as address decoders, adders, comparators, and complex state machines, without using extra Macrocells.

The register within a MAX Macrocell may be programmed for either D, T, JK, or SR operation. It may alternately be configured as a flow-through latch for minimum input to output delays, or bypassed entirely for purely combinatorial logic. In addition, each register supports both asynchronous preset and clear, allowing asynchronous loading of counters or shift registers, as found in many standard TTL functions. These registers may be clocked with a synchronous system clock, or clocked independently from the logic array.

Figure 3. Macrocell Block Diagram

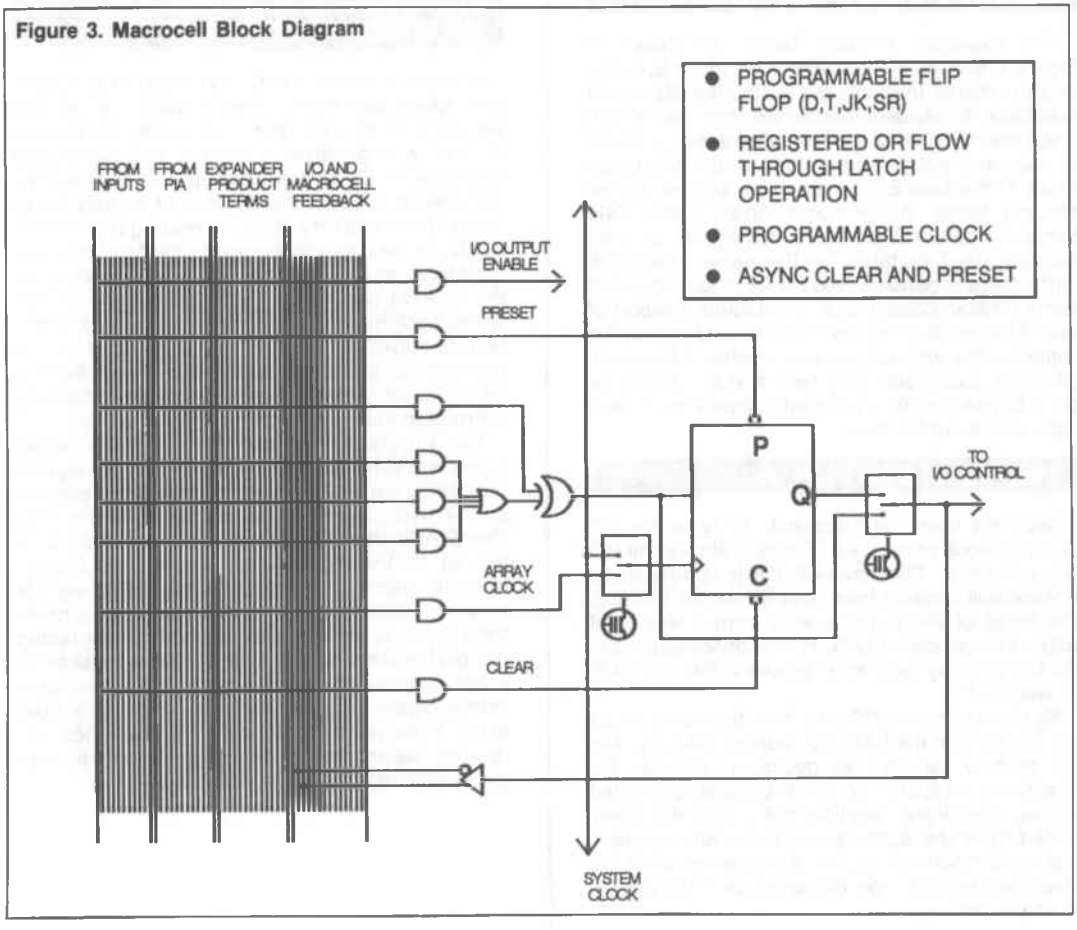
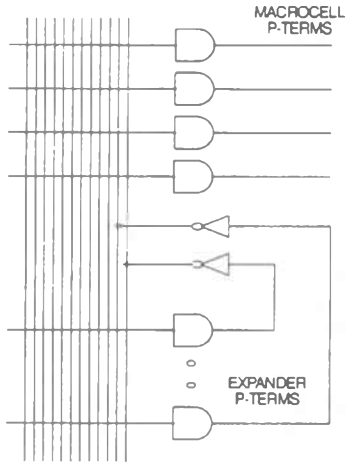


Figure 4.



EXPANDER PRODUCT TERMS

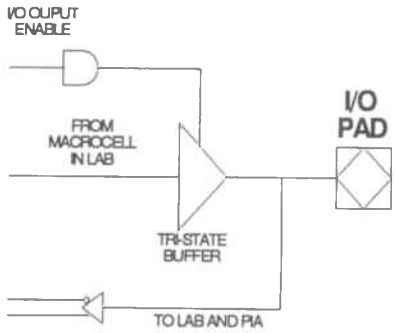
The Expander Product Terms, as shown in Figure 4, are fed by the Dedicated Input Bus, the Programmable Interconnect Array, the Macrocell Feedback, Expanders themselves, and the I/O pin feedbacks. The outputs of the Expanders then go to each and every product term in the Macrocell Array. This allows Expanders to be "shared" by the product terms in the Logic Array Block. One Expander can all Macrocells in the LAB, or even multiple product terms in the same Macrocell. Since these Expanders feed the secondary product terms (Preset, Clear, Clock, and Output Enable) of each Macrocell, complex logic functions may be implemented without utilizing another Macrocell. Likewise, Expanders may feed and be shared by other Expanders, to implement complex multi-level logic and input latches.

THE I/O BLOCK

Separate from the Macrocell Array is the I/O Control Block of the LAB. Figure 5 shows the I/O block diagram. The tristate buffer is controlled by a Macrocell product term, and drives the I/O pad. The input of this buffer comes from a Macrocell within the associated LAB. The feedback path from the I/O pin may feed other blocks within the LAB, as well as PIA.

By decoupling the I/O pins from the flip-flops, all the registers in the LAB are "buried", allowing the I/O pins to be used as dedicated outputs, Bi-directional outputs, or as additional dedicated inputs. Therefore, applications requiring many buried flip-flops, such as counters, shift registers, and state machines, no longer consume both the Macrocell register and the associated I/O pin, as in earlier devices.

Figure 5. I/O Control



THE PROGRAMMABLE INTERCONNECT ARRAY

A major problem which has limited PLD density and speed has been signal routing, i.e. getting signals from one Macrocell to another. For smaller devices, a single array is used and all signals are available to all Macrocells. But, as the devices increase in density, the number of signals being routed becomes very large, increasing the amount of silicon used for interconnections. Also, because the signal must be global, the added loading on the internal connection path reduces the overall speed performance of the device. The MAX architecture solves these problems. It is based on the concept of small, flexible Logic Array Blocks, which, in the larger devices, are interconnected by a Programmable Interconnect Array, or PIA.

The Programmable Interconnect Array solves interconnect limitations by routing only the signals needed by each LAB. The architecture is designed so that every signal on the chip is within the PIA. The PIA is then programmed to give each LAB access to the signals that it requires. Consequently, each LAB receives only the signals needed. This effectively solves any routing problems that may arise in a design, without degrading the performance of the device. Unlike masked or programmable gate arrays, which induce variable delays dependent on routing, the PIA has a fixed delay from point to point. This eliminates undesired skews among logic signals, which may cause glitches in internal or external logic.

FAMILY MEMBERS

The MAX family is an entire set of modular building blocks, optimized for high speed and high density. Listed below are the 6 current members of the family.

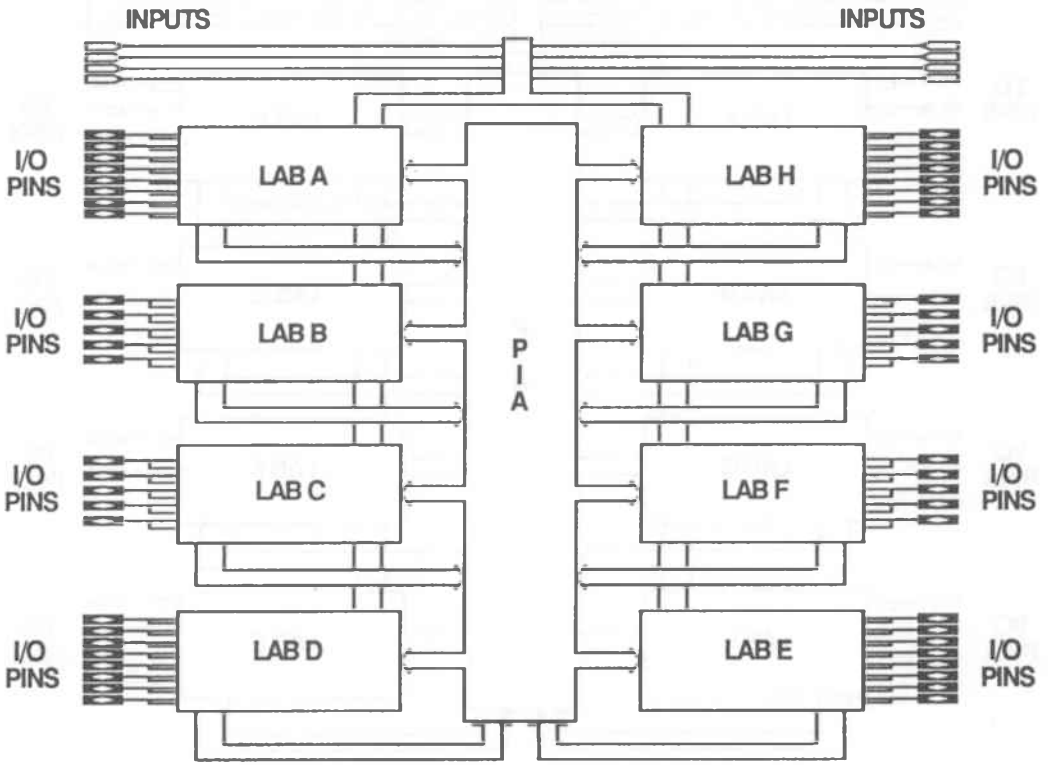
EPM5128

- 128 Macrocells in 8 LABs.
- 8 Dedicated Inputs, 52 Bi-directional I/O pins.
- Programmable Interconnect Array.
- Available in 68 Pin JLCC, PLCC, and PGA.

The 128 Macrocells in the EPM5128 are divided into 8 Logic Array Blocks, 16 per LAB. There are 256 Expander Product Terms, 32 per LAB, to be used and shared by the Macrocells within each LAB. Each LAB is interconnected with a Programmable Interconnect Array, allowing all signals to be routed throughout the chip.

The speed and density of the EPM5128 allows it to be used in a wide range of applications, from replacement of large amounts of 7400 series TTL logic, to complex controllers and multi-function chips. With greater than 25 times the functionality of 20 pin PLDS, the EPM5128 allows the replacement of over 50 TTL devices. By replacing large amounts of logic, the EPM5128 reduces board space, part count, and increases system reliablity.

Figure 6. EPM5128 Block Diagram



FEATURES

INPUTS	I/O PINS	LABS	MACROCELLS PER LAB	TOTAL MACROCELLS	EXPANDERS	PIA
8	52	8	16	128	256	YES

7

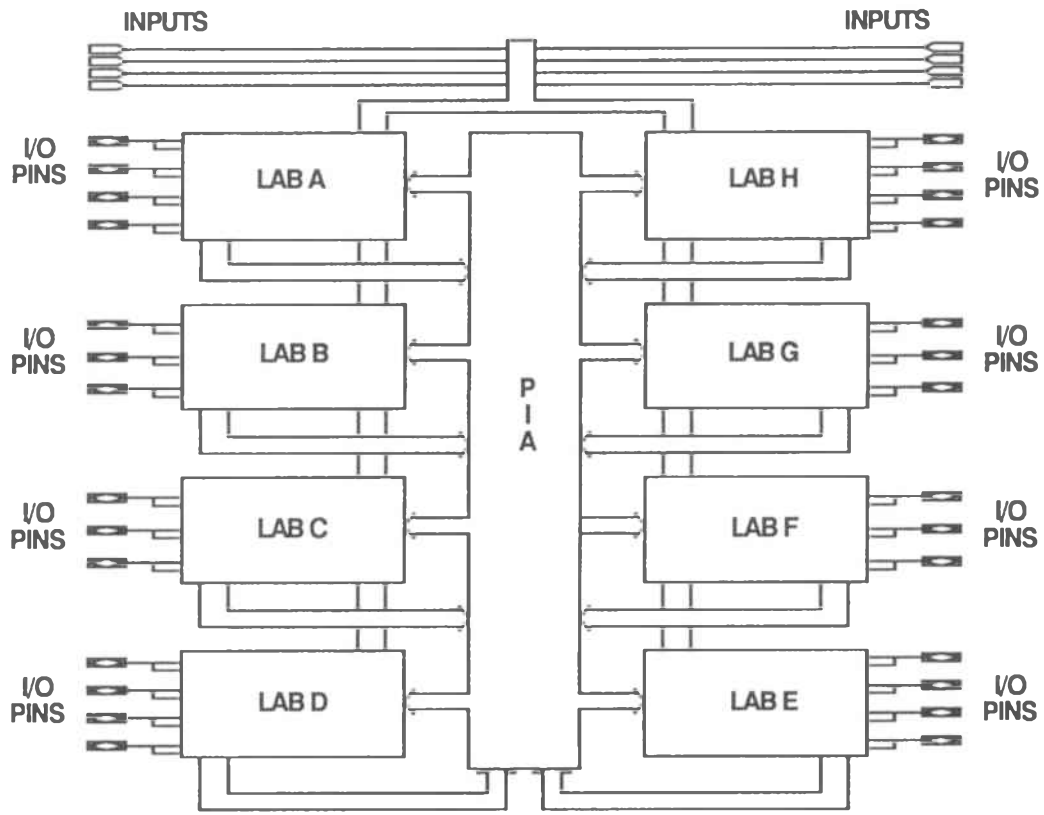
EPM5127

- 128 Macrocells in 8 LABs.
- 8 dedicated inputs, 28 Bi-directional I/O pins.
- 256 Expander Product Terms.
- Programmable Interconnect Array.
- Available in 40 Pin CDIP, PDIP, and 44 Pin JLCC or PLCC.

The EPM5127 packs the same LSI density of the EPM5128 into a smaller, 40 pin DIP or 44 pin JLCC package. Designed for applications in which large amounts of logic must be packed into a very small area, the EPM5127 is ideally suited for applications which require large amounts of buried logic.

It has the same number of Macrocells and expanders as the EPM5128, and a Programmable Interconnect Array to allow communications between the LABs. Each LAB has an I/O block, with LABs A, D, E, and H having 4 Bi-directional tri-stateable I/O pins, and the rest having 3 I/O pins. Like all other EPLDs in the MAX family, these I/O pins support dual feedback. In this way any Macrocells may be buried, with only the output of Macrocells needed off-chip connected to I/O pins.

Figure 7. EPM5127 Block Diagram



FEATURES

INPUTS	I/O PINS	LABS	MACROCELLS PER LAB	TOTAL MACROCELLS	EXPANDERS	PIA
8	28	8	16	128	256	YES

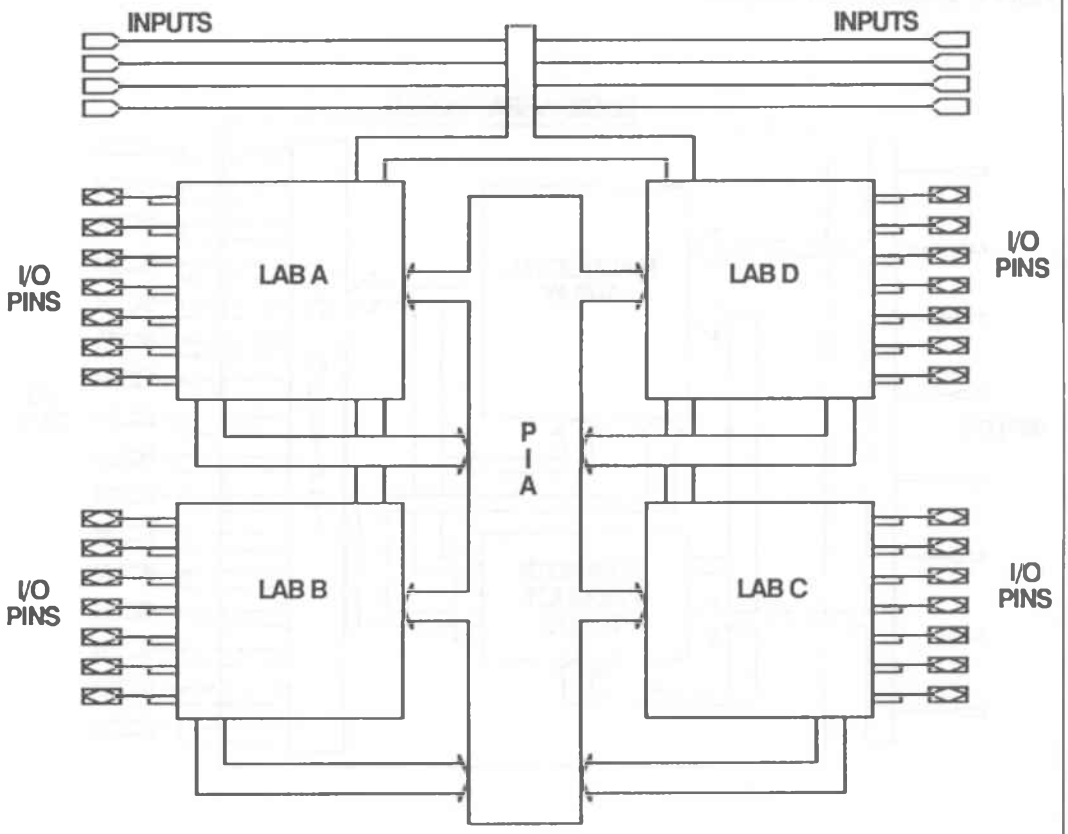
EPM5064

- 64 MAX Macrocells in 4 LABs.
- 8 Dedicated Inputs, 28 tri-stateable, Bi-directional I/O pins.
- Programmable Interconnect Array.
- Available in 40 Pin CDIP, PDIP, and 44 Pin JLCC, PLCC.

The EPM5064 block diagram is shown in Figure 8. It has 16 Macrocells and 32 Expander Product Terms in each of its 4 Logic Array Blocks. Decoupled from the Macrocells in the LABs, each I/O control block has 7 I/O pins. Therefore, if each I/O pin was fed by a Macrocell, there are still 9 buried Macrocells per LAB that may be used for embedded logic. The signals generated within each LAB are routed to every LAB through the Programmable Interconnect Array.

The EPM5064 is perfect for designs with large I/O requirements, along with healthy amounts of buried logic. Excellent for a wide range of applications, the EPM5064 can reduce board space by absorbing large amounts of glue logic. Due to the large number of I/O pins, 16 bit data paths are no problem.

Figure 8. EPM5064 Block Diagram



FEATURES

INPUTS	I/O PINS	LABS	MACROCELLS PER LAB	TOTAL MACROCELLS	EXPANDERS	PIA
8	28	4	16	64	128	YES

7

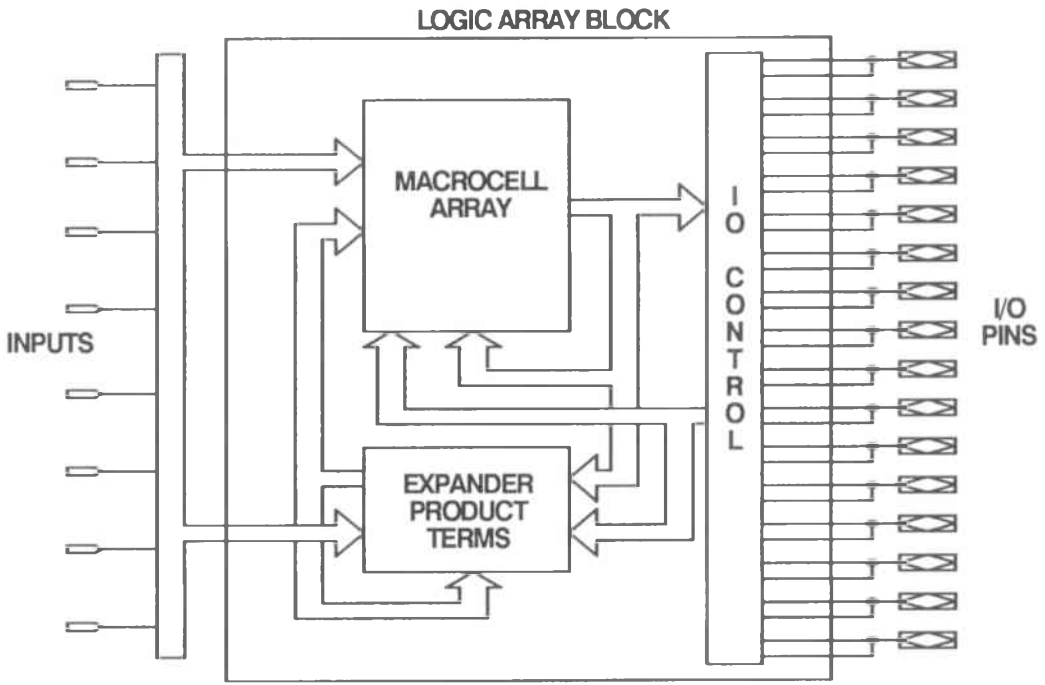
EPM5032

- High performance, high density replacement for TTL, 74HC, and custom logic.
- 32 Macrocells, 64 Expander Product Terms in one LAB.
- 8 dedicated inputs, 16 I/O pins.
- Small outline 28 Pin 300Mil CDIP,PDIP, or 28 Pin JLCC, PLCC package

Available in a 28 pin 300 mil DIP or JLCC, the EPM5032 represents the densest EPLD of this size. 8 dedicated inputs and 16 Bi-directional I/O pins communicate to one Logic Array Block. In the EPM5032 LAB there are 32 Macrocells and 64 Expander Product Terms. Figure 9 shows that even if all of the I/O pins are being driven by Macrocells, there are still 16 "buried" Macrocells available. All inputs, Macrocells and I/O pins are interconnected within the LAB.

The speed and density of the EPM5032 makes it a natural for all types of applications. With just this one device, the designer can implement complex state machines, registered logic, and combinatorial "glue" logic, without using multiple chips. This architectural flexibility allows the EPM5032 to replace multi-chip TTL solutions, whether they are synchronous, asynchronous, combinatorial, or all three.

Figure 9. EPM5032 Block Diagram



FEATURES

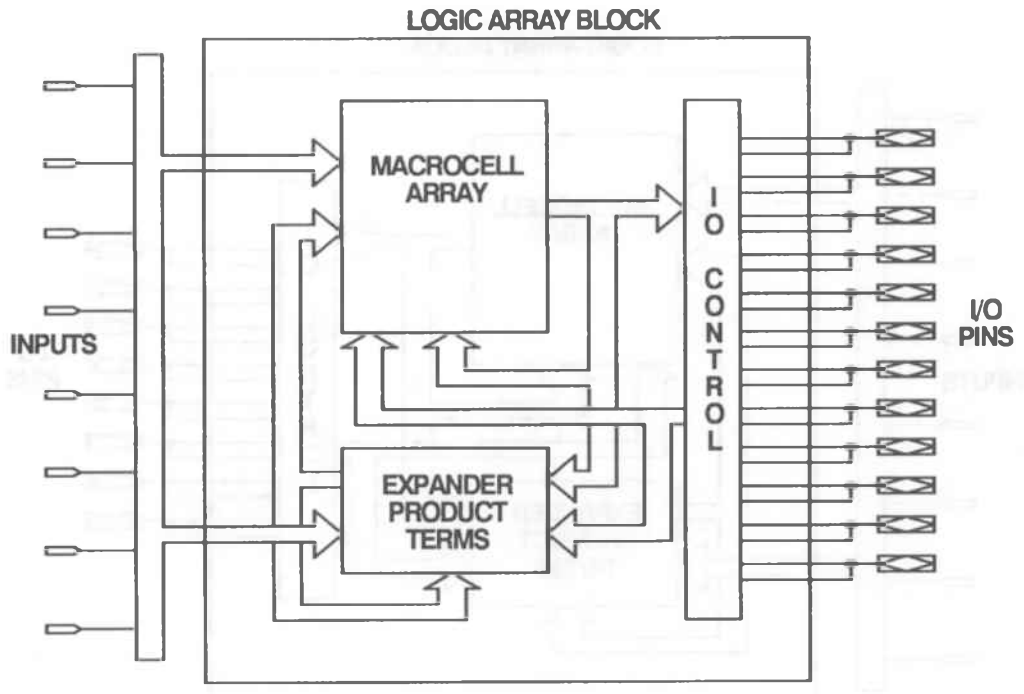
INPUTS	I/O PINS	LABS	MACROCELLS PER LAB	TOTAL MACROCELLS	EXPANDERS	PIA
8	16	1	32	32	64	NO

EPM5024

- High speed, high density MAX EPLD.
- Full MAX Macrocell features.
  —Programmable flip-flop or flow through latch
  —Asynchronous Clear and Preset
- 24 Macrocells, 48 Expanders in one LAB.
- 24 Pin 300 mil CDIP, PDIP packages.

Shown in Figure 10, the EPM5024 touts 24 Macrocells and 48 Expander Product Terms in its one Logic Array Block. As with all MAX EPLDs, there are 8 dedicated inputs, with one that may be configured as a synchronous clock line for high speed clocking applications. 12 of the 24 Macrocells may be connected to the 12 Bi-directional I/O pins on this device. Alternately, the tri-stateable I/O pins may be used as dedicated inputs, or Bi-directional I/O. This part features superior logic density in a 24 pin 300 mil DIP, or space-saving 28 pin JLCC.

Figure 10. EPM5024 Block Diagram



FEATURES

INPUTS	I/O PINS	LABS	MACROCELLS PER LAB	TOTAL MACROCELLS	EXPANDERS	PIA
8	12	1	24	24	48	NO

7

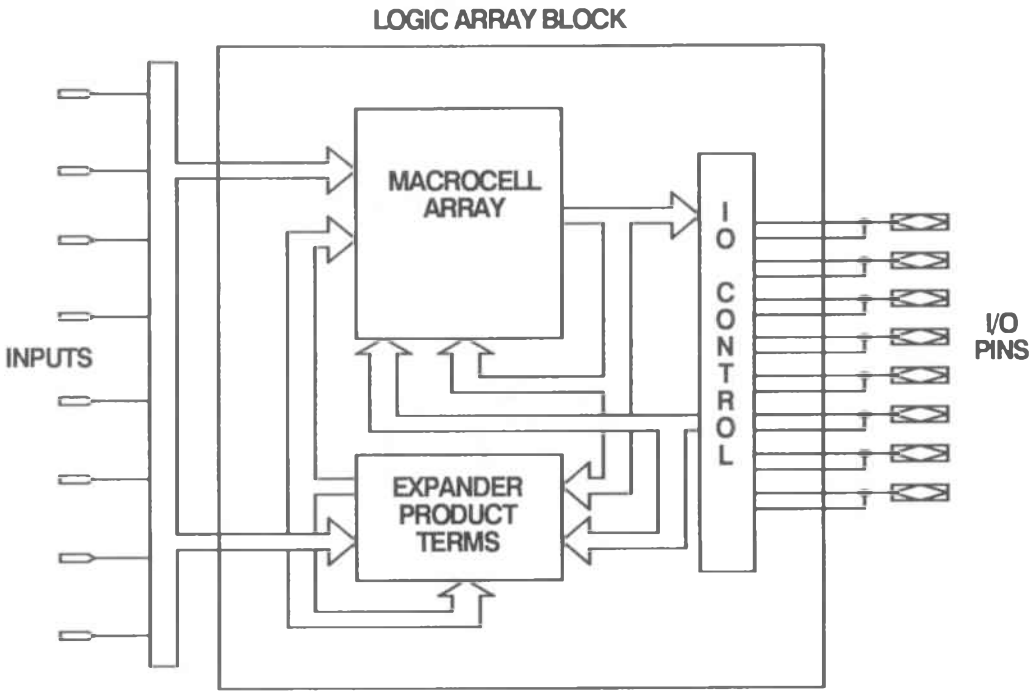
EPM5016

- 16 Macrocells and 32 Expander Product Terms in one MAX Logic Array Block.
- 8 dedicated inputs, 8 Bi-directional I/O pins.
- Synchronous or programmable clocking.
- Flow-through latch capability for fast latched applications.
- Available in 20 Pin CDIP and PDIP.

The EPM5016 provides 16 Macrocells in a 20 pin dip package. Like all members of the MAX family, it has 8 dedicated inputs. In its one LAB, there are 32 Expander product terms that may be used and shared by the product terms in the Macrocell array. The 16 Macrocells in the Macrocell Array feed 8 I/O pads. Refer to Figure 11.

This device is ideally suited for applications where generic 20 pin PLDs do not have the flexibility or density needed. Typical applications include state machines, fast latched address decoders, or large shift registers or counters, which a typical 20 pin device could not support. These are execellent examples of EPM5016 applications.

Figure 11. EPM5016 Block Diagram



FEATURES

INPUTS	I/O PINS	LABS	MACROCELLS PER LAB	TOTAL MACROCELLS	EXPANDERS	PIA
8	8	1	16	16	32	NO

MAX+PLUS
DEVELOPMENT SYSTEM
GENERAL DESCRIPTION

The Altera MAX+PLUS Development System represents a complete hardware and software solution for implementing designs into Altera's MAX (Multiple Array MatriX) family of EPLDs. MAX+PLUS is a sophisticated Computer Aided Design (CAD) system that includes design entry, design simulation, and device programming. Hosted on an IBM PC-AT or compatible machine, MAX+PLUS gives the designer the tools to quickly and efficiently implement complex logic designs. A block diagram is shown in Figure 12.

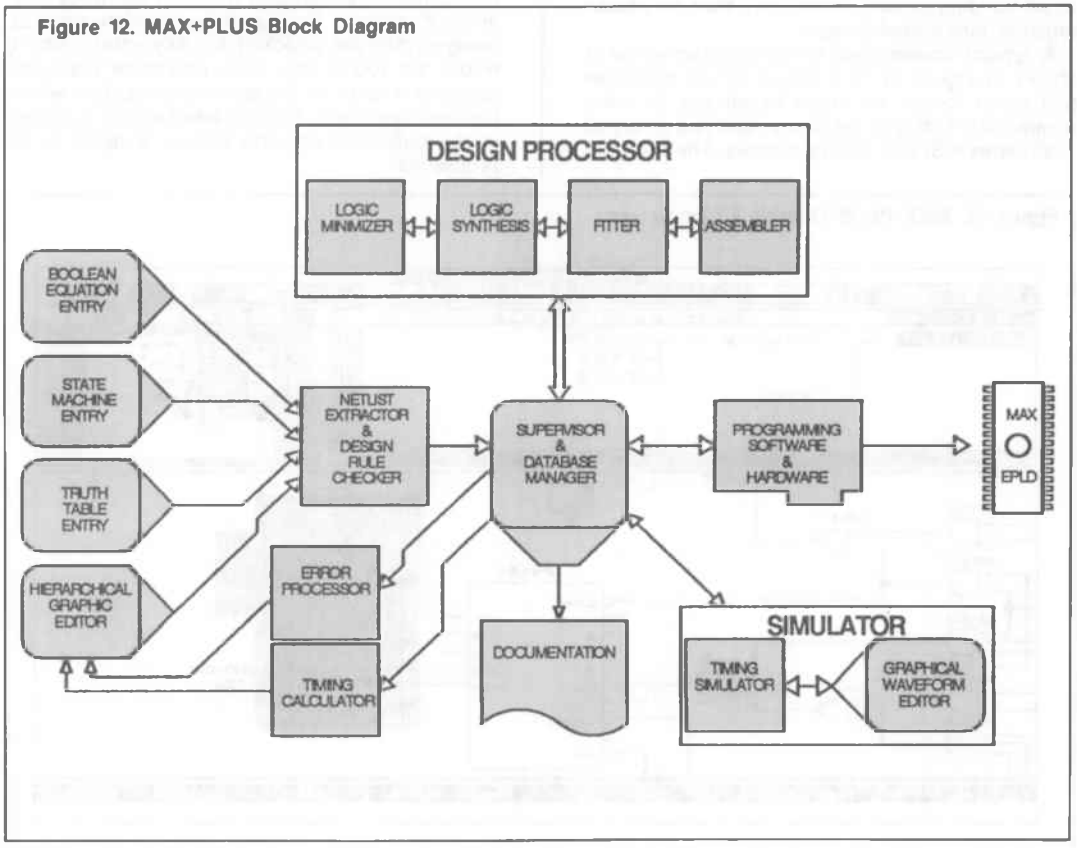
Designs are entered in MAX+PLUS using a hierarchical graphic editor. This editor has such features as multiple windows, multiple zoom levels, unlimited hierarchy levels, symbol editing, and a library of 7400 series devices in addition to basic SSI gate and register primitives. Also available is a Timing Calculator, in which the designer may pick two places in the schematic, and the software will display typical timing between those two points. Boolean Equation, Netlist, State Machine, and Truth Table entry mechanisms may be used in conjunction with the graphic editor, giving added flexibility to the design environment.

In addition to a hierarchical design environment, MAX+PLUS has a sophisticated processing engine to exploit the MAX architecture. MAX+PLUS uses an advanced logic synthesizer and heuristic rules to process a design into a file for programming and/or simulation.

MAX+PLUS features a powerful event-driven simulator which displays typical timing results in an interactive waveform editor display. In this waveform editor, input vector waveforms may be directly modified and a new simulation run immediately.

Unlike most design environments, MAX+PLUS is unified, with all sections controlled by the Supervisor and Data Base Manager. By unifying the software, MAX+PLUS can offer an automatic error locator. If a design rule has been violated, the error processor will list an error message, the

Figure 12. MAX+PLUS Block Diagram



probable cause, and pop the designer into the schematic to the exact node where the mistake was made.

DESIGN ENTRY

Design entry is easily accomplished with MAX+PLUS. MAX+PLUS provides multiple entry mechanisms, including traditional Boolean equation entry. Also available are State Machine and Truth Table entry, using a high-level state machine language. Because MAX EPLDs offer the designer large amounts of logic capability, Altera has created a Hierarchical Graphic Editor to ease the design process.

GRAPHIC EDITOR

The hierarchical design approach used by the graphic editor allows the designer to work with either a top-down or a bottom-up approach. The top down method allows the designer to start with a high level block diagram, and then move down and design each block individually. The bottom up method allows the simulation and verification of small building blocks, which may then be pieced together into a final design.

A typical screen shot of the graphic editor is shown in Figure 13. It is mouse driven and uses pull down menus or single keystrokes to enter commands. Aiding in the design task is a library of 7400 series MSI and SSI logic gates. The designer may use these and/or create his own custom symbols. Custom functions are easily created in the hierarchy by first designing the function. Then a symbol is made, which represents that schematic. In this way a custom function may be used in multiple places in the current design, or saved and used in subsequent designs.

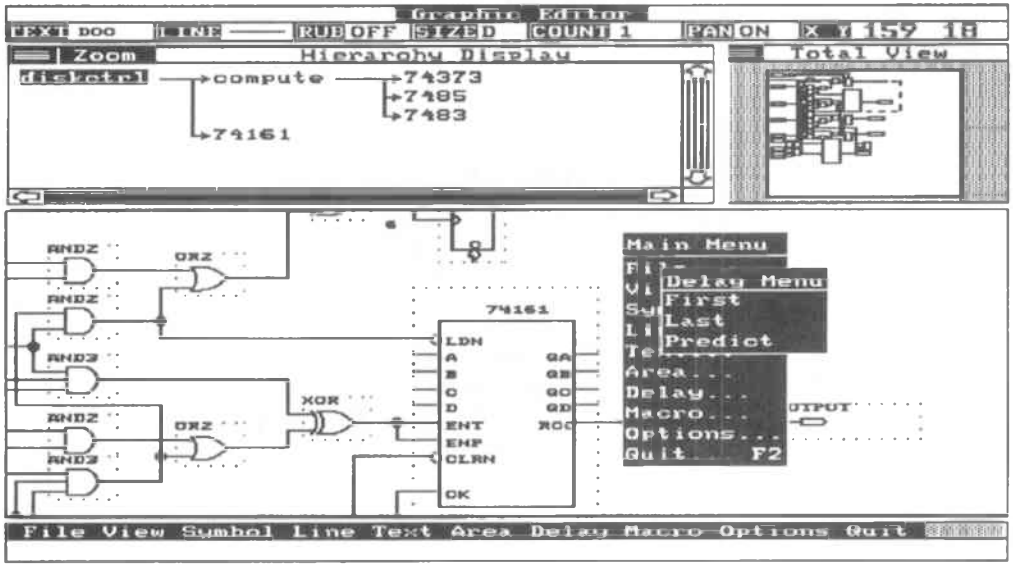
The function of any symbol created may be defined using graphic entry, state machine, Boolean, or truth table descriptions. This provides a wide range of flexibility for the designer, allowing Boolean equations to be combined with state machine entry in a hierarchical schematic.

The timing calculator within the graphic editor gives the designer instant feedback concerning timing delays inherent in a path. By placing two probes on different parts of the schematic, the designer immediately knows the worst case timing of the processed design. This is a valuable addition for design debugging and documentation.

DESIGN PROCESSOR

After the design is entered, a push of the mouse button invokes the powerful MAX+PLUS processor. First a netlist is extracted from the complete hierarchical design. During the extraction process, design rules are checked for any errors, and if errors are found, the error processor leads the designer directly to the schematic location where the error occurred. The extracted design is placed in the database, and the design is ready to be processed.

Figure 13. MAX+PLUS Graphic Editor Screen



The versatile MAX architecture, with its Expander Product Terms and mutual exclusivity, requires a dedicated processor to take optimal advantage of the MAX features, one that does much more than simplify logic. The logic synthesizer in MAX+PLUS uses several knowledge-based synthesis rules to factor and map logic onto the multi-level MAX architecture. It will then choose the mapping approach that ensures the most efficient use of the silicon resources. The synthesizer will also remove any unused logic or registers from the design.

The next module in the design processor is the fitter. Its function is similar to a placement and router used in semicustom gate arrays. Using heuristic rules, it takes the synthesized design and optimally places it within the chosen MAX EPLD. With the larger devices, it also routes the signals across the Programmable Interconnect Array, freeing the designer from interconnection issues.

TIMING SIMULATOR

Rounding out the software offering is a powerful timing simulator to aid in the verification and debugging of MAX designs. The simulator is a graphical, event driven software package that yields true, worst case timings based upon user-defined input vectors.

Waveforms may be viewed using a Graphical Waveform Editor, which allows graphical definitions and editing of input waveforms. The designer can define his input waveform using the mouse to draw the actual waveform as a function of time. There are also powerful waveform editing commands, all menu driven, to aid in the development of the input vectors. Such options as pre-defining, copying, and repeating waveforms are all available to the user. If graphical definition is not desired, there is a powerful vector description language for developing input vectors.

The simulator itself has all the capabilities one would expect from this type of design environment. Observing buried nodes, accessing flip-flop control inputs, and initializing and forcing nodes to specified values are all available within the timing simulator. The user may also specify breakpoints during the simulation itself, and execute subroutines dependant upon the breakpoints. All of these tools aid the designer in verifying and debugging the design, even before breadboarding.

The simulator also has advanced A.C. timing detection. The software will warn the user when set-up and hold times to flip flops are being violated, and when there is oscillation present in the simulation. Also, the user may define a minimum pulse width, in which any pulse within the design that is smaller than a certain size will be classified as a glitch and the designer will be informed.

SUPERVISOR AND ERROR PROCESSOR

All facets of the MAX+PLUS system are overseen by the Supervisor and Data Base Manager. By tying all of the software together, the designer has a unified operational environment. All the software has the same "look and feel", so that complex commands and languages are not needed.

Automatic error processing is an added benefit of this approach. If an error occurs during the processing of the design, the software will automatically tell the user what the error is, and the probable cause. Then, by pressing a single key, the software will automatically go to the schematic in the graphic editor and pinpoint the location of the error.

MAX Product Overview Rev 1.0

NOTES

ADDITIONAL INFORMATION	PAGE NO.

8

FEATURES

- **Third party support for:**
  - **—Schematic Capture**
  - **—Design Processing**
  - **—Device Programming**
- **Data I/O programming hardware family and pinout codes.**

INTRODUCTION

The third party support tools listed below have been reviewed by the Altera Applications Department and appear to meet the specifications published by their manufacturers. Similar Altera products are listed for completeness.

Altera can accept no responsibility for the suitability or accuracy of third party development software or programming hardware. Altera has developed a procedure, described in the Production Programming Specification section of this Handbook, to minimize the risk of improperly programming large quantities of devices during production.

SCHEMATIC ENTRY

Figure 1 shows EPLD schematic capture software available. Each schematic capture package must create an Altera Design File (ADF) to be compiled by A+PLUS (Altera Programmable Logic User System). Some vendors do not support Altera's TTL MacroFunction library.

DESIGN PROCESSING

Reviewed design processing software is provided by Altera and three third party vendors: Data I/O (ABEL), P-CAD (CUPL), and Intel (iPLDS). Third party vendors may not support all EPLD architectures or device types. Please check appropriate third party literature for details.

When using CUPL or ABEL and designing with the EP1210, EP1800, or EP1810, knowledge of the device architecture is required. These EPLDs contain both local and global macrocells. As a result, proper pin assignments is essential. Figure 2 details design processing support and the Altera devices which are supported.

PROGRAMMING HARDWARE

Device programming hardware is available from several third party programmers: Data I/O, Digilec, Kontron, and Stag. Figure 3 shows the support each vendor provides.

Data I/O provides the most comprehensive Altera device support. Data I/O supports DIP and J-leaded packages for the EP310, EP320, EP600, EP900, EP1210, and EP1800 with three different programmer models: Model 29B, Model 60 and the Unisite 40. Data I/O assigns each Altera device a family and pinout code, common to all software-controlled Data I/O programmers. Figure 4 lists the respective family and pinout codes for each device.

Figure 1. Schematic Capture Support

VENDOR	SOFTWARE	ADF CONVERSION	MACROFUNCTION SUPPORT	EPLD SUPPORT
ALTERA	LOGICAPS	YES[1]	YES	EP310, EP320, EP600, EP610, EP900, EP910, EP1210, EP1800, EP1810, EPB1400
DATA I/O-FUTURENET	DASH4	YES[1]	NO	EP310, EP320, EP600, EP610, EP900, EP910, EP1210, EP1800, EPB1400
P-CAD	PC-CAPS	YES[1]	NO	EP310, EP320, EP600, EP900, EP1210, EP1800
VIEWLOGIC	WORKVIEW	YES[2]	YES	EP310, EP320, EP600, EP900, EP1210, EP1800

1. Altera Supplied
2. Vendor Supplied

Figure 2. Design Processing Software

VENDOR	SOFTWARE	REVISION	DEVICES SUPPORTED
ALTERA	A+PLUS	5.03	EP310, EP320, EP600, EP610, EP900, EP910, EP1210, EP1800, EP1810, EPB1400
	MCMAP	1.0	EPB2001
	SAM+PLUS	1.01	EPS444, EPS448
DATA I/O	ABEL	3.0	EP310, EP320, EP600, EP900, EP1210, EP1800
P-CAD	CUPL	2.15	EP310, EP320, EP600, EP900
INTEL	iPLS	1.5	EP310, EP320, EP600, EP900, EP1210, EP1800

Figure 3. Programming Hardware

VENDOR	UNIT	DEVICES SUPPORTED*
ALTERA	PLE3-12(A)	EP310D, EP320D, EP1210D
	PLE3-12 (A) and Appropriate Adapter	All Altera Devices and All Packages
DATA I/O	Model 29B with:	
	303A-009 Adapter (Rev V03)	EP310D
	303A-010 Adapter (Rev V03)	EP900D/J, EP1210D/J
	303A-011A Adapter (Rev V02)	EP310D, EP320D, EP600D
	303A-011B Adapter (Rev V02)	EP600J
	Model 60 (Rev V07)	EP310D, EP600D
	Unisite 40 (Rev 1.7)	EP320D, EP600D/J, EP900D/J, EP1210D/J, EP1800J
DIGELEC	Model 860	EP310D, EP320D, EP600D, EP900D, EP1210D, EP1800J
KONTRON	Model EPP80	EP310D, EP320D, EP600D, EP900D, EP1210D
STAG	PPZ	EP310D
	ZL30A	EP310D

* D—DIP PACKAGE (PLASTIC AND CERAMIC)
J— J-LEAD CHIP CARRIER (PLASTIC AND CERAMIC)

Figure 4. Family and Pinout Codes

Device	Package	Family	Pinout
EP310	DIP	44	50
EP320	DIP	44	95
EP600	DIP	26	59
EP900	JLCC, PLCC	026	759
EP900	DIP	26	96
EP900	JLCC, PLCC	026	796
EP1200	DIP	26	97
EP1210	DIP	44	97
EP1210	JLCC, PLCC	044	797
EP1800	JLCC, PLCC	26	9A

AB21D Rev 1.0

8

ALTERA PRODUCTION PROGRAMMING PROCEDURE

INTRODUCTION

This document describes procedures and guidelines to program Altera EPLDs with Altera PC based programming hardware and software. The following assumes the user is familiar with PCs and the DOS operating system.

Failures during programming should be noted. If more than 3 programming failures occur consecutively or if a total of 5 devices in a lot fail, halt programming and contact Altera Applications at (408) 984-2805 x102 for assistance.

Altera holds no liability for EPLD programming failures which occur outside the quantity limits listed in this document.

REQUIREMENTS

SYSTEM REQUIREMENTS:

IBM XT or AT personal computer, or compatible, with:

640 Kbytes of memory
Color or Monochrome Display
1 Mbyte of free disk space
Empty card slot
DOS version 2.0 or later

Programming should be performed with PCs which operate at 8 MHz or below. Random programming failures may result when operating PC at 10 MHz or greater.

HARDWARE REQUIREMENTS:

Altera Programming Card
— Version LP3 or LP4

Altera Programming Module
— PLE3-12(A) (Directly supports EP310, EP320, and EP1210 DIP EPLDs)

Altera Programming Adaptors

— Dip	J-Lead	PGA
PLED600	PLEJ600	PLEG1800
PLED900	PLEJ900	
PLED448	PLEJ1210	
PLED1400	PLEJ1800	
	PLEJ448	
	PLEJ1400	

Software Requirements:

LOGICMAP version 4.5 or later needed for LP3
LOGICMAP version 5.01 or later needed for LP4

Control EPLDs—Erasable Windowed Version

Ten EPLDs which will be used to test all required software and hardware before proceeding with production programming. These devices should be erased for at least 30 minutes using a standard EPROM eraser.

Software Back-ups

Make back-up copies of all JEDEC file(s) on floppy disks.

PREPARATION

A. Install the Altera software and hardware. (See A+PLUS Users manual for instructions).

B. Set the current directory to the area where the Altera software resides. (Usually this directory is called APLUS).

C. Copy the JEDEC file(s) you wish to program into this directory.

SYSTEM VERIFICATION

A. Connect the programming module into the Altera programming card.

B. Invoke the programming software by typing LOGICMAP.

At this point the software automatically performs a self test to the hardware. If any gross hardware problems are encountered, the software will display "SELF TEST FAILED". If this message appears, call Altera Applications for assistance.

C. Program the 10 control devices (window versions) with the same JEDEC file which will be used for the production EPLDs.

To program:

1. Move box cursor to PROGRAM DEVICE and hit the ENTER key.
2. Type in the appropriate JEDEC file (.JED extension is not necessary) and hit ENTER.
3. Type in the appropriate JEDEC and package designation (e.g. EP1800J, EP900, EP600J) and hit ENTER.
4. The software will display the HEADER information contained in the JEDEC file as well as the current status of the TURBO AND SECURITY PROTECT bits. If the user wishes to change these conditions before pro-

gramming, refer to Appendix A for instructions.

5. Insert an erased EPLD into the programming module. Ensure Pin 1 is correctly positioned. All devices are shipped erased from the factory.
6. Hit the SPACE BAR to program the device.
7. LOGICMAP begins device programming and verification. When completed, LOGICMAP indicates if programming was successful, or if programming errors occurred.
8. If the device has been successfully programmed, remove the EPLD from the programming module, insert another erased device and hit the SPACE BAR to program. If errors occur during programming, go to the **PROGRAMMING ERRORS** section.
9. Continue steps 6 thru 8 for all 10 control devices.
10. If all 10 devices program successfully, go to the **PRODUCTION PROGRAMMING** section.

NOTE: When programming J-leaded EPLDs, almost all programming errors are caused by poor contact between the device pins and the socket. J-leaded adaptors should be visually inspected after every 500 programming cycles, to ensure programming socket leads make good electrical contact. Straighten bent programming socket leads with a pair of tweezers. If the programming socket is irreversably damaged, it should be scrapped and replaced.

PRODUCTION PROGRAMMING

To Program devices:

A. Invoke programming software by typing LOGICMAP.

B. Move box cursor to PROGRAM DEVICE and hit the ENTER key.

C. Type in the name of the JEDEC file to be programmed, and hit ENTER (.JED extension is not necessary).

D. The software will display the HEADER information contained in the JEDEC file as well as the current status of the TURBO and SECURITY PROTECT bits. If one wishes to change these conditions before programming, refer to Appendix A for instructions.

E. Insert a new EPLD into the programming module. Ensure Pin 1 is correctly positioned.

F. Hit the SPACE BAR to program.

G. LOGICMAP begins device programming and verification. When completed, LOGICMAP indicates if programming was successful, or if programming errors occurred.

H. If the device has been successfully programmed, remove the EPLD from the programming module, insert another erased device and hit the SPACE BAR to program. If errors occur during programming, go to the **PROGRAMMING ERRORS** section.

I. Continue step F thru H for all production units.

PROGRAMMING ERRORS

If a programming error occurs:

A. Record error message displayed by LOGICMAP.

B. Remove device and re-insert into socket to ensure good contact is made.

C. Re-program device.

D. If device fails again, record error message and label both device and error message with same identification.

E. Try a new device, if it fails on first programming attempt, repeat steps A-D.

NOTES:

If 3 devices fail consecutively, stop programming and contact Altera Applications.

If greater than 5 units fail in a lot, stop programming and contact Altera Applications.

APPENDIX A

SECURITY PROTECT AND TURBO BIT CONTROL

Prior to programming EPLDs, the Verify Protect and Turbo bits may be altered.

SECURITY PROTECT BIT

Programming the Verify Protect bit ensures the design implemented cannot be retrieved or copied. Once the verify protect bit has been programmed, the EPLD can no longer be examined or verified with Altera or any third party programming equipment.

SECURITY = ON	EPLD is verify protected
SECURITY = OFF	EPLD is not verify protected (default)

TURBO BIT

Programming the Turbo bit disables the low standby power mode of the EPLD. Unless zero standby power is required, it is always recommended to program the Turbo bit.

TURBO = ON	Turbo bit programmed
TURBO = OFF	Turbo bit not programmed (Zero Power Operation)

MODIFYING THE TURBO OR SECURITY PROTECT BITS

To change either Turbo or Security Protect:

1. From the LOGICMAP main menu select EDIT JEDEC FILE
2. Type JEDEC file name (.JED extension is not required) and hit ENTER.
3. Type DEVICE name (EP1800J, EP900, EP600J, etc.) and hit ENTER.
4. The HEADER information in the JEDEC file (if any) will be displayed. Press PgDn key for Device Level Window.
5. At the top of the Device Level Window is the current status of the SECURITY and TURBO bits. To change the SECURITY bit, hold the Alt key down and press function key F9. To change the TURBO bit, hold the Alt down and press function key F10.

AB56A Rev 2.0

FEATURES

- **Circuit board real estate savings motivate use of J Leaded packages.**
- **Types of sockets for J-Leaded packages.**
- **Carrier boards for use with wire wrap panels and J-Leaded packages.**
- **How to select an appropriate socket for your EPLD.**

INTRODUCTION

EPLDs solve two problems designers face today. They reduce both the real estate and power required to implement digital systems. Altera offers J-leaded versions of its EPLDs to further reduce the real estate demands of the system. These small packages are generally intended for surface mounting.

Despite these advancements, surface mounting is still considered a young technology, although considerable research and use have proved it's viability. Most industrial applications still use traditional through-hole soldering. Surface mount assembling has its own demands on the development and manufacturing processes. It requires different CAD symbols for PCB layout, different test and reliability procedures for buried vias within PC boards, and a different soldering process for production (vapor phase versus wave solder). Bonding EPLDs to a PC board removes the possibility of convenient erasure and reprogramming; of particular importance during development.

A popular compromise to gain the size advantage of the JLCC without the corresponding headaches of surface mounting is to socket the components. One can still use conventional mounting techniques, either through hole soldering to a PC board, or mounting in a socketed carrier board for wire wrap.

MECHANICAL CONSIDERATIONS FOR SOCKETS

The three principal concerns in the selection of a socket are size, contact mating force, and ease of use.

The size of the socket varies with the vendor. Socket size usually varies between "production" type and "test and burn in" type. Production sockets are small, low cost units that do not vary much in size from vendor to vendor. Typical sizes are shown in Table 1.

Table 1.

EPLD Package Dimensions

28 pin	.700 x .700 (inches)
44 pin	.900 x .900 (inches)
68 pin	1.20 x 1.20 (inches)

The test and burn-in socket is a much larger, more expensive socket tailored for easy insertion and extraction with no stress on the component's leads. Their large size makes them impractical for production, although they may find use during development.

Contact mating force is important for two reasons. (1) The mating force provides the electrical connection, and insufficient mating force will lead to intermittent contacts. A marginal contact may also fail as a result of corrosion or oxidation between the leads of the socket and component over time. (2) Vibration may cause a system failure if the sockets have insufficient force to retain the components. A normal force of 150g at the pin is considered to be the minimum force required to provide reliable electrical contact for tin-lead sockets mating with Altera's tin-lead EPLD pins.

Ease of use is another concern, particularly since reprogrammable EPLDs will be swapped in and out of the sockets during the development cycle. Test and burn-in sockets are the easiest to work with during development. Unfortunately, they are generally large and have different pin geometries from production sockets. When used during development, the PC board may have to be laid out again for the production sockets. Production sockets have different mechanisms for chip extraction; varying from having a hole in the PC board to drive the component out with a nail (not recommended) to low cost custom extraction tools that slip over the top or into the corners of the socket.

Vendors may provide additional information about their products, such as material selection, prevention of solder ingress during wave soldering, or lead shape. Altera recommends qualifying sockets, just as with other components, before committing to a given vendor.

PACKAGING OPTIONS FOR WIRE WRAP APPLICATIONS

Wire wrap applications require a through-hole mount incompatible with the J-leaded package. Further, the sockets specified do not typically mechanically conform to most wire wrap panels.

Wire wrap card have machined receptacles in rows with 100 mil spacing between receptacles and 300 mil spacing between rows.

Carrier boards are an effective way to bridge the gap. By mounting a socket to a carrier board a small real estate penalty is paid to gain the convenience of wire wrap. Some carrier boards have signal routing with shorter paths, or 45 degree bends to minimize signal reflection.

DIMENSIONS OF EPLDs

Table 2 specifies the contact distance for Altera's EPLDs (in mils).

Table 2.

EPLD Contact Distances

EPLD	PIN COUNT	CONTACT DISTANCE MIN.	MAX.
EP1800J, EP1810J	68	970	995
EP1800L, EP1810L	68	985	990
EP1210J	44	672	688
EP1210L	44	685	695
EP900J, EP910J	44	672	695
EP900L, EP910L	44	685	690
EPB1400J	44	672	688
EPB1400L	44	685	695
EP600J, EP610J	28	475	495
EP600L, EP610L	28	485	490
EPS448J	28	475	495
EPS448L	28	485	490

WHO TO CONTACT:

Below are the corporate offices of the vendors noted in this Application Brief. Contact them for additional information.

AMP Inc.	(717) 569-0100
Augat	(617) 588-6110
Burndy Corp.	(203) 838-4444
Mupac Corp.	(617) 588-6110
3M	(214) 647-0392

INFORMATION CONTAINED IN THIS APPLICATION BRIEF IS BASED UPON INFORMATION PROVIDED TO ALTERA BY VARIOUS VENDORS, AND IS BELIEVED TO BE ACCURATE. ALTERA ASSUMES NO LIABILITY FOR THE USE OF THIRD PARTY PRODUCTS NOTED IN THIS APPLICATION BRIEF.

AB46 Rev 2.0

PRODUCTION SOCKETS FOR EPLDs

68 PIN JLCC/PLCC

VENDOR/ PART NUMBER	CONTACT DISTANCE (MILS) MIN	MAX	NORMAL FORCE (IN g. FOR 980 mil CONTACT DISTANCE)	COMMENTS
3M 2-0068-06234-050-038-077	942	997	340	GOOD CONTACT DISTANCE SIMPLE EXTRACTION MECHANISM.
BURNDY QILE68P-410T	962	995	290	GOOD CONTACT DISTANCE THOUGH DIFFICULT TO EXTRACT COMPONENTS.
AMP 821574-1 (TIN) 641749-1 (GOLD)	985 NOT AVAILABLE	995	NOT AVAILABLE NOT AVAILABLE	*NOT RECOMMENDED* POSSIBLE INTERMITTENT CONTACT DUE TO CONTACT DISTANCE FOR ALTERA PACKAGES.
T & B AINSLEY PCC-068T-01	NOT AVAILABLE		NOT AVAILABLE	*NOT RECOMMENDED* GENERALLY INTERMITTENT WITH ALTERA PACKAGES.

48 PIN JLCC/PLCC

VENDOR/ PART NUMBER	CONTACT DISTANCE (MILS) MIN	MAX	NORMAL FORCE (IN g. FOR 680 mil CONTACT DISTANCE)	COMMENTS
3M 2-0044-06232-050-038-077	650	695	340	GOOD CONTACT DISTANCE.
BURNDY QILE44P-410T	660	695	290	GOOD CONTACT DISTANCE.
AMP 821575-1 (TIN)	685	695	NOT AVAILABLE	*NOT RECOMMENDED* POSSIBLE INTERMITTENCY DUE TO CONTACT DISTANCE FOR ALTERA PACKAGES.

28 PIN JLCC/PLCC

VENDOR/ PART NUMBER	CONTACT DISTANCE (MILS) MIN	MAX	NORMAL FORCE (IN g. FOR 480 mil CONTACT DISTANCE)	COMMENTS
BURNDY QILE28P-410T	460	495	290	GOOD CONTACT DISTANCE.
AMP 821581-1 (TIN)	485	495	NOT AVAILABLE	*NOT RECOMMENDED* POSSIBLE INTERMITTENCY DUE TO CONTACT DISTANCE FOR ALTERA PACKAGES.

TEST AND BURN IN SOCKETS

VENDOR/ PART NUMBER	DIMENSIONS	EPLD SUITABILITY	COMMENTS
3M			
268-6345-00 244-6343-00	1.38" x 1.38" 1.08" x 1.08"	EP1800J, EP1810J, EP900J, EP910J, EP1210J, EPB1400J	VERY SMALL FOOTPRINT FOR A TEST AND BURN-IN SOCKET. VERY EASY TO INSERT AND EXTRACT COMPONENTS. SIDEWIPE CONTACTS SHOULD BE CHECKED FOR GOOD CONNECTION.
AMP			
821682-6 2-821682-4 3-821682-0	1.385" x .995" 1.085" x .695"	EP1800J, EP1810J, EP900J, EP910J, EP600J, EP610J, EP1210J, EPB1400J, EPS448J	VERY SMALL FOOTPRINT ALLOWS HIGHLY INTEGRATED DEVELOPMENT. SIMPLE EXTRACTION MECHANISM. SIDEWIPE CONTACTS SHOULD BE CHECKED FOR GOOD CONNECTION.

WIRE WRAP ADAPTORS/PLCC ADAPTORS

VENDOR/ PART NUMBER	DIMENSIONS	EPLD SUITABILITY	COMMENTS
ANTONA			
ANC-9068P ANC-9068C	1.90" x 2.10" 1.90" x 2.10"	EP1800G, EP1800J, EP1810G, EP1810J	ADAPTOR BOARD WITH WIRE WRAPPABLE POSTS FOR EITHER PGA OR J-LEADED.
ADVANCED INTERCONNECTIONS			
1586-28 1586-44 1586-68	.600" x .600" .800" x .800" 1.10" x 1.10"	EP1800J, EP1810J, EP900J, EP910J, EP600J, EP610J, EP1210J, EPB1400J, EPS448J	THESE ADAPTORS WILL CONVERT A J-LEADED PINOUT INTO A PGA PINOUT.

CARRIER BOARDS

VENDOR/ PART NUMBER	DIMENSIONS	EPLD SUITABILITY	COMMENTS
AUGAT		EP1800J, EP1810J,	CARRIER BOARDS FOR STANDARD WIRE WRAP PANELS (.300" BETWEEN ROWS). THE CARRIER BOARDS ALREADY HAVE A SOCKET INSTALLED (SURFACE MOUNTED).
SSM068-AA-B1PA	1.185" x 1.94"	EP900J, EP910J,	
SSM044-AA-B1PA	1.185" x 1.25"	EP600J, EP610J,	
SSM028-AA-B1PA	0.800" x 1.25"	EP1210J, EPB1400J, EPS448J	
MUPAC			FOR MUPAC'S HIGH DENSITY WIRE WRAP PANELS. THIS NUMBER IS FOR ADAPTOR BOARD ONLY — SOLDER A 68 PIN PRODUCTION SOCKET ON.
367-2751-01	1.50" x 1.70"	EP1800J EP1810J	
367-4500-01	1.50" x 1.70"	EP1800J EP1810J	AS ABOVE WITH 68 PIN SOCKET INSTALLED. (SOCKET VENDOR UNKNOWN)
ROBINSON NUGENT			
APB-ICC-68	1.375" x 3.125"	EP1800J EP1810J	CARRIER BOARD FOR UNIVERSAL WIRE WRAP PANEL, BUT WITH A LARGE FOOTPRINT. ORDER BARE BOARD AND SOLDER ON A PRODUCTION SOCKET.

FEATURES

- **Schematic design techniques.**
- **Solution to "Too many product terms" error.**
- **Solution to Illegal Clock, Clear, or Output Enable errors.**

INTRODUCTION

Altera saw a need for high level CAD tools to take advantage of high density EPLDs. Entering designs for EPLDs using Boolean equation methods became time consuming. Logicaps, a schematic capture program, and the associated libraries were created to allow the design input using familiar TTL MSI and SSI components. Powerful A+PLUS software takes this TTL schematic and yields a programmed part.

In the process of taking the schematic and fitting it into a part, minor changes may be required to allow the design to match the EPLD architecture. One architectural restriction is the complexity of combinatorial logic that may be placed into a macrocell. Problems may arises when a network of combinatorial logic feeding an output primitive becomes too complex to fit in the macrocell containing the logic.

The A+PLUS software indicates this problem with the following error messages:

"***INFO-FIT—Too many Pterms for this macrocell"

"***INFO-FIT—Too many Pterms for any macrocell"

"***ERR-EXP—Equation will expand too large"

"***INFO-FIT—Illegal inversion of <resource> input"

"***INFO-FIT—Too many pterms for <resource> input"

This Application Brief outlines procedures to correct these errors. An understanding of EPLD architecture is assumed. For a complete EPLD architectural discussion see the DataBook.

DESIGN FITTING

The procedure for solving fitting errors is outlined in the flow chart shown in Figure 1. After determining the error was caused by too many product terms or an illegal Clock, Clear, or Output Enable, the designer must locate where it occurred in the design.

A+PLUS locates the error by giving the name of an output primitive (Altera defined output symbol) in the schematic. If the output node was not named by the user, A+PLUS will name the node by either giving the X-Y position in the schematic or by giving the symbol number of a TTL MacroFunction within the design (see the Error Location section of this brief).

The next step is to trace the network of logic driving the listed node. The network is all the combinatorial logic leading to the output primitive. Rules for tracing a network are discussed in the Network Location section of this brief.

After isolating the error, there are two possible approaches to fit the design. The first approach is to optimize the logic, allowing it to fit within a single macrocell. The second approach is to separate the logic and use two macrocells.

LOGIC OPTIMIZATION

One way to optimize the logic is by switching to a more efficient flip-flop. Toggle flip-flops, for

Figure 1. Flowchart for solving a fitting error.

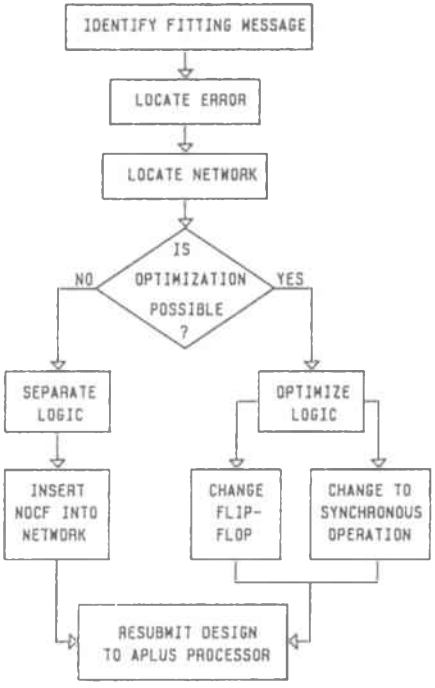


Figure 2a. Design Example 1—*This design yields the fitting error shown in Figure 2b.*

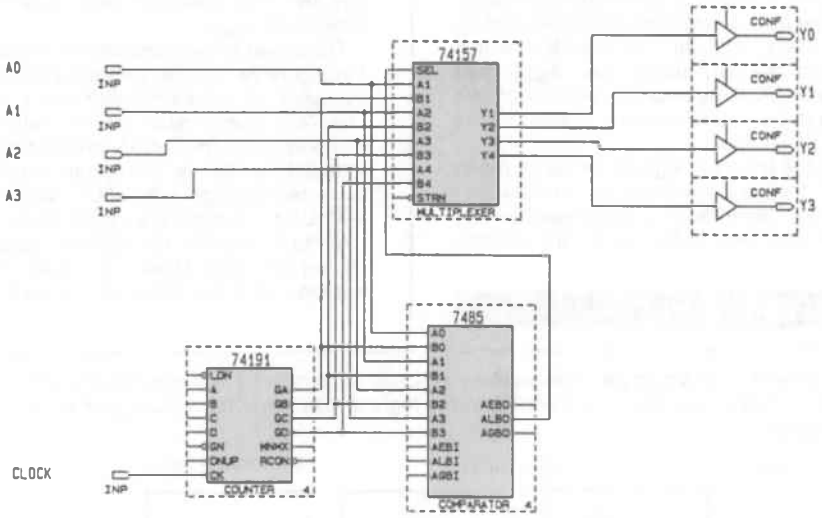
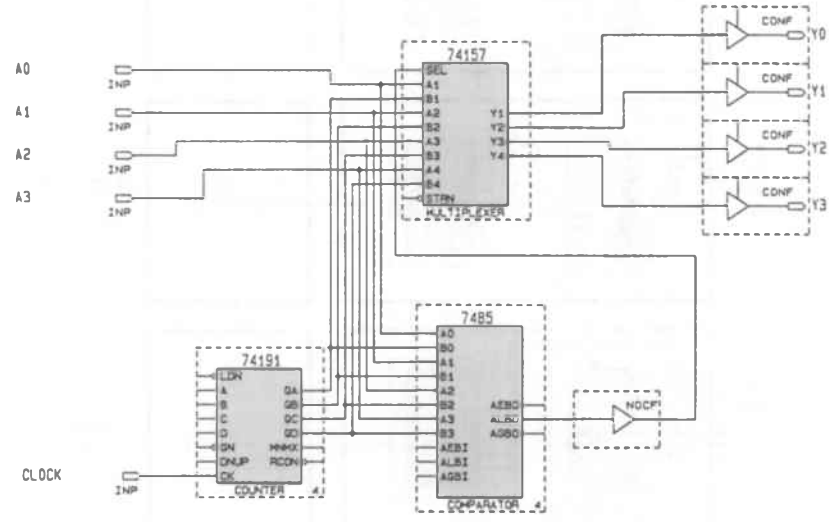


Figure 2b. Example 1 Error Message—*Error message from Example 1.*

```
***INFO-ADP-LEF analyzed
***INFO-FIT-Too many (19) pterms for any macrocell: "Y0"
***INFO-FIT-Too many (11) pterms for any macrocell: "Y1"
***INFO-FIT-A+PLUS is unable to fit this design
***INFO-FIT-No fit possible
```

Figure 2c. Example 1 Solution—*Example 1 after solving the fitting error by inserting an NOCF primitive after the 7485.*



example, are a more efficient for counters than D-type registers. Switching to toggle flip-flops or to a MacroFunction which uses toggle flip-flops will allow multi-bit counters to fit into one macrocell per bit. The MacroFunctions named '4COUNT', '8COUNT', '74161T', and all TTL counters which have a 'T' after the part number use toggle flip-flops (also see EPLD Application section of this Handbook and the MacroFunction section of the A+PLUS User's Manual).

Optimizing the logic for register clocks or register clears is done by switching to synchronous operation. This technique is discussed in the "Clock, Clear, and Output Enable Errors" section of this brief.

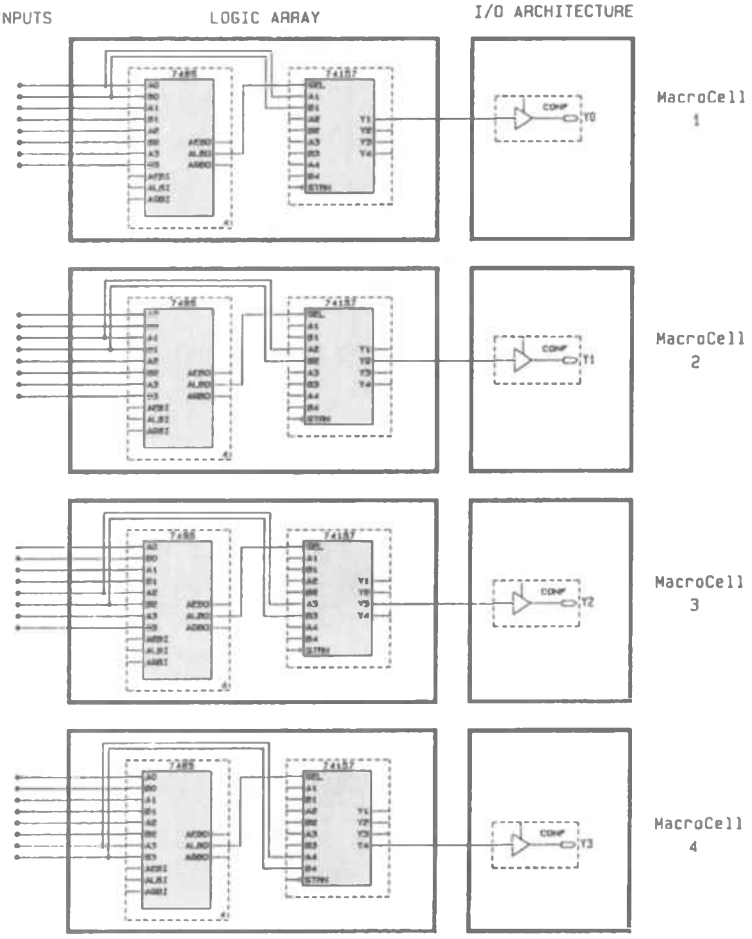
LOGIC SEPARATION

If logic optimization is not possible, the second option is dividing the logic network in half and placing each half in a separate macrocell. In this way, two macrocells are used to build a complex network of logic.

The power of logic separation is seen in Example 1 in Figure 2a. The design uses an EP610 to choose the lesser of a 4-bit number and a 4-bit counter. The 7485 comparator selects between the two numbers and the 74157 multiplexer sends the selected number to the output. Logicaps creates the Altera Design File (.ADF) which is used by A+PLUS to compile the object code.

A+PLUS signals the design does not fit by issuing the error shown in Figure 2b. The error message tells the designer the logic driving the

Figure 3a. Allocation of Logic into Macrocells—*The allocation of logic into macrocells before separation of Example 1 (See Figure 2a). The INPUTS to the logic arrays come from input pins or feedback from other macrocells.*



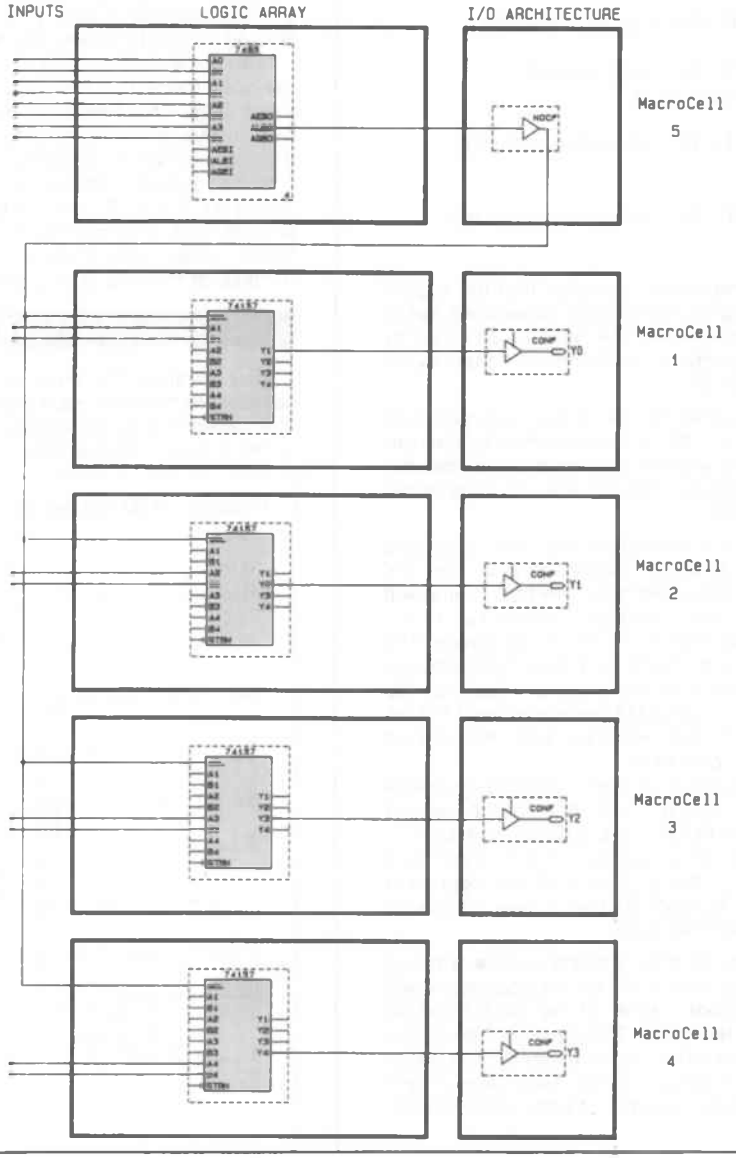
output primitives labeled Y0 and Y1 is too complex to fit into a single macrocell.

Figure 2c shows the problem is removed by placing a No Output Combinatorial Feedback (NOCF) design primitive after the output of the comparator. After inserting the NOCF, processing through A+PLUS is successful.

The effect of placing a NOCF primitive after the comparator is shown in Figure 3. Figure 3a shows how the logic was distributed between macrocells before inserting the NOCF. A+PLUS is unable to fit both the 7485 and the 74157 into the same macrocell. Figure 3b shows that the NOCF used a separate macrocell to implement the logic within the comparator. The feedback for NOCF is then fed to the select input of the multiplexer. By isolating the comparator, the logic is reduced in the macrocells leading to Y0 through Y3.

Separating the logic in this way has slowed propagation of the signal. An input to the comparator must pass through two logic arrays before reaching the final output primitive.

Figure 3b. NOCF Insertion—*Isolating of the 7485 by inserting a NOCF (See Figure 2c).*



ERROR LOCATION

A+PLUS assists in making design changes by indicating where in the schematic logic fitting is required. A+PLUS indicates the error location by listing one of the following error messages on the screen and in the Error log file (.LOG).

***ERR-EXP—Equation will expand too large.

***ERR-EXP—Left-hand side of equation is <signal name>.

***ERR-EXP—Can't expand equation.

***INFO-FIT—Too many pterms for <resource> input.

***INFO-FIT—Too many pterms for this macrocell.

***INFO-FIT—Too many pterms for any macrocell.

Each of these errors indicates that the logical network driving the listed node name does not fit into a single macrocell. The node name given by A+PLUS will locate the node on the schematic in one of three ways:

1. The node name may be a user-defined node. All outputs or internal nodes named by the user will be listed with the user name. The designer may simply locate that name on the node within the schematic.
2. For nodes not named by the user, LogiCaps generates a name which reflects the X-Y position in the schematic. The node name will be of the form ".pxxxyyy" where xxx is the X-coordinate and yyy is the Y-coordinate. The p is used by A+PLUS to differentiate between multiple sheets of schematics. The first file listed to the FILENAME prompt of the A+PLUS Design Processor will have p=0; the second will be p=1, and so on.

For example, the node name "1037008" indicates the node at location X=37, Y=8 in the second file given to A+PLUS. In LogiCaps, the current position of the cursor appears in the upper right hand corner of the screen and the command <W>INDOW <M>OVE 37,8 will cause the cursor to quickly locate the node.

3. The node name may indicate a node within a MacroFunction. In this case the node name will give the symbol number of the MacroFunction containing the node. The name will be of the form ".pMmmmNn" where mmm is the symbol number and n is an internal node name. The p again indicates the order of entry into A+PLUS.

For example, the node name ".0M004N1" indicates a node within MacroFunction number 4 in the first schematic given to A+PLUS. To locate the MacroFunction, simply load the first file into LogiCaps and type the command <S>YMBOL <F>IND 4 which will cause the cursor to indicate the MacroFunction.

Although there is no direct access to a node within a MacroFunction, the internal logic may be viewed by turning to the Altera Design Library (MacroFunctions) in the A+PLUS Reference Manual.

In all cases, the node in the error message is driven by the output or feedback from a registered or combinatorial output primitive. It is the logic leading to this primitive that requires fitting.

In Example 1, Figure 2b, the error message gave the node name Y0 which is driven by a Combinatorial Output No Feedback (CONF) output primitive. In this case the input to the CONF must be divided into two macrocells.

If the output primitive is a register, the data inputs (D, T, J, K, S, or R) to that register must be divided into two macrocells. Errors relating to Clock, Clear, and Output Enable signals are handled at the end of this brief.

NETWORK LOCATION

After locating the error in the schematic, the network demanding separation must be traced. The network is all combinatorial logic leading up to the output primitive listed in the error.

Figure 4. 74157 Schematic

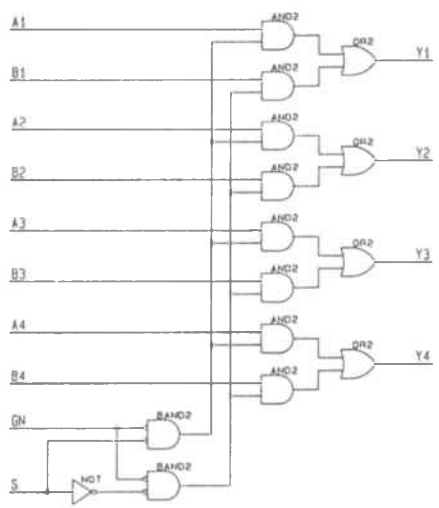
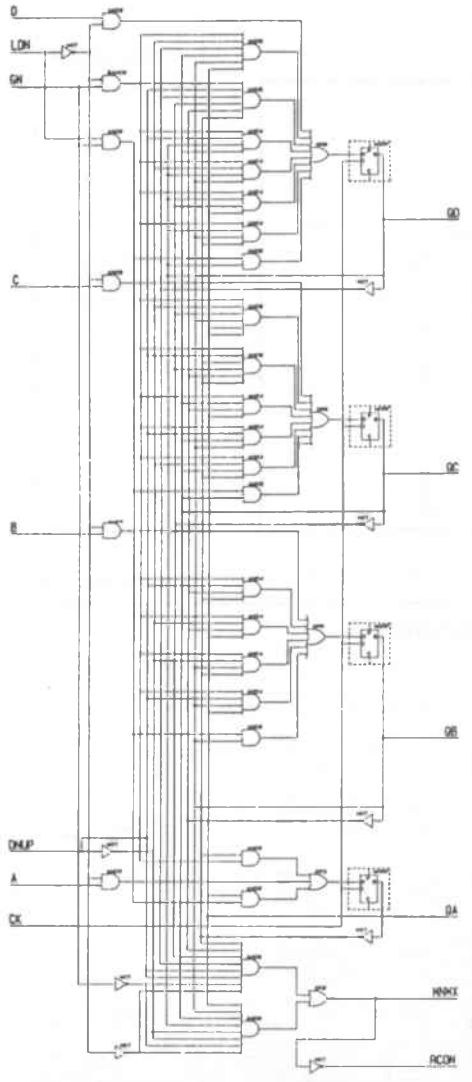


Figure 5. 74191 Schematic—*The Q outputs of the 74191 come from register feedback.*



It is not necessary to trace all the nodes given in the error message, because multiple nodes often point to the same problem. As with all error tracing, solving the first error may subsequently solve the others. There are two basic rules that assist in tracing a network:

1. The network includes all the combinatorial logic leading to the output primitive. To decide whether a MacroFunction contributes combinatorial logic, turn to the schematic in the Altera Design Library. If there is not an output primitive (Register or NOCF) before the output of the MacroFunction, then the internal logic contributes to the network. Figure 4 shows the Symbol Library schematic of the 74157. Notice that the MacroFunction could be replaced with purely combinatorial logic.
2. The network does NOT include any logic located beyond an output primitive. Output primitives represent the end of a macrocell. The feedback from that macrocell represents a single input term. To determine whether a MacroFunction contains an output primitive turn to the schematic in the Altera Design Library. Figure 5 shows the schematic of the 74191 counter. Notice that the outputs Q1 through Q4 come from the feedback of a No Output Register Feedback (NORF) output primitive.

For quick reference, the MacroFunction outputs which are purely combinatorial are listed in Table 1. These outputs contribute logic to any succeeding network.

Figure 6 shows how to trace the network driving the Y0 output from Example 1. In this case the MacroFunctions have been replaced by their logical equivalent. Compare the detailed view of Figure 6 with the higher level view previously shown in Figure 3a.

Table 1. Combinatorial Outputs of Macro-Functions—*Outputs listed contribute purely combinatorial logic to the network succeeding them.*

MarcoFunction	Outputs
7400-7452	ALL
7480	SUM, SUMn
7482	SUM1, SUM2
7483	ALL
7485	ALL
7487	ALL
74138-74158	ALL
74160-74163	RCO
74160T-74163T	RCO
74180	ALL
74181	Gn, Pn, AEQB, Cn4
74183-74185	ALL
74190	RCON, MNMX
74191	RCON, MNMX
74190T	RCON, MNMX
74191T	RCON, MNMX
74192T	CYn, BRWn
74193T	CYn, BRWn
74278	ALL
74280	ALL
74518	PO
21MUX	ALL
4COUNT	COUT
8COUNT	COUT
MULT2	ALL
MULT24	ALL
MULT4	ALL
UNICNT2	COUT

Figure 6. Logic in Macrocell 1 from Example 1—*Inputs to the macrocell are from input pins or feedback from a NOCF or NORF primitive. The terms FB1-FB4 are feedback from buried NOCF's within the 7485. (See Figure 2a and 3a).*

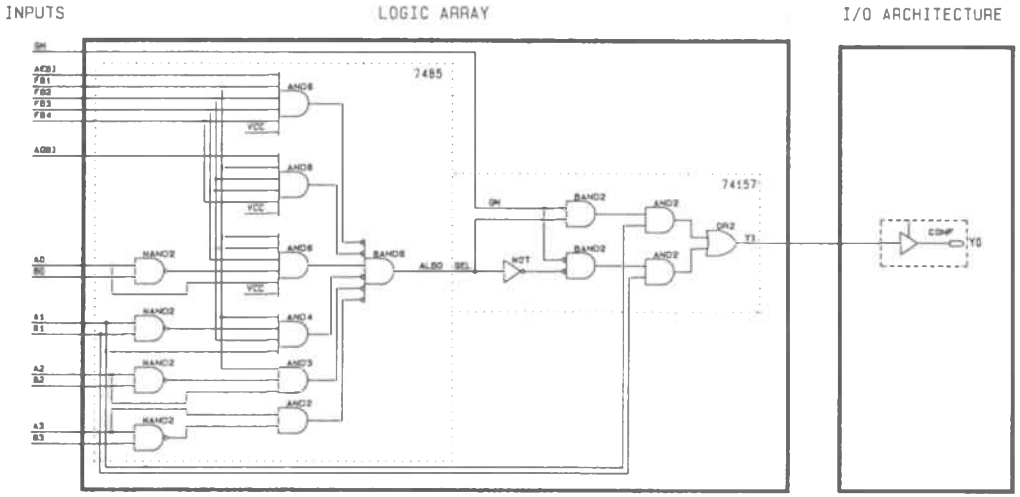
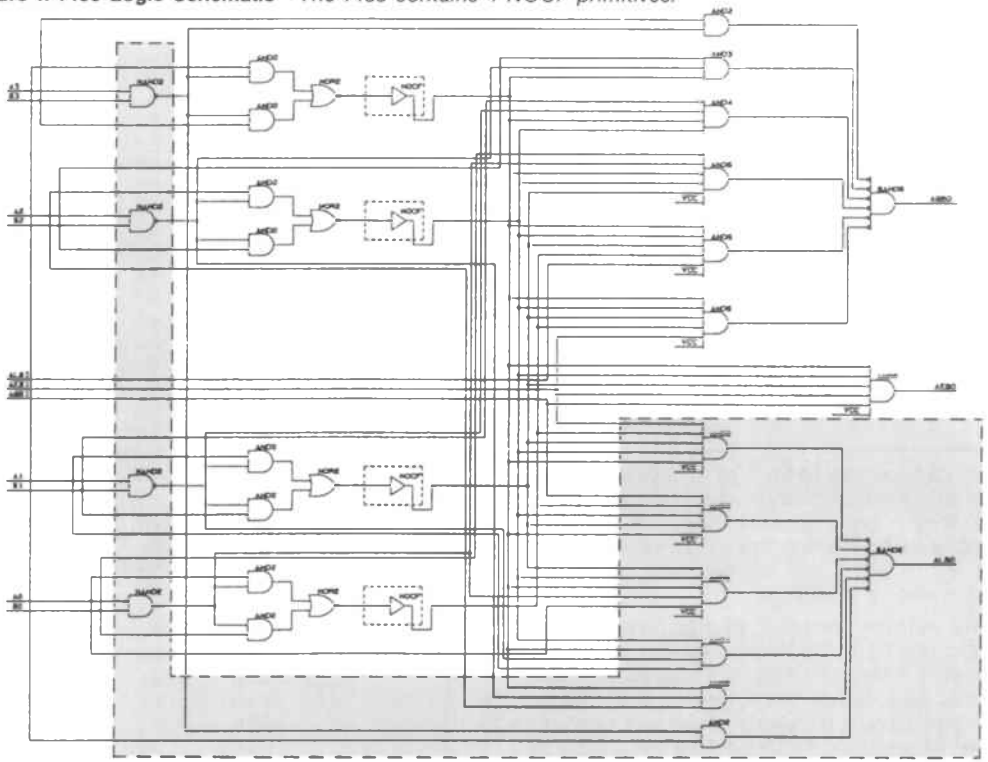


Figure 7. 7485 Logic Schematic—*The 7485 contains 4 NOCF primitives.*



Combinatorial logic from both the 7485 comparator and the 74157 multiplexer contributes to the original Y0 network. This network was found to be too complex to fit into a single macrocell.

All inputs to the macrocell come from the EPLD internal device bus. The signals A0-A3 come directly from input primitives, and B0-B3 come from the feedback or register primitives found in the 74191 counter. The signals named FB1-FB4 are feedback from 4 NOCF output primitives found within the 7485 comparator itself. Figure 7 shows these primitives in the 7485 schematic.

Note that NOT ALL of the logic in the 7485 schematic is placed in the Y0 network. The four NOCF's within the MacroFunction itself allocate 4 separate macrocells where some of the combinatorial logic is computed. Only the logic outlined in Figure 7 contributes to the network in Figures 6 and 3a. Figure 3a did not show these separate macrocells for simplicity.

NOCF PLACEMENT

Once the network is isolated, NOCF placement must be determined so that the resulting subnetworks will fit conveniently into 2 macrocells. Although it is not always obvious where to place the NOCF, there are rules which help in most cases:

1. Combinatorial outputs from MacroFunctions often need to pass through a NOCF before being used as further inputs. The logic produced by the combinatorial output from a MacroFunction is often quite complex, and may need to be isolated in a macrocell before proceeding to further logic. In Example 1 the combinatorial output from the 7485 comparator is too complex to be fed directly into the 74157 multiplexer. Placing a NOCF here solved the problem by isolating the comparator in a separate macrocell. See Table 1 for a list of combinatorial macrocells.
2. Complex inputs to MacroFunctions may need to be passed through a NOCF before proceeding to the MacroFunction. MacroFunctions contain a large amount of logic and may become quite complex if combined with additional logic from their inputs. In Example 1, the SEL input to the 74157 multiplexer comes from combinatorial logic produced by the 7485 comparator. Inserting a NOCF before the input solved the fitting problem.
3. Do NOT place NOCF's directly before or after input or output primitives. Input and output primitives are the endpoints of the macrocells, and would not reduce the logic. In Example 1, placing a NOCF after the input primitives or after the 74191 counter would not change the Y0 network. In fact, such placement would result in a wasted macrocell.
4. When more than one node is listed in an error, a NOCF often needs to be placed where the networks intersect. Points of network intersection reveal places where separation would reduce both neworks. In Example 1 the networks leading to Y0 and Y1 both contain the select input on the multiplexer. Placing a NOCF here solves both problems.

   These rules determine NOCF placement in most cases. Notice that in Example 1, each of the rules, applied independently, would lead to correct NOCF placement.

CLOCK, CLEAR AND OUTPUT ENABLE ERRORS

Related to the "Too Many Pterms" error are the errors indicating illegal Clock, Clear, or Output Enable inputs to the output primitives. These secondary inputs consume less of the logic array than the primary data inputs, and consequently have more restrictive computational ability. Specifically, each of these secondary inputs must be reduceable to one product term. A+PLUS treats these errors in the same it treats the "Too Many Pterms" error.

The error message A+PLUS issues lists the needed information in one of the following ways.

***INFO-FIT—Illegal inversion of <resource> input: <nodename>

***INFO-FIT—Too may pterms for <resource> input: <nodename>.

The product term restriction of secondary inputs leads to the following set of design rules:

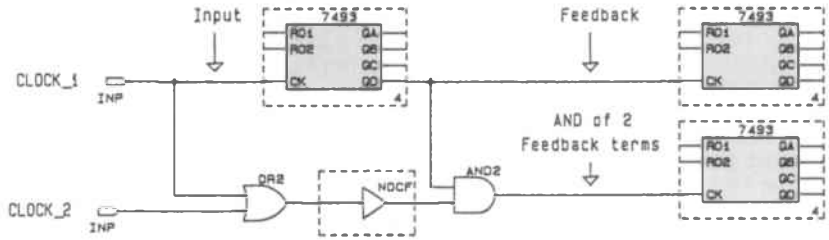
1. Clock inputs must come from an input primitive or:
   a) Feedback from any output primitive.
   b) Non-combinatorial output from a MacroFunction.
   c) A simple AND or NOR fuction of any of the above.

Figure 8 shows some legal and illegal clocking configurations. (See also Appendix A in the A+PLUS Reference Manual).

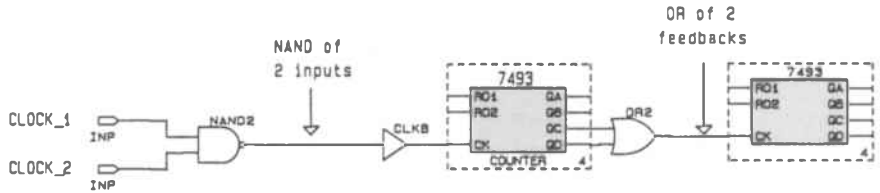
2. Inputs to asynchronous clear pins must be one of the following:
   a) An input primitive.
   b) Feedback from any output primitive.
   c) Non-combinatorial outputs from a MacroFunction.
   d) A simple AND or NOR functions of any of the above.

Figure 9 shows legal and illegal clear inputs.

Figure 8. Legal and Illegal Clock Inputs

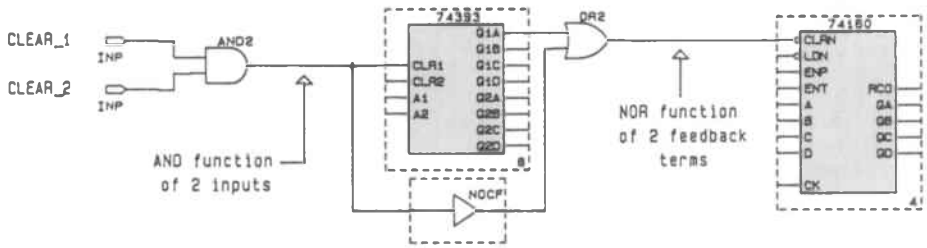


A. *Some legal clock inputs*

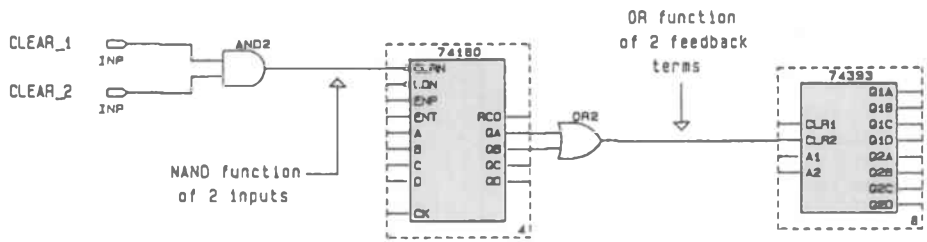


B. *Some illegal clock inputs*

Figure 9. Legal and Illegal Clear Inputs



A. *Some legal clear inputs*



B. *Some illegal clear inputs*

Figure 10a. Asynchronous Linking of Two Counters—*Asynchronous linking of two counters produces the error message shown in Figure 10b.*

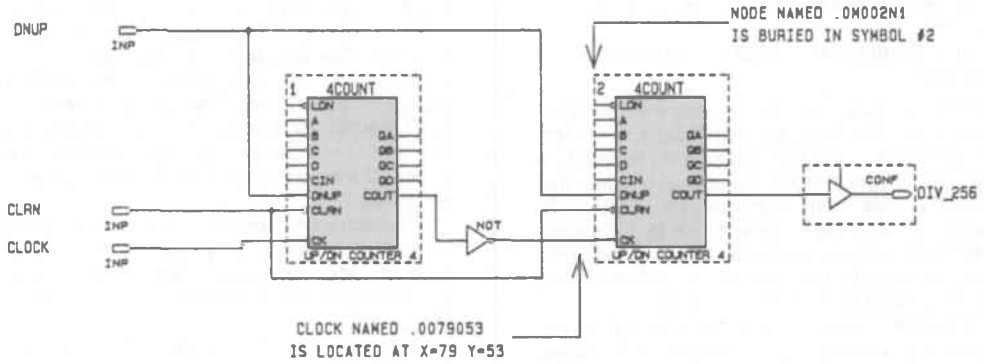
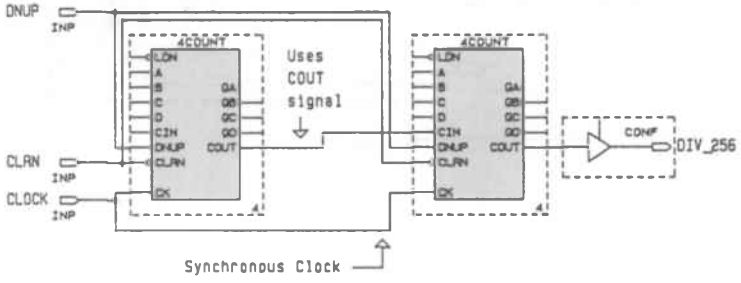


Figure 10b. Error Messages—*Error message from the asynchronous design shown in Figure 10a. The nodes and clock listed are indicated in Figure 10c.*

```
***INFO-ADP-LEF analyzed
***INFO-FIT-Illegal inversion of Clock input
        (.0079053): ".0M002N0"
***INFO-FIT-Too many (2) pterms for Clock input
        (.0079053): ".0M002N0"
***INFO-FIT-Illegal inversion of Clock input
        (.0079053): ".0M002N1"
***INFO-FIT-Too many (2) pterms for Clock input
        (.0079053): ".0M002N1"
***INFO-FIT-Illegal inversion of Clock input
        (.0079053): ".0M002N2"
***INFO-FIT-Too many (2) pterms for Clock input
        (.0079053): ".0M002N2"
***INFO-FIT-Illegal inversion of Clock input
        (.0079053): ".0M002N3"
***INFO-FIT-Too many (2) pterms for Clock input
        (.0079053): ".0M002N3"
***INFO-FIT-A+PLUS is unable to fit this design.
***INFO-FIT-No fit possible.
```

Figure 10c. Synchronous Listing of Counters—*Synchronous linking of counters allows the design to fit into an EPLD.*



3. Inputs to Output Enable pins must be one of the following:
   a) An input primitive.
   b) Feedback from any output primitive.
   c) Non-combinatorial outputs from a Macro-Function.
   d) A simple AND or NOR function of any of the above.

The first approach to solving these errors should be optimizing the logic by switching to synchronous operation. It is often possible to switch to a single synchronous clock by applying additional logic to the data input of the register. Figure 10a shows a divide by 256 counter made by cascading two 4COUNT MacroFunctions. The second stage is asynchronously clocked by inversion of the Carry Out (COUT) from the first stage.

This design produces the error message shown in Figure 10b which indicates that the clock named ".0079053" is not a legal clock signal. The COUT signal is a complex combinatorial output from the 4COUNT MacroFunction, and thus is not a legal input to the clock of the flip flop. The name ".00079053" indicates position X=79, Y=53 in the schematic.

The solution shown in Figure 10c is to switch to a synchronous clocking scheme and apply the COUT signal to the Carry In (CIN) of the second stage. As shown in the Altera Design Library, the CIN controls the clocking of the counter by gating the logic to the toggle inputs of the registers. In addition to fitting into an EPLD, the synchronous design has the added advantage that the outputs are no longer skewed as in asynchronous designs

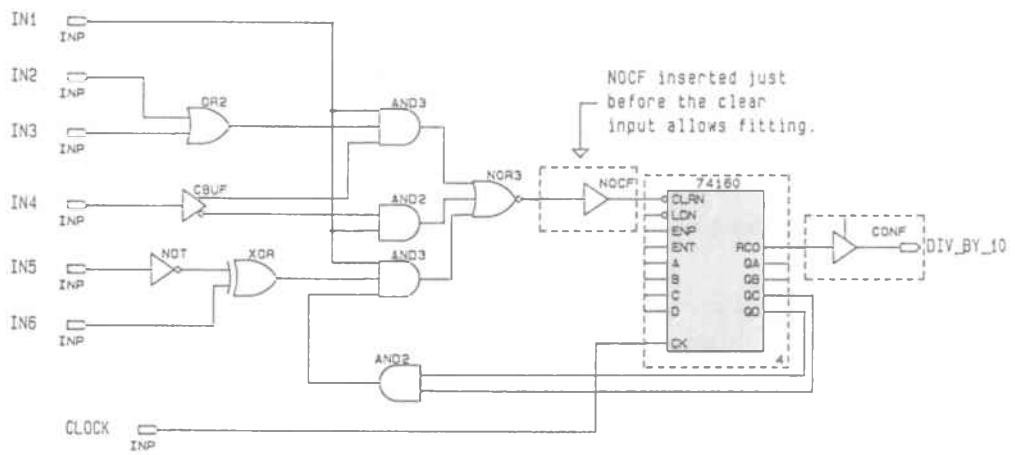
Synchronous Clear inputs are incorporated into the data inputs of the register and are not restricted by the one product term limitation of asynchronous Clears. The counters 74162, 74163, and UNICNT all have synchronous clears. In addition, the synchronous load input could be used as a clear if the parallel inputs are held low or left to the default.

If synchronous operation is not possible, the second approach is to separate the logic. In the case of secondary inputs, separation is achieved by placing a NOCF primitive directly before the secondary input.

This technique is identical to the separation of the logic network discussed previously. In this case, a NOCF is used to reduce the signal to a single term before it is fed back to the register primitive. Using this techique, a designer can create as complicated a secondary input as required. Figure 11 shows how this approach is applied to a complex clear signal to produce asynchronous clearing.

AB34 Rev 2.0

Figure 11. *A complex clear signal may be implemented by inserting a NOCF.*



FEATURES

- **Definition of Metastability.**
- **Experimental setup for Metastability measurements.**
- **Metastability characteristics of EPLDs.**

INTRODUCTION

This Application Brief describes the inherent problems associated with the synchronization of asynchronous signals. In particular, the phenomenon of Metastability in clocked flip-flop elements is explored. As a means for predicting and guaranteeing required Mean Time Between Failure (MTBF) rates in circuits employing Altera EPLDs, experimental data is presented for Altera's devices when the associated flip-flops are used in asynchronous signal synchronizer applications. For comparison, reference data on standard TTL components is also presented. Altera EPLD flip-flops have Metastability characteristics superior to LSTTL series flip-flops. A methodology is presented which allows the calculation of MTBF numbers in these situations.

THE SYNCHRONIZATION PROBLEM

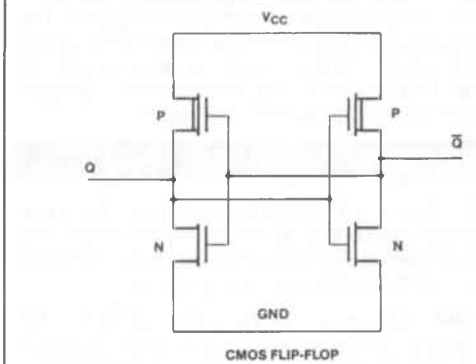
The synchronization of communication between asynchronously clocked systems is a fundamental design problem. The universe does not run on a single master clock, and as a result, matching System A's local clock to System B's is necessary.

Most systems are designed synchronously. That is, all signal transitions within and generated by a system have their transitions referenced to an edge of a master clock. In addition, synchronous systems require synchronous (or synchronized) inputs to operate reliably. Otherwise, race conditions, set-up time violations and other logic problems occur. The goal is to match or synchronize external inputs with each system's local clock to insure operational reliability and the required Mean Time Between Failure (MTBF) for the composite systems(s).

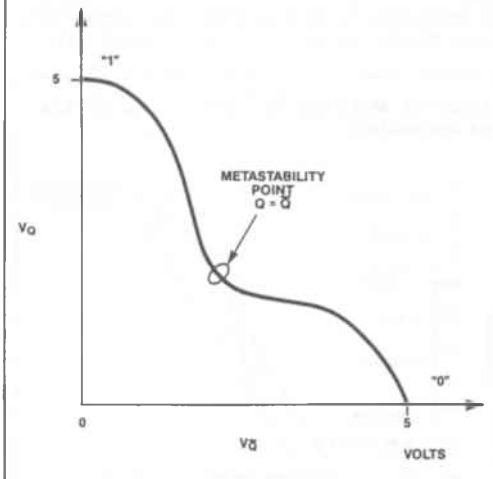
At a practical level, the edge-triggered flip-flop has long been a favorite mechanism to obtain this synchronization. Clocked by the system's master clock or a derivative, it can provide a mechanism to synchronize transitions on its Data input with the Clock and output the result to the system. All of its output transitions are synchronous with its Clock. The addition of this single synchronizer element seems to solve all the problems. Unfortunately, life is never this simple.

However, when synchronizing asynchronous signals, we can no longer guarantee the minimum timings required by the flip-flop data sheet to give deterministic behavior. For example, if a signal changes at a flip-flop's Data input from a zero to a one at precisely the instant the Clock changes what state should the output reflect? Since the minimum setup and hold times have been violated, electrical parameters and logic functions are no longer guaranteed. The combination of these logical/electrical uncertainties manifests itself in a state known as Metastability.

Figure 1. Metastability Mechanism



CMOS FLIP-FLOP



FLIP-FLOP OUTPUT
SWITCHING WAVEFORM

METASTABILITY

A flip-flop is typically defined as a bi-stable element, with the Q-Output at logic one and the Q-Output at logic zero or vice versa. The label bi-stable is misleading, since a third "stable" state is possible wherein both nodes are at identical voltages. This state is called a "Meta-stable" state as any minute disturbance will push the flip-flop in one direction or the other. Figure 1 illustrates these concepts. Once disturbed, the natural regenerative nature of the flip-flop will insure its transition to a normal stability point. During the duration of the Metastable event, however, we see the odd state where both Q and $\overline{Q}$ are at equal and intermediate logic levels.

Clocking a flip-flop with its Data input in transition, as occurs during synchronization, is exactly the situation which can lead to a Meta-stable event. The resulting indeterminate output logic levels could in fact produce unpredictable results if allowed to propagate throughout the system. The likelihood of hitting this meta-stable window will be a function of the width of the window and the signal frequencies being synchronized. Experimental data confirms this conclusion.

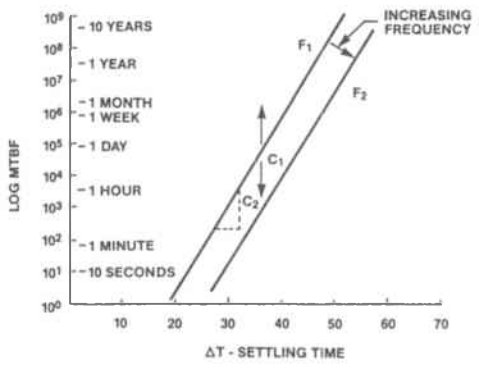
THEORETICAL MODEL

The theory of synchronizer operation has been analyzed extensively in other publications. The basic equation for the MTBF of a synchronizer due to Metastability events is given by

$$\mathbf{MTBF} = [F_{CLOCK} \times F_{DATA} \times C_1 \times e^{(-C_2 \times \Delta T)}]^{-1}$$

If this equation is plotted on semi-log graph paper, the MTBF/ΔT relationship appears as a straight line (Figure 2). In the above expression, ΔT represents the amount of settling time allowed for the flip-flop to settle to a valid, stable state.

Figure 2. Metastability Function and Effects of Parameters



The constants involved in this equation (C_1 and C_2) reflect particular characteristics of the device and more importantly, the process technology used to manufacture the device. Different devices fabricated on the same technology will have similar Metastability characteristics. Device design tricks and optimizations do not have a marked effect. Fundamental technology parameters such as on-chip capacitances and inverter gain predominate.

The constant C_1 linearly scales the MTBF equation. As such, the smaller the value of C_1, the higher the MTBF. It affects the MTBF/ΔT curve in an absolute sense, tending to translate it along the MTBF axis.

The constant C_2 affects the slope of the MTBF/ΔT plot. As such, it is a measure of how quickly, in a relative sense, the flip-flop snaps out of Metastability. The steeper the relationship on the plot, the better settling can be expected.

The form of the above equation is such that MTBF is a linear function of Clock and Data frequencies. Given data for a set of input frequencies, one may predict results for other operating conditions.

EXPERIMENTAL ARRANGEMENT

In order to experimentally evaluate the relationship between Metastability event frequency and input signal frequencies, it is necessary to have a means for defining and measuring when a Metastability event occurs. For the purposes of our experiment and this process, we will define a flip-flop to be in a Metastable state whenever its output voltage Q is greater than V_{IL} maximum and less than V_{IH} minimum for a duration longer than normal output transition times. For TTL levels, this of course corresponds to 0.8V < Output Voltage < 2.0V.

An experimental circuit to measure Metastable event has the following:

- A synchronizing device to be evaluated, the Device Under Test (DUT).
- Two independent signal sources to act as local system Clock and Data inputs.
- A means to compare the DUT's output to V_{IH} and V_{IL} levels. These may be dedicated comparator devices, or as shown in Figure 3, inverters with appropriate bias voltages applied to the device grounds. The latter arrangement has the benefit that it is a very high-speed arrangement (few nanosecond delay, dependent on logic family) and minimizes the number of power supplies required compared to most dedicated comparators.
- A means to strobe the comparators' outputs at variable delay times from the DUT Clock in order to detect the occurrence of a Metastable event of a given duration. At a practical level, as shown in the figure, this can be an inverted version of the DUT Clock. By varying the width

of the DUT Clock pulse, a variable delay bewteen the two rising edges is then obtained.

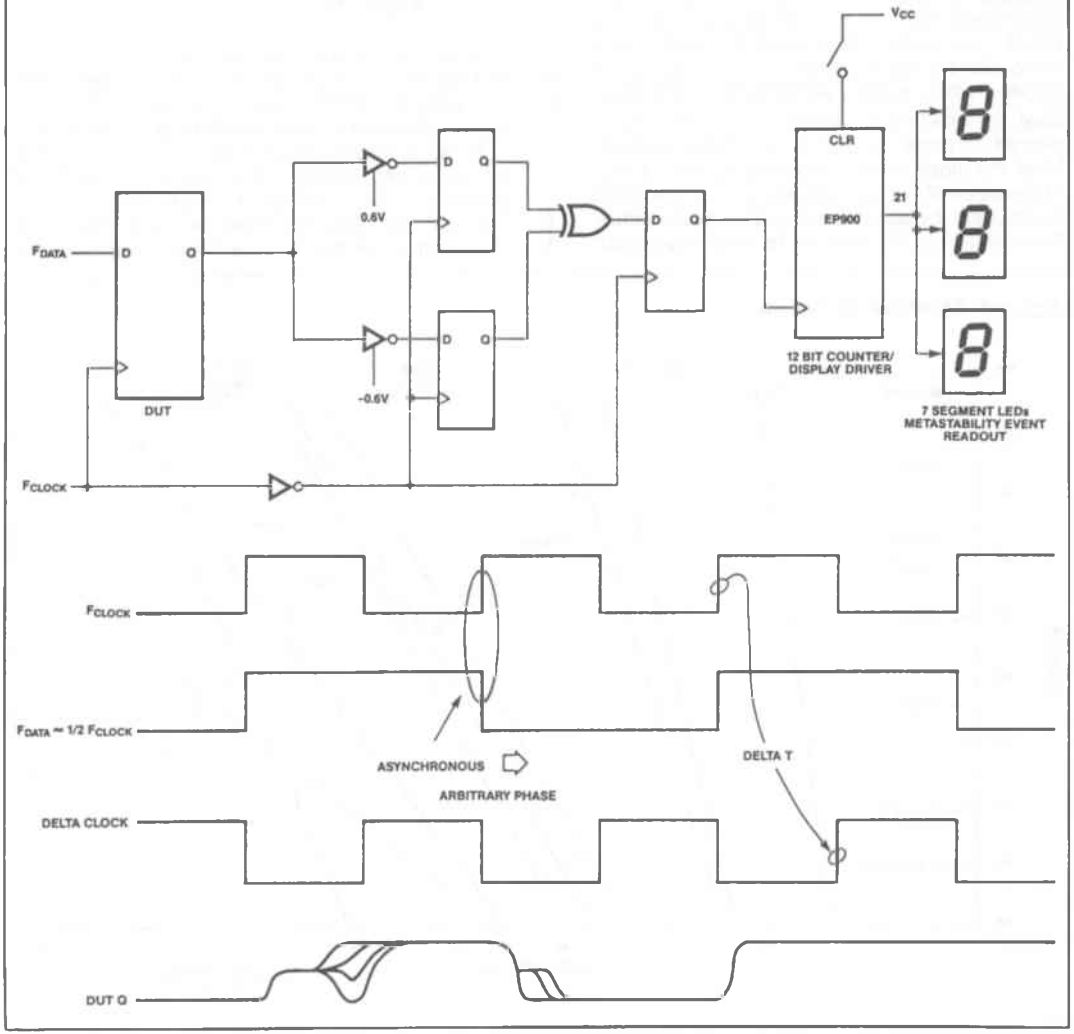
- Finally, a counter which counts the Metastabililty events. Given this arrangement, the exercise reduces to taking experimental data, where Metastability event counts as a function of ΔT and length of observation are measured. Length of the observation divided by the number of events gives an MTBF number. Plotting the data yields the by-now familiar characteristic lines.

EXPERIMENTAL RESULTS

In addition to Altera's family of EPLDs, several other logic devices of various descriptions were analyzed for Metastability characteristics. These included standard TTL 7474 devices from LS, ALS and FAST logic families, as well as 16R4A and 16R4L PAL devices from different manufacturers. The data is plotted in Figure 4. The following qualitative points are worth making:

- The Metastability characteristics of the EP310, EP600 and EP1210 Macrocell flip-flops are virtually identical. This is consistent with our previous statement that these characteristics are much more technology dependent than individual device/circuit dependent.
- The EP310, EP600 and EP1210 devices exhibit Metastability characteristics which are substantially better than the earlier EP300 and EP1200 devices. This is attributable to the fact that later

Figure 3. Experimental Setup



devices are fabricated on the more advanced CHMOS IIe process. The technology dependence of flip-flop settling time is thus reinforced.

- Altera EPLDs exhibit somewhat superior Metastability characteristics relative to LSTTL devices. When integrating such designs, EPLD flip-flops may be used as synchronizers with equivalent characteristics.
- Altera EPLDs exhibit markedly better Metastability characteristics when compared to low-power PAL devices such as the 16R4L. Given this, EPLDs may be used for equivalent performance with superior synchronizer characteristics, while frequently saving additional power.
- In those cases where ultra-fast settling time are required of a synchronizer, SSI/MSI logic families such as FAST or ALS still provide superior characteristics to high-density solutions such as EPLDs. Often, the ability to integrate multiple logic levels onto a single device, such as in an EPLD, can reduce the need for such rapid stand-alone settling times, but this option is available where isolated performance is the issue.
- Data for the input latches of the EP1210 is plotted to show their utility as synchronizers. Note the slope of the curve approximates that of the basic Macrocell flip-flop, but is translated to the right by approximately 25 nanoseconds. This translation reflects the fact that the output of these input latches must propagate through the EP1210 logic array before it is "visible" at the device pin. The Metastability characteristics of these latches are virtually identical to those of the Macrocell flip-flop, but their observability is offset by this delay.

Given the plotted data, the derivation of the two constants C_1 and C_2 for the MTBF equation is relatively easy. C_2 defines the slope of the line. Once C_2 is defined, C_1 can be determined. For the case where $F_{DATA} = 0.5 \times F_{CLOCK}$ (representing a transition on every synchronizing clock edge), the equations reduce to

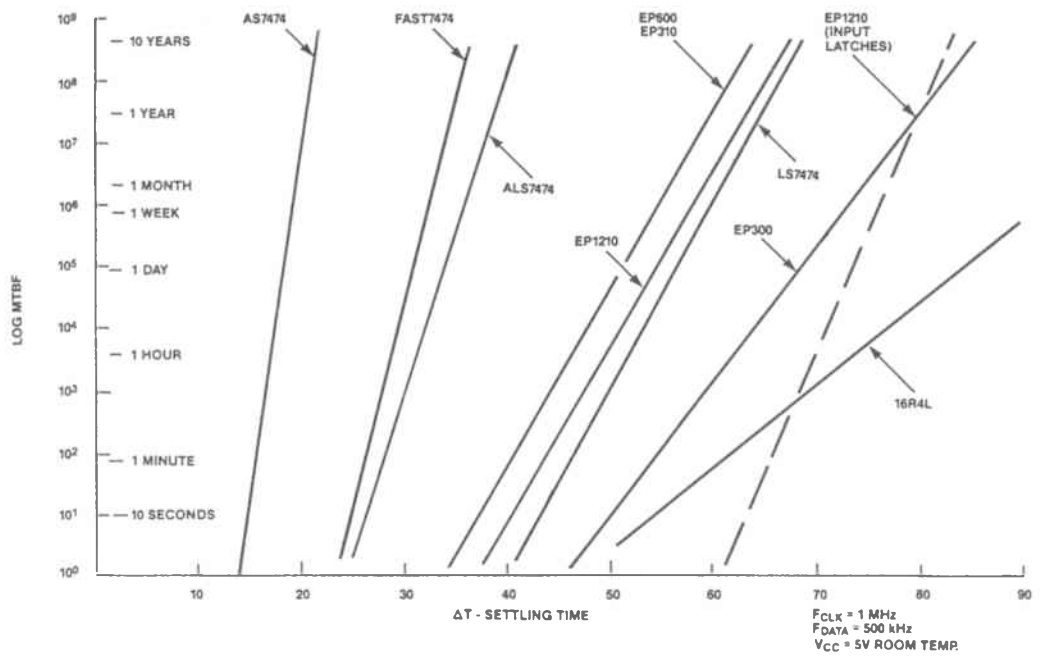
$$C_2 = \frac{\ln (MTBF_2 - MTBF_1)}{\Delta T_2 - \Delta T_1}$$

$$C_1 = \frac{2 \times e^{(C_2 \times \Delta T)}}{MTBF * f^2}$$

Table 1 summarizes the values of C_1 and C_2 for Altera's EPLDs, as well as alternative devices. From these values, MTBF calculations can be made for a particular system/clock rate/device combinations.

As an example, assume use of an EP600 in a synchronizing application and required MTBF of one year (approximately 3×10^7 seconds). In addition, assume the system clock rate is 10MHz, while the input to be synchronized has a frequency of

Figure 4. Experimental Results



2MHz. To calculate the minimum settling time allowance required to assure the specified MTBF, use the constants shown below for the EP600 and solve the metastability equation:

$$MTBF = [F_{CLOCK} \times F_{DATA} \times C_1 \times e^{(-C_2 \times \Delta T)}]^{-1}$$

or solving for settling time required

$$\Delta T = \frac{\ln [MTBF \times F_{CLOCK} \times F_{DATA} \times C_1]}{C_2}$$

substituting gives

$$\Delta T = \frac{\ln [3x10^7 \times 10x10^6 \times 2x10^6 \times .0751]}{0.658}$$

$$= \frac{\ln [4.5 \times 10^{19}]}{0.658}$$

$$= \frac{45.25}{0.658} = 68 \text{ nanoseconds}$$

In order to insure an MTBF of 1 year using an EP600, the application should allow approximately 68 nanoseconds of settling time before the output of the EP600 synchronizing macrocell is evaluated or required to be stable elsewhere in the system. Similar calculations can be made for any combination of MTBFs and Clock/Data frequencies required. It is interesting to note that due to the logarithmic relationship between MTBF and settling time, dramatic changes in MTBF can be obtained from small changes in ΔT. For example, if the MTBF requirement is increased from 1 year to 10 years in the calculations shown above, the ΔT allowance need only be increased by 4 nanoseconds (68 vs 72)! Increased design margin is relatively inexpensive, and should be examined in the context of the specific application.

Table 1.
Metastability Equation Constants vs. Device

Device	C_1	C_2
EPLD: EP300/EP1200	0.0153	0.501
EPLD: EP310/EP600/EP900/ EP1210/EP1800	0.0751	0.658
FAST 7474	4.148×10^4	1.586
LS 7474	2.357	0.697
16R4L	1.596×10^{-6}	0.300
16R4A	5.000×10^{-4}	0.965

$MTBF = [F_{CLOCK} \times F_{DATA} \times C_1 \times e^{(-C_2 \times \Delta T)}]^{-1}$

SUMMARY

In summary, Altera's current generation of EPLDs, including the EP310, EP600 and EP1210 have Metastability characteristics which make them superior to standard LSTTL flip-flops when utilized in synchronizer applications. In addition, they outperform typical low-power PAL devices substantially. Given adequate settling time, any required MTBF may be predicted as shown above and obtained for circuits employing these devices.

When designing synchronizer circuits, it is prudent to provide adequate guardbands. Synchronization is a probabilistic phenomenon, and MTBF numbers are just that: mean or average times taken over a large sample. A circuit may have an MTBF of 10 years, yet still has a finite probability of failure in its first few minutes of use. In all high reliability applications, the potential of a Metastable event can never be totally discounted.

AN9 Rev 1.0

FEATURES

- **Tested in comformance to method 1019.2 of MIL-STD-883C.**
- **Total Dose radiation tolerance in excess of 100 krad for EPROM cells.**

TOTAL DOSE GAMMA RADIATION HARDNESS OF ALTERA EPLDs

The EPROM process has been measured in a series of experiments conducted by Altera. The EP600DM was chosen as the vehicle for evaluation. Based on the failure mechanisms determined by the experiments, results on this product are expected to be typical of all Altera EPLD products.

Under the worst case conditions described below, the EP600 was found to function properly over the full military power supply range until total doses in excess of 10 krad(Si) are reached. Other conditions, which are not worst case, can result in total dose tolerance far in excess of 10 krad(Si).

The experiments were conducted using a cobalt-60 isotope, with a dose rate of 750 krad(Si) per minute, ±10%. Test units were powered up with V_{CC} = 5.0V, and all other pins grounded, both during the irradiation and subsequent to it, until measurements were made. All conditions of temperature control, dose rate, and electrical bias were in conformance to method 1019.2 of MIL-STD-883C.

Numerous key parameters were measured. The margin of programmed EPROM cells, as indicated by the maximum V_{CC} value for error-free verification and functional operation of the part, proved to be the most important. The complete list of parameters monitored is given below:

1. Programmed cell margin (from V_{CC} max)
2. Erased cell margin (from V_{CC} min)
3. AC timing (from T_{PD})
4. I_{CC} Standby vs Input voltage level
5. Output high and output low drive currents

These measurements made it possible to monitor the extent of EPROM cell programming, and also to detect changes in junction leakage and transistor threshold voltage.

As mentioned above, results show that radiation-induced charge loss of programmed EPROM cells is by far the single most important phenomenon limiting the total dose radiation hardness. Although 40 krad(Si) was the highest total dose used in these experiments, the observed variations in erased cell margin and in leakages and threshold voltages imply that the total dose radiation tolerance for these parameters is probably in excess of 100 krad(Si).

Degradation of programmed EPROM cell margins was the first cause of failure in all the experiments. It was observed that the rate of this degradation, and consequently, the total dose radiation tolerance, was a strong function of irradiation. If the top gate is biased at V_{CC} during the irradiation, the tendency for the cell to lose charge is greatly reduced. In this case, total dose radiation tolerance was observed to be in excess of 30 krad(Si) for 5.5V operation, and in excess of 40 krad(Si) for 5.0V operation. If, on the other hand, the top gate is grounded during the irradiation, the total dose radiation tolerance was reduced to about 14 krad(Si) for 5.5V operation, and to about 18 krad(Si) for 5.0V operation. This behavior suggests that energetic holes created in the bulk Si by the radiation, and attracted toward the floating gate by its negative (programmed) charge, are the dominant cause of cell charge loss.

Since it may be quite common in a system for the top gate over some of the programmed cells to be grounded during irradiation, the worst case result above of approximately 14 krad(Si) for 5.5V operation should be regarded as the best estimate of the radiation tolerance of these products. Most radiation tolerance experiments on EPROM technologies which have been reported by other workers involve top gates which are biased at V_{CC} during at least part of the irradiation; these must be interpreted cautiously since they are not conducted under worst case conditions and may consequently give falsely high radiation tolerance values.

In conclusion, some insights about programming can be offered. The results above were obtained using Altera's PLDS2 development system, in its standard configuration, as the programming unit. It was observed that the programmed cell margin fell off essentially linearly with increasing total radiation dose. It is therefore important in maximizing radiation hardness that adequate cell margin be programmed into the part prior to irradiation. Users are cautioned to employ the PLDS2 or other Altera-approved systems which provide optimized cell programming.

AB51 Rev 1.0

A+PLUS—Altera Programmable Logic User Software, is a series of software modules that transform a logic design into a programming file for Altera's general-purpose & function-specific EPLDs.

ADDRESS—The indentification by name, number, or label of a storage location, whether by a register, memory location, or other data source or destination.

ADLIB—a powerful user tool for creating customized MacroFunctions or hierarchical nested logic blocks.

ADP (Altera Design Processor)—this processor transforms the input format into optimized code used to program the target EPLD.

ALGORITHM—A prescribed sequence of well-defined rules or operations for the solution of a problem in a specified number of steps.

AMPERE—The rate of flow of electrons; a unit of electrical current. One ampere equals a flow of one coulomb per second. A current of one ampere results from one volt across one ohm of resistance.

ARCHITECTURE CONTROL BLOCK—The programmable logic block responsible for configuring architectural features of the macrocell, such as registered or combinatorial output and feedback.

ASIC—Applications Specific Integrated Circuit, normally refers to a custom, gate array, or standard cell device engineered out of the normal user design environment.

ASMILE—Altera State Machine Input Language.

ASSEMBLER—A computer program that converts source programs into object programs by translating symbolic operation codes and addresses into machine recognizable languages and absolute addresses.

ASSEMBLY LANGUAGE—A set of computer instruction symbols that includes operation instructions, address labels, and data labels which the assembler converts into machine recognizable codes.

ASYNCHRONOUS—Irregular or unpredictable, without occurring at a constant repetition rate; also not occurring at the same time or at the same time as a given repetitive signal, such as a clock.

BAUD—A measure of transmission rate derived from the amount of data that is transmitted over a specified period of time—usually one second—as opposed to the signal frequency.

BIT—The smallest unit of information in a digital system, represented as a Binary digIT (BIT). Often used to describe the capacity of a product or function.

BURIED MACROCELL—A Macrocell with no output to an external pin, but rather the output is routed back into the EPLD to use as an input for additional logic operations.

BUSTER—BUS-oriented regiSTER intensive EPLD device family designed specifically for bus coupling and bus-connected peripheral applications.

BUSTER PRIMITIVES—A special group of primitives designed for the BUSTER family of devices to simplify bus oriented functions:

RINP8—8-Bit Input Register (74377), Input from External Pins.

RBUSI—8-Bit Input Register (74377), Input from Internal Bus.

RINP8A—8-Bit Input Register (74377), Input from External Pins, No Write Strobe (WS).

RBUSIA—8-Bit Input Register (74377), Input from Internal Bus, No Write Strobe (WS).

LINP8—8-Bit Input Latch (74373), Input from External Pins.

LBUSI—8-Bit Input Latch (74373), Input from Internal Bus.

LBUSO—8-Bit Output Latch (74373), Output to Internal Bus.

BUSX—Bus Transceiver (74245), Output to Bus Port, Input to Internal Bus.

BYTE—A grouping of bits (usually eight or nine) that can represent either a character or a numerical value.

CAD—Computer Aided Design.

CERDIP—Ceramic, Dual In-Line Package.

CMOS—Complementary Metal/Oxide Semiconductor—an Metal/Oxide Semiconductor device containing both N-channel and P-channel active elements. The foundation for Altera technologies.

CPU (Central Processing Unit)—the portion of a computing system that controls the interpretation and execution of instructions and includes arithmetic capability.

CAPACITOR—A device consisting essentially of two conducting surfaces separated by an insulator (non-conducting material or dielectric such as air, paper, or mica), which stores electrical energy, blocks DC, and permits the flow of AC.

CHIP—One square on a wafer containing a single integrated circuit. The substrate on which all active and passive components of a circuit are fabricated, also called a die.

CLEAR—To restore a storage device, register or a memory to a predetermined state, also called reset.

COCF—Combinatorial Output, Combinatorial Feedback buffer primitive.

COIF—Combinatorial Output, Input (from the pin) Feedback buffer primitive.

COLF—Combinatorial Output, Latched Feedback buffer primitive.

COMPILER—A computer software program that acts as a powerful assembler which can call forth whole groups of instructions from one symbolic instruction.

CONF—Combinatorial Output, No Feedback buffer primitive.

COMPLEMENT—The opposite binary value, the opposite state, or the process of reversing binary values or states.

CORF—Combinatorial Output, Registered Feedback buffer primitive.

COUNTER—A circuit that counts pulses, often reacting or causing a reaction to a predetermined pulse or series of pulses. Also called a divider, sometimes called an accumulator.

DIP—Dual In-line Package.

DOS—Disk Operating System—usually used when referring to IBM PC or Compatible machines. Also called MS-DOS or PC-DOS.

DEVICE—A single electronic part. A specific design on the die that determines the function of that die.

DIE—A single integrated circuit separated from the wafer in which it was made (plural: dice, also sometimes called a bar). Also commonly refered to as a chip.

DUPLEX—Pertains to the operation of a communication system that accommodates simultaneous transmission and receipt of signals (full duplex) or one direction at a time (half duplex).

EPLD—Erasable Programmable Logic Device.

EPROM—Erasable Programmable Read-Only Memory.

ELECTROSTATIC—Relates to static electricity, or electricity at rest.

ESD—ElectroStatic Discharge. Relates to the transfer of static electricity, usually into a conductive component or material.

ENABLE—The activation of a circuit, device, or system, etc., through the removal of some prohibiting factor or input.

ENCODE—To assign and substitute a set of characters, digits, and symbols to represent another set; a common usage is to assign numerical codes to alphanumeric characters for computer processing.

FALL TIME—A measure of the time required for a pulse to decrease from 90% to 10% of its maximum specified level, also defined as the time required for a circuit to change its output from a high level to a low level.

FEEDBACK—In EPLD terms, used to denote the return of a signal from the output driver or register of a device back into the programmable logic array or input.

FILE—A group of data treated as a unit.

FITTER—The software module responsible for translating a design from it's original input format (equation, schematic, etc.) to a device-specific programming map usable by the device programmer.

FLATTENER—The software module responsible for converting a design from high-level Macro-Functions to low-level gate primitives.

FLIP-FLOP—A circuit with two stable states that can be changed from one to the other.

FLOATING—Pertains to the condition of a device or circuit that is neither grounded nor connected to any potential.

FREQUENCY—The number of occurrences of an iterative happening per unit of time.

FUNCTIONAL SIMULATION—A test of the ability of a proposed device to function, without regard to parametric measurements.

GATE—Circuit with two or more inputs and one output, the state of which depends upon the conditions of the inputs. Also, the control electrode of a CMOS or other field-effect transistor.

GEOMETRY (DEVICE)—The layout of the components and interconnects on the die.

GROUND STRAP—Any grounding device used to prevent static electricity from entering a device or product.

I/O—Input/Output

I/O MACROCELL—The Macrocell in an EPLD which is connected to and controls the configuration of an I/O pin on the device.

INHIBIT—To keep something from happening by the application of an appropriate signal.

INP—Input Pin Buffer primitive.

INPUT—The signal applied to a circuit or device; also, the point of signal application.

INSTRUCTION—Coded operational information generally including at least one address or operand.

INTERFACE—The point of a circuit which handles the transition between connections, signals, modes, or activities.

INTERRUPT—A break or halt in the performance of a routine, from internal or external sources, that does not prevent the computer from continuing that routine from that point at a later time.

INVERSION—The process of reversing logic functions; from positive to negative, from 1 to 0, or vice versa.

INVERTER—A circuit or device that reverses a logic signal, or compliments the logic function of the input.

JOJF—J-K flip-flop Output, J-K flip-flop Feedback buffer primitive.

JONF—J-K flip-flop Output, No Feedback buffer primitive.

JEDEC FILE—An industry standard file format defined originally for representing fuse maps in PAL devices, and often used to drive programming hardware for PLDs.

JLCC—J Lead Chip Carrier, refers to the J shape of the package leads.

LATCH—A simple logic storage element, in the simplest case using two cross-coupled gates that store a pulse applied to one logic input until a pulse is applied to the other logic input, thus storing the complementary data in the latch.

LINP—Latched Input Pin Buffer primitive.

LAB—Logic Array Block.

LATCH VOLTAGE—The effective input voltage at which a flip-flop changes states.

LEADING EDGE (SIGNAL)—The first portion or transition of a pulse or signal level change.

LOGICAPS—Altera schematic capture logic entry package

LOGICMAP II—Interface software that programs EPLDs from JEDEC files created by the Altera Design Processor.

LOW FREQUENCY—The frequency band encompassing 30 to 300 kHz.

MACROCELL—A fundamental EPLD logic block, usually consisting of three basic elements: a logic array, a configurable register, and a configurable I/O buffer.

MACROMUNCHING—Software function of Altera software which automatically eliminates unused portions of the TTL MacroFunction to reduce overall design logic requirements.

MATRIX—A logic network oriented as a rectangular array of input and output lines perpendicular to each other with some of the intersections connected to provide a specific output upon the activation of some combination of the inputs.

MAX—Multiple Array matriX, a new Altera family architecture featuring large logic arrays with small array performance.

MAX+PLUS—An integrated software environment that transforms a logic design into a programming file for Altera's MAX family of high density EPLDs.

MULTIPLEX—To perform two or more functions simultaneously or mix or transmit two or more signals simultaneously.

MULTIPLIER—A device that can supply the product of two or more inputs.

NOCF—No Output Combinatorial Feedback buffer primitive.

PARASITIC—Undesired frequency signals in a device or system.

PIA—Programmable Interconnect Array.

PLA—Programmable Logic Array.

PLCC—Plastic Leadless Chip Carrier package.

PGA—Pin Grid Array package.

PRIMITIVE SYMBOL—Logic and I/O symbols providing the basic building blocks for logic schematics.

QUIESCENT—Not active, the condition of a device or circuit when no input is applied.

RISE TIME—The amount of time required for a signal level change to increase from 10% to 90% of its final specified value.

ROCF—Registed Output, Combinatorial Feedback buffer primitive.

ROIF—Registered Output, Input (pin) Feedback buffer primitive.

ROLF—Registered Output, Latched Feedback buffer primitive.

RONF—Registered Output No Feedback buffer primitive.

RORF—Registered Output, Registered Feedback buffer primitive.

SALSA—Altera Logic Minimizer using heuristic algorithms.

SAM—Stand-Alone Microsequencer used for complex control applications.

SAMSIM—Functional simulator for SAM providing an interactive debug environment.

SAM+PLUS—SAM development software system.

SETTLING TIME—The time required after the transition of a variable for it to stabilize within established narrow limits at its desired final value.

SKEW—Delay or offset between actual and intended placement in time, position, or space.

SMF—State Machine File.

SOSF—SR latch Output, SR latch Feedback buffer primitive.

SONF—SR latch Output, No Feedback buffer primitive.

STAGE ASSIGNMENTS—Logic output based on input or input plus previous output factors.

STATE MACHINE—A logic function defined by its clock selection and state assignments coupled to its state variables and transition definitions.

STATE VARIABLE—A named variable assigned to or manipulated by state assignments.

SYNCHRONOUS—Occurring at the same time relative to a repetitious clock or pulse.

TOIF—Toggle flip-flop Output, Input (pin) Feedback buffer primitive.

TOTF—Toggle flip-flop Output, Toggle flip-flop Feedback buffer primitive.

TONF—Toggle flip-flop Output, No Feedback buffer primitive.

TTL MACROFUNCTION—A high level logic building block allowing EPLD design using TTL logic symbols.

TURBO-BIT—Programmable option on several Altera devices to control the automatic power-down feature that enables the low standby power mode of the device.

UCIC—User Configurable Integrated Circuit, a term used to indentify devices, such as EPLDs, which are reconfigurable by the end user of the device, as opposed to factory configurability notable in ASIC devices.

16CUDSLR ("16cudslur")—16 bit up/down counter TTL MacroFunction.

.ADF—Altera Design File, used as logic processor input format for A+PLUS.

.CMD—Command File for functional simulator input in A+PLUS.

.JED—JEDEC Standard Programming File in A+PLUS.

.LEF—Logic Equation File in A+PLUS.

.LOG—Command Log Output File from A+PLUS functional simulator.

.RPT—Utilization Report File from the A+PLUS fitter.

.SDF—Secondary Design File from A+PLUS flattener/macromuncher to the design file translator.

.TBL—Vector Table Output File from the A+PLUS functional simulator.

.WAV—Waveform Output File from the A+PLUS functional simulator.

REPRESENTATAIVES AND DISTRIBUTORS **PAGE NO.**

ALABAMA
Montgomery Marketing, Inc.
4922 Cotton Row
Huntsville, AL 35805
(205) 830-0498

ARIZONA
Tusar
6016 E. Larkspur
Scottsdale, AZ 85254
(602) 998-3688

ARKANSAS
Technical Marketing, Inc.
3320 Wiley Post Road
Carrollton, TX 75006
(214) 387-3601

CALIFORNIA
Addem
1015 Chestnut Avenue
Suite 330
Carlsbad, CA 92008
(619) 729-9216

Exis Incorporated
2860 Zanker Road
Suite 108
San Jose, CA 95134
(408) 433-3947

Hi-Tech Rep Company
1111 El Camino Real
Suite 108
Tustin, CA 92680
(714) 730-9561

Hi-Tech Rep Company
31332 Via Colinas
Suite 109
Westlake Village, CA 91362
(818) 706-2916

COLORADO
Promotional Technology
7490 Club House Road
Suite 204
Boulder, CO 80301
(303) 530-4774

DISTRICT OF COLUMBIA
Robert Electronic Sales
5525 Twin Knolls Road
Suite 331
Columbia, MD 21045
(301) 982-1177

CONNECTICUT
Sales Inc.
237 Hall Avenue
Wallingford, CT 06492
(203) 269-8853

DELAWARE
BGR Associates
Evesham Commons
525 Route 73
Suite 100
Marlton, NJ 08053
(609) 983-1020

FLORIDA
EIR, Inc.
1057 Maitland Center Commons
Maitland, FL 32751
(305) 660-9600

GEORGIA
Montgomery Marketing, Inc.
3000 Northwoods Parkway
Suite 245
Norcross, GA 30071
(404) 447-6124

IDAHO
Promotional Technology
7490 Club House Road
Suite 204
Boulder, CO 80301
(303) 530-4774

Westerberg & Associates Inc.
12505 NE Bel-Red Road
Suite 112
Bellevue, WA 98005
(206) 453-8881

ILLINOIS
Oasis Sales Corporation
1101 Tonne Road
Elk Grove Village, IL 60007
(312) 640-1850

Midwest Technical Sales Inc.
136 Cedar Crest Ct.
St. Charles, MO 63301
(314) 441-1012

INDIANA
Electro Reps Inc.
7240 Shadeland Station
Suite 275
Indianapolis, IN 46250
(317) 842-7202

IOWA
Midwest Technical Sales
2510 White Eagle Trail, SE
Cedar Rapids, IA 52403
(319) 365-4011

KANSAS
Midwest Technical Sales
15301 W. 87th Street
Suite 200
Lenexa, KS 66219
(913) 888-5100

Midwest Technical Sales Inc.
21901 LaVista
Wichita, KS 67052
(316) 794-8565

KENTUCKY
Electro Reps Inc.
7240 Shadeland Station
Suite 275
Indianapolis, IN 46250
(317) 842-7202

LOUISIANA
Technical Marketing Inc.
2901 Wilcrest Drive
Suite 139
Houston, TX 77042
(713) 783-4497

MAINE
Technology Sales Inc.
332 Second Avenue
Waltham, MA 02154
(617) 890-5700

MARYLAND
Robert Electronic Sales
5525 Twin Knolls Road
Suite 331
Columbia, MD 21045
(301) 995-1900

MASSACHUSETTS
Technology Sales Inc.
332 Second Avenue
Waltham, MA 02154
(617) 890-5700

MICHIGAN
Rathsburg Associates Inc.
17600 Northland Park
Suite 100
Southfield, MI 48086
(313) 559-9700

MINNESOTA
Cahill, Schmitz & Cahill, Inc.
315 North Pierce
St. Paul, MN 55104
(612) 646-7217

MISSISSIPPI
Montgomery Marketing, Inc.
3000 Northwoods Parkway
Suite 245
Norcross, GA 30071
(404) 447-6124

MISSOURI
Midwest Technical Sales Inc.
136 Cedar Crest Ct.
St. Charles, MO 63301
(314) 441-1012

MONTANA
Promotional Technology
7490 Club House Road
Suite 204
Boulder, CO 80301
(303) 530-4774

NEBRASKA
Midwest Technical Sales Inc.
15301 W. 87th Street
Suite 200
Lenexa, KS 66219
(913) 888-5100

Midwest Technical Sales Inc.
2510 White Eagle Trail, SE
Cedar Rapids, IA 52403
(319) 365-4011

NEVADA
Exis Incorporated
2860 Zanker Road
Suite 108
San Jose, CA 95134
(408) 433-3947

Tusar
6016 E. Larkspur
Scottsdale, AZ 85254
(602) 998-3688

NEW HAMPSHIRE
Technology Sales Inc.
332 Second Avenue
Waltham, MA 02154
(617) 890-5700

NEW JERSEY
BGR Associates
Evesham Commons
525 Route 73
Suite 100
Marlton, NJ 08053
(609) 983-1020

ERA, Inc.
354 Veterans Memorial Hwy.
Commack, NY 11725
(516) 543-0510

NEW MEXICO
Nelco Electronix
4801 General Bradley, N.E.
Albuquerque, NM 87111
(505) 292-3657

NEW YORK (Metro)
ERA, Inc.
354 Veterans Memorial Hwy.
Commack NY 11725
(516) 543-0510

NEW YORK STATE
Technology Sales Inc.
470 Perinton Hills Office Park
Fairport, NY 14450
(716) 223-7500

ALTERA SALES REPRESENTATIVES

NORTH CAROLINA
Montgomery Marketing, Inc.
1200 Trinity Road
Raleigh, NC 27607
(919) 851-0010

NORTH DAKOTA
Cahill, Schmitz & Cahill, Inc.
315 North Pierce
St. Paul, MN 55104
(612) 646-7217

OHIO
The Lyons Corporation
4812 Frederick Road
Suite 101
Dayton, OH 45414
(513) 278-0714

The Lyons Corporation
4615 W. Streetsboro Road
Richfield, OH 44286
(216) 659-9224

OKLAHOMA
Technical Marketing, Inc.
3320 Wiley Post Road
Carrollton, TX 75006
(214) 387-3601

OREGON
Westerberg & Associates Inc.
7165 SW Fir Loop
Portland, OR 97223
(503) 620-1931

PENNSYLVANIA
The Lyons Corporation
4812 Frederick Road
Suite 101
Dayton, OH 45414
(513) 278-0714

BGR Associates
Evesham Commons
525 Route 73
Suite 100
Marlton, NJ 08053
(609) 983-1020

PUERTO RICO
Technology Sales Inc.
Box 121
San German, PR 00753-1021
(809) 892-4745

RHODE ISLAND
Technology Sales Inc.
60 Church Street
Suite 18
Yalesville, CT 06492
(203) 269-8853

SOUTH CAROLINA
Montgomery Marketing, Inc.
1200 Trinity Road
Raleigh, NC 27607
(919) 851-0010

SOUTH DAKOTA
Cahill, Schmitz & Cahill, Inc.
315 North Pierce
St. Paul, MN 55104
(612) 646-7217

TENNESSEE
Montgomery Marketing, Inc.
4922 Cotton Row
Huntsville, AL 35805
(205) 830-0498

TEXAS
Technical Marketing, Inc.
3320 Wiley Post Road
Carrollton, TX 75006
(214) 387-3601

Technical Marketing, Inc.
2901 Wilcrest Drive
Suite 139
Houston, TX 77042
(713) 783-4497

Technical Marketing, Inc.
1315 Sam Bass Circle
Suite B-3
Round Rock, TX 78681
(512) 244-2291

UTAH
Promotional Technology
7490 Club House Road
Suite 204
Boulder, CO 80301
(303) 530-4774

VERMONT
Technology Sales, Inc.
632 Second Avenue
Waltham, MA 02154
(617) 890-5700

VIRGINIA
Robert Electronic Sales
7637 Hull Street Road
Suite 103
Richmond, VA 23235
(804) 276-3979

WASHINGTON
Westerberg & Associates Inc.
12505 NE Bel-Red Road
Suite 112
Bellevue, WA 98005
(206) 453-8881

WEST VIRGINIA
The Lyons Corporation
4812 Frederick Road
Suite 101
Dayton, OH 45414
(513) 278-0714

WISCONSIN
Oasis Sales Corporation
1305 N. Barker Road
Brookfield, WI 53005
(414) 782-6660

Cahill, Schmitz & Cahill, Inc.
315 North Pierce
St. Paul, MN 55104
(612) 646-7217

WYOMING
Promotional Technology
7490 Club House Road
Suite 204
Boulder, CO 80301
(303) 530-4774

CANADA
Kaytronics
106-10334-152 A Street
Surrey
BC, Canada V3R 7P8
(604) 581-5005

Kaytronics
4019 Carling Avenue
Suite #204
Kanata
Ontario, Canada K2K 2A3
(613) 592-6606

Kaytronics
Unit No. 1, 331 Bowes Road
Concord
Ontario, Canada L4K 1B1
(416) 669-2262

Kaytronics
5800 Thimens Blvd.
Ville St. Pierre
Quebec, Canada H4S 1S5
(514) 745-5800

ALTERA SALES OFFICES

ALTERA CORPORATION
3525 Monroe Street,
Santa Clara, CA 95051
(408) 984-2800 Telex 888496
FAX (408) 248-6924

ALTERA NORTHEAST OFFICE
945 Concord Street,
Framingham, MA 01701
(617) 626-0181 Telex 948477
FAX (617) 879-0698

ALTERA MIDWEST OFFICE
200 W. Higgins Road, Suite 216,
Schaumburg, IL 60195
(312) 310-8522 TWX 510 101 1409
FAX (312) 310-0909

ALTERA SOUTHEAST OFFICE
1080 Holcomb Bridge Road
Suite 100
Roswell, GA 30076
(404) 594-7621 Telex 382207
FAX (404) 998 9830

ALTERA SOUTHWEST OFFICE
17100 Gillette Avenue
Irvine, CA 92714
(714) 474-9616
FAX (714) 261 8697

ALTERA EUROPE
Avenue de la Tanche 2
B-1160 Bruxelles
BELGIUM
(02) 674-5223 Telex 25387
FAX (02) 674-5207

ALTERA UK
42 Queen Street
Maidenhead
Berkshire, England SL61JE
(44) 628 32516
Telex: 851 940 16389
FAX (44) 628 770892

ALABAMA
Pioneer Standard
4825 University
Huntsville, AL 35816
(205) 837-9300

Schweber Electronics
4910 Corporate Drive
Huntsville, AL 35805
(205) 895-0480

ARIZONA
Schweber Electronics
11049 N 23rd Drive
Suite 100
Phoenix, AZ 85029
(602) 997-4874

Wyle Laboratories
17855 N. Black Canyon Hwy.
Phoenix, AZ 85023
(602) 866-2888

CALIFORNIA
Anthem
1040 East Brokaw
San Jose, CA 95131
(408) 282-1587

Schweber Electronics
1225 W. 190th
Suite 360
Gardena, CA 90248
(213) 327-8409

Schweber Electronics
21139 Victory Blvd.
Canoga Park, CA 91303
(818) 999-4702

Schweber Electronics
90 East Tasman Drive
San Jose, CA 95134
(408) 432-7171

Schweber Electronics
17822 Gillette Avenue
Irvine, CA 92714
(714) 863-0200

Schweber Electronics
6750 Nancy Ridge Drive
Suite D & E, Building 7
Carroll Ridge Business Park
San Diego, CA 92121
(619) 450-0454

Schweber Electronics
1771 Tribute Road
Suite B
Sacramento, CA 95815
(916) 929-9732

Schweber Electronics
371 Van Ness Way
Suite 100
Torrance, CA 90501
(213) 320-8090

Wyle Laboratories
11151 Sun Center Drive
Rancho Cordova, CA 95670
(916) 638-5282

Wyle Laboratories
3000 Bowers Avenue
Santa Clara, CA 95051
(408) 727-2500

Wyle Laboratories
90 East Tasman Drive
San Jose, CA 95134
(408) 946-7171

Wyle Laboratories
17872 Cowan Avenue
Irvine, CA 92714
(714) 863-9953

Wyle Laboratories
124 Maryland Street
El Segundo, CA 90245
(213) 322-8100

Wyle Laboratories
9525 Chesapeake Drive
San Diego, CA 92123
(619) 565-9171

Wyle Laboratories
26677 W. Agoura Road
Calabasis, CA 91302
(818) 880-9001

COLORADO
Schweber Electronics
Highland Tech Business Park
Suite 200
8955 East Nichols Avenue
Englewood, CO 80112
(303) 799-0258

Wyle Laboratories
451 E 124th Street
Thorton, CO 80241
(303) 457-0053

CONNECTICUT
Lionex Corporation
170 Research Parkway
Meriden, CT 06450
(203) 237-2282

Pioneer Standard
112 Main Street
Norwalk, CT 06851
(203) 853-1515

Schweber Electronics
Commerce Park
Finance Drive
Danbury, CT 06810
(203) 748-7080

FLORIDA
Pioneer Standard
221 North Lake Blvd.
Altamonte Springs, FL 32701
(305) 834-9090

Pioneer Standard
674 South Military Trail
Deerfield Beach, FL 33441
(305) 428-8877

Schweber Electronics
3665 Park Center Blvd. N.
Pompanood, FL 33064
(305) 977-7511

Schweber Electronics
215 N Lake Blvd.
Altmonte Springs, FL 32701
(305) 331-7555

GEORGIA
Pioneer Standard
3100F Northwoods Place
Norcross, GA 30071
(404) 448-1711

Schweber Electronics
2979 Pacific Drive
Suite E
Norcross, GA 30092
(404) 449-4732

IOWA
Schweber Electronics
5270 N Park Place N E
Cedar Rapids, IA 52402
(319) 373-1417

ILLINOIS
Hall-Mark Electronics
210 Mittel Drive
Woodale, IL 60191
(312) 860-3800

Pioneer Standard
1551 Carmen Drive
Elk Grove Village, IL 60007
(312) 437-9680

Schweber Electronics
904 Cambridge Drive
Elk Grove Village, IL 60007
(312) 364-3750

INDIANA
Hall-Mark Electronics
4275 W. 96th Street
Indianapolis, IN 46268
(317) 872-8875

Pioneer Standard
6408 Castleplace Drive
Indianapolis, IN 46250
(317) 849-7300

KANSAS
Pioneer Standard
10551 Lackman Road
Lenexa, KS 66215
(913) 492-0500

Schweber Electronics
10300 W 103rd Street
Suite 200
Overland Park, KS 66214
(913) 492-2922

MASSACHUSETTS
Lionex Corporation
36 Jonspin Road
Wilmington, MA 01887
(617) 657-5170

Pioneer Standard
44 Hartwell Avenue
Lexington, MA 02173
(617) 861-9200

Schweber Electronics
25 Wiggins Avenue
Bedford, MA 01730
(617) 275-5100

Schweber Electronics
265 Ballardvale Street
Wilmington, MA 01887
(617) 657-8760

MARYLAND
Pioneer Standard
9100 Gaither Road
Gaithersburg, MD 20877
(301) 921-0660

Schweber Electronics
9330 Gaither Road
Gaithersburg, MD 20877
(301) 840-5900

MICHIGAN
Pioneer Standard
13485 Stamford
Livonia, MI 48150
(313) 525-1800

Schweber Electronics
12060 Hubbard Drive
Livonia, MI 48150
(313) 525-8100

MINNESOTA
Hall-Mark Electronics
10300 Valley View Road
Suite 101
Eden Prairie, MN 55344
(612) 941-2600

Pioneer Standard
7625 Golden Triangle Drive
Suite G
Eden Prairie, MN 55344
(612) 944-3355

Schweber Electronics
7424 W 78th Street
Edina, MI 55435
(612) 941-5280

MISSOURI
Schweber Electronics
502 Earth City Expressway
Earth City, MO 63045
(314) 739-0526

NORTH CAROLINA
Pioneer Standard
9801 A Southern Pine Blvd.
Charlotte, NC 28210
(704) 527-8188

Pioneer Standard
2810 Meridian Pkwy.
Suite 148
Durham, NC 27713
(919) 544-5400

Schweber Electronics
1 North Commerce Center
5285 North Blvd.
Raleigh, NC 27604
(919) 876-0000

NEW HAMPSHIRE
Schweber Electronics
Bellford Farms, Bldg. 2
Kilton & So. River Rd.
Manchester, NH 03102
(603) 625-2250

NEW JERSEY
Lionex Corporation
311 Route 46 West
Fairfield, NJ 07006
(201) 227-7960

Pioneer Standard
45 Route 46
Pine Brook, NJ 07058
(201) 575-3510

Schweber Electronics
18 Madison Road
Fairfield, NJ 07006
(201) 227-7880

NEW MEXICO
Alliance Electronics
11030 Cochiti SE
Albuquerque, NM 87123
(505) 292-3360

NEW YORK
Lionex Corporation
400 Oser Avenue
Hauppauge, NY 11787
(516) 273-1660

Pioneer Standard
60 Crossways Park West
Woodbury, NY 11797
(516) 921-8700

Pioneer Standard
68 Corporate Drive
Bingham, NY 13904
(607) 722-9300

Pioneer Standard
840 Fairport Park
Fairport, NY 14450
(716) 381-7070

Schweber Electronics
3 Town Line Circle
Rochester, NY 14623
(716) 424-2222

Schweber Electronics
Jericho Turnpike
Westbury, NY 11590
(516) 334-7474

OHIO
Hall-Mark Electronics
5821 Harper Road
Solon, OH 44139
(216) 349-4631

Hall-Mark Electronics
400 E. Wilson Ridge Rd.
Suite S
Worthington, OH 43085
(614) 888-3313

Pioneer Standard
4800 East 131st Street
Cleveland, OH 44105
(216) 587-3600

Pioneer Standard
4433 Interpoint Blvd.
Dayton, OH 45424
(513) 236-9900

Schweber Electronics
7865 Paragon Road
Suite 210
Dayton, OH 45459
(513) 439-1800

Schweber Electronics
23880 Commerce Park Road
Beachwood, OH 44122
(216) 464-2970

OKLAHOMA
Hall-Mark Electronics
5460 S. 103 East Avenue
Tulsa, OK 75145

Quality Components
3158 South 108th East Avenue
Suite 274
Tulsa, OK 74146
(918) 664-8812

Schweber Electronics
4815 South Sheridan
Fountain Plaza
Suite 109
Tulsa, OK 74145
(918) 622-8000

OREGON
Wyle Laboratories
5289 NE Elam Young Pkwy.
Bldg. E100
Hillsboro, OR 97123
(503) 640-6000

PENNSYLVANIA
Lionex Corporation
101 Rock Road
Horsham, PA 19044
(215) 443-5150

Pioneer Standard
261 Gibralter Road
Horsham, PA 19044
(215) 674-4000

Pioneer Standard
259 Kappa Drive
Pittsburg, PA 15238
(412) 782-2300

Schweber Electronics
1000 R I D C Plaza
Suite 203
Pittsburg, PA 15238
(412) 782-1600

Schweber Electronics
231 Gibralter Road
Horsham, PA 19044
(215) 441-0600

TEXAS
Hall-Mark Electronics
11333 Pagemill Road
Dallas, TX 75243-8399
(214) 343-5903

Hall-Mark Electronics
12211 Technology Blvd.
Austin, TX 78727
(512) 258-8848

Hall-Mark Electronics
8000 West Glen
Houston, TX 77063
(713) 781-6100

Pioneer Standard
5853 Point West Drive
Houston, TX 77036
(713) 988-5555

Pioneer Standard
1826 Kramer Lane #D
Austin, TX 78758-4239
(512) 835-4000

Pioneer Standard
13710 Omega Road
Dallas, TX 75244
(214) 386-7300

Quality Components
2120 West Braker Lane
Suite M
Austin, TX 78758
(512) 835-0220

Quality Components
1005 Industrial Blvd.
Sugar Land, TX 77487
(713) 240-2255

Quality Components
4257 Kellway Circle
Addison, TX 75001
(214) 733-4300

Schweber Electronics
4202 Beltway Drive
Dallas, TX 75234
(214) 661-5010

Schweber Electronics
10625 Richmond Avenue
Suite 100
Houston, TX 77042
(713) 784-3600

Schweber Electronics
6300 La Calma Drive
Suite 240
Austin, TX 78752
(512) 458-8253

Wyle Laboratories
2120 West Braker Lane
Suite F
Austin, TX 78758
(512) 834-9957

Wyle Laboratories
1810 Greenville Avenue
Richardson, TX 75081
(214) 235-9953

Wyle Laboratories
11001 South Wilcrest
Suite 100
Houston, TX 77099
(713) 879-9953

UTAH
Wyle Laboratories
1325 W. 2200 So.
Suite E
West Valley City, UT 84119
(801) 974-9953

WASHINGTON
Wyle Laboratories
1750 132nd Avenue NE
Bellevue, WA 98005
(206) 453-8300

WISCONSIN
Hall-Mark Electronics
16255 West Lincoln Avenue
New Berlin, WI 53151
(414) 797-7844

Schweber Electronics
3050 S. Calhoun Road
New Berlin, WI 53151-3549
(414) 784-9020

QUEBEC
Future Electronics
237 Hymus Blvd.
Pointe-Claire, Quebec
H9R 5C7
(514) 694-7710

ONTARIO
Future Electronics
Baxter Center
1050 Baster Rd.
Ottawa, Ontario
K2C 3P2
(613) 820-8313

Future Electronics
82 St. Regis Crescent North
Downsview, Ontario
M3S 123
(416) 638-4771

SEMAD Electronics
85 Spy Court
Markham, Ontario
L3R 4Z4
(416) 475-3922

SEMAD Electronics
8563 Government Street
Burnaby, BC
V3N 4S9
(604) 420-9889

SEMAD Electronics
243 Place Frontenac
Point Claire, Quebec
H9R 4Z7
(514) 694-0860

SEMAD Electronics
1827 Woodward Drive
Suite 303
Ottawa, Ontario
K2C 0R3
(613) 727-8325

SEMAD Electronics
75 Glendeer Drive SE
Suite 210
Calgary, Alberta
T2H 2S8
(403) 252-5664

ALBERTA
Future Electronics
3220 5th Ave.
North East Calgary, Alberta
T2A 5N1
(403) 235-5325

Future Electronics
5312 Calgary Trail
Edmonton, Alberta
T6H 458
(403) 438-2858

BRITISH COLUMBIA
Future Electronics
1695 Boundary Road
Vancouver, B.C.
V5K 4X7
(604) 294-1166

ARGENTINA
YEL S.R.L.
Cangallo 1454
Piso 8-Of. 41
1037 Buenos Aires
Argentina
Telephone: 01-462211
Telex: 18605 YEL AR
Telefax: 01-45-2551

AUSTRALIA
Hardie Technologies
(NSD Australia)
205 Middleborough Road
Box Hill, Victoria, 3128
Australia
Telephone: (03) 890 0970
Telex: 37857
Telefax: (03) 899 0819

AUSTRIA
Hitronik
St. Veit-Gasse 51
A-1130 Wien
Austria
Telephone: (0222) 824 199
Telex: 134404
Telefax: (0222) 826 440

BELGIUM
D & D Electronics
Vlle Olympiadelaan 93
2020 Antwerpen
Telephone: (03) 827 7934
Telex: 73121
Telefax: (03) 828-7254

BRAZIL
Intectra
2629 Terminal Blvd.
Mountain View, CA 94043 USA
Telephone: (415) 967-8818
Telex: 345545

DENMARK
E.V. Johanssen Elektronik A/S
Titangade 15
DK-2200 Kobenhavn N
Denmark
Telephone: (01) 83 90 22
Telex: 16522 evicas dk
Telefax: (01) 83 92 22

FINLAND
Yleiselektroniikka Oy
P.O. Box 73
Luomannotko 6
SF-02201 Espoo
Finland
Telephone: (358) 0-452-12-55
Telex: (857) 123212 Yleoy sf
Telefax: (358) 0-428-932

FRANCE
Tekelec-Airtronic
Headquarters
Cite Des Bruyeres
Rue Carle Vernet
92310 Sevres
Telephone: 16 (1) 45 34 75 35
Telex: 204 552
Telefax: 16 (1) 45 07 21 91

Tekelec-Airtronic
Paris Sud
Tour Evry 2
523 Place des Terrasses
91034 Evry Cedex
Telephone: 60 77 82 66
Telex: 691 158 F

Tekelec-Airtronic
Paris 92
BP NR 2
1 Rue Carle Vernet
92310 Sevres
Telephone: 45 34 75 35
Telex: 204 552 F

Tekelec-Airtronic
Paris 78
5 Allee du Bourbonnais
78310 Maurepas
Telephone: 30 62 00 58
Telex: 698 121 F

Tekelec-Airtronic
Paris Est
424, La Closerie Bat A
Clos Mont d'Est
93160 Noisy Le Grand
Telephone: 43 04 62 00
Telex: 220 368 F

Tekelec-Airtronic
Paris Nord
8 Avenue Salvador Allende
93804 Epinary Cedex
Telephone: 48 21 60 44
Telex: 630 260 F

Tekelec-Airtronic
Lyon
26 Rue de la Baisse
69100 Villeurbanne
Telephone: 78 84 00 08
Telex: 370 481 F

Tekelec-Airtronic
Grenoble
15 Avenue Granier
BP91
38240 Meylan
Telephone: 76 41 11 36
Telex: 980 207 F

Tekelec-Airtronic
Aix en Provence
Batiment "Le Mercure"
Avenue Ampere BP 77
13762 Les Milles Cedex
Telephone: 42 24 40 45
Telex: 440 928 F

Tekelec-Airtronic
Toulouse
22/24 Boulevard Thibaud
31084 Toulouse Cedex
Telephone: 61 40 83 94
Telex: 520 374 F

Tekelec-Airtronic
Bordeaux
Parc Club
Cadera Nord
33700 Merignac
Telephone: 56 34 84 11
Telex: 550 589 F

Tekelec-Airtronic
Lille
Immeuble Moulin 2
5 Rue du Colibri
59650 Villeneuve D' ASCQ
Telephone: 20 05 17 00
Telex: 160 011 F

Tekelec-Airtronic
Strasbourg
1 Rue Gustave Adolphe Hirn
67000 Strasbourg
Telephone: 88 22 31 51
Telex: 880 765 F

Tekelec-Airtronic
Rennes
20 Avenue de Crimee
B.P. 2246
35022 Rennes Cedex
Telephone: 99 50 62 35
Telex: 740 414 F

GERMANY
Electronic 2000
Muenchen
Stahlgruberring 12
8000 Muenchen 82
Telephone: 0 89/42 00 1-0
Telex: 522 561
Telefax: 089/42001-129

Electronic 2000
Gerlingen
Benzstrasse 1
7016 Gerlingen
Telephone: 071 56/356-0
Telex: 7-245 265

Electronic 2000
Nuernberg
AuBere Sulzbacher Str. 37
8500 Nuernberg 20
Telephone: 09 11/59 50 58
Telex: 6-26 495

Electronic 2000
Frankfurt
Schmidtstrasse 49
6000 Frankfurt/M 1
Telephone: 069/73 04 81
Telex: 4-189 486

Electronic 2000
Duesseldorf
Heinrich-Hertz-Str. 34
4006 Duesseldorf-Ekrath
Telephone: 02 11/2040 91-94
Telex: 8-586 810

Electronic 2000
Hamburg
Uberseering 25
2000 Hamburg 60
Telephone: 040/6 30 40 81
Telex: 2-164 921

Electronic 2000
Berlin
Otto-Suhr-Allee 9
1000 Berlin 10
Telephone: 030/341 70 81-82
Telex: 185 323

INDIA
Capricorn Systems
International, Inc.
1430 Tully Road, Suite 405
San Jose, CA 95122
Telephone: (408) 294-2833
Telex: 714997729 (CAPCNUI)
Telefax: (408) 294-0355

SRI Ram Associates
14 First Floor
DVG Road, Basavanagudi
Bangalore 560 004
India
Telephone: (0812) 602140
Telex: (953) 08458162 SRIS IN

ISRAEL
Vectronics Ltd.
60 Medinat Hayehehudim Street
P.O. Box 2024
Herzlia B 46120
Israel
Telephone: (052) 556-070
Telex: 342579
Telefax: (052) 556-508

ITALY
Inter-Rep
10148 Torino
Via Orbetello, 98
Telephone: 011/21.65.901
(15 linee)
Telex: 221422
Telefax: 011/21.65.915

Inter-Rep
2151 Milano
Via Gadames, 128
Telephone: 02/30.11.620 (rra.)
Telex: 221422

Inter-Rep
36016 Thiene
Via Valbella, 10 (cond. Alfa)
Telephone: 0445/36.49.61-36.38.90
Telex: 431222
Telefax: 0445/36.14.33

Inter-Rep
40129 Bologna
Via E. Mattei, 40
Telephone: 051/53.11.99 (rra)
Telex: 226079 EXEL

ALTERA INTERNATIONAL DISTRIBUTORS

ITALY (CONT'D)
Inter-Rep
50127 Firenze
Via Panciatichi, 40
Telephone:
055/43.60.392-43.60.422
Telefax: 055/43.10.35

Inter-Rep
00159 Rama
Via Tiburtina, 436
Telephone: 06/43 90 470
Telefax: 06/43 80 676

JAPAN
Japan Macnics Corporation
516 Imaiminami-Cho
Nakahara-Ku
Kawasaki-City
211 Japan
Telephone: (044) 711-0022
Telex: 28988
Telefax: (044) 711-2214

Japan Macnics Corporation
Shin-Osaka Hikari Bldg.
20-19, Higashi-Nakajima
1-Chome
Osaka-City 533
Japan
Telephone: (06) 325-0800
Telefax: (06) 325-2200

Paltek Corporation
3-8-18 Yoga
Setagaya-Ku
Tokyo 158
Japan
Telephone: (03) 707-5455
Telex: 02425205
Telefax: (03) 707-5338

KOREA
MJL Corporation
Korea Branch
Samwhan Camus Bldg.
17-3 Youido-Dong
Yeungdeungpo-Ku
Seoul, Korea
Telephone: (02) 784-8000
Telex: K28907 MJL
Telefax: (02) 784-4644

MUL Corporation - USA Office
622 Rosedale Road
Princeton, NJ 08540
Telephone: (609) 683-1700
Telex: 843457 MJL
Telefax: (609) 683-7447

MEXICO
Intectra
2629 Terminal Blvd.
Mountain View, CA 94043 USA
Telephone: (415) 967-8818
Telex: 345545

NETHERLANDS
Diode-Nederland
Meidoornkade 22
3992 AE Houten
Telephone: (03403) 91234
Telex: 47388
Telefax: (03403) 77904

NETHERLANDS (CONT'D)
Diode-Nederland
Hengelostraat
7521 PA Enschede
Telephone: (03403) 91234
Telex: 44277
Telefax: (053) 337415

NORWAY
Eltron A/S
Aslakveien 20F
0753 Oslo 7
Norway
Telephone: (02) 50 06 50
Telex: 77144

SINGAPORE
Impact Sound Pte. Ltd.
7500A Beach Road
#09-322 The Plaza
Singapore 0719
Telephone: 2914953
Telex: 39142
Telefax: 2962400

SPAIN
Selco
Paseo de la Habana, 190
28036-Madrid
Spain
Telephone: (01) 405-4213
Telex: 45458
Telefax: (01) 259-2284

SWEDEN
Fertronic AB
Norra Gubberogatan 32
S-41663 Goeteborg
Sweden
Telephone: (031) 84 84 50
Telex: 85421442
Telefax: (031) 21 51 25

Fertronic AB
Box 1279, S-171 24
Solna, Sweden
Visitaddress: Dalvagen 12
Telephone: (08) 83 00 60
Telex: 11181 fertron s.
Telefax: (08) 83 28 20

SWITZERLAND
Stolz AG
Taefernstrasse 15
5405 Baden-Daettwil
Telephone: (056) 84 01 51
Telex: 825088
Telefax: (056) 83 19 63

Stolz AG
Av. Louis-Casal 81
CH-1216 Geneve
Telephone: (022) 98 78 77

TAIWAN
Galaxy Far East Corporation
8F-6 390, Sec. 1
Fu Hsing S. Road
Taipei, Taiwan
Telephone: (02) 705 7266
Telex: 26110
Telefax: (02) 704 6729

UNITED KINGDOM
Ambar/Cascom Ltd.
Rabans Close
Aylesbury
Bucks HP19 3RS
England
Telephone: (0296) 434141
Telex: 837427
Telefax: (0296) 29670

Thame Components Ltd.
Thame Park Road
Thame, Oxon OX9 3XD
Telephone: (084421) 4561
Telex: 837917
Telefax: (084421) 7185

ALTERA SALES OFFICES

ALTERA CORPORATION
3525 Monroe Street,
Santa Clara, CA 95051
(408) 984-2800 Telex 888496
FAX (408) 248-6924

ALTERA NORTHEAST OFFICE
945 Concord Street,
Framingham, MA 01701
(617) 626-0181 Telex 948477
FAX (617) 879-0698

ALTERA MIDWEST OFFICE
200 W. Higgins Road, Suite 216,
Schaumburg, IL 60195
(312) 310-8522 TWX 510 101 1409
FAX (312) 310-0909

ALTERA SOUTHEAST OFFICE
1080 Holcomb Bridge Road
Suite 100
Roswell, GA 30076
(404) 594-7621 Telex 382207
FAX (404) 998 9830

ALTERA SOUTHWEST OFFICE
17100 Gillette Avenue
Irvine, CA 92714
(714) 474-9616
FAX (714) 261 8697

ALTERA EUROPE
Avenue de la Tanche 2
B-1160 Bruxelles
BELGIUM
(02) 674-5223 Telex 25387
FAX (02) 674-5207

ALTERA UK
42 Queen Street
Maidenhead
Berkshire, England SL61JE
(44) 628 32516
Telex: 851 940 16389
FAX (44) 628 770892