PAC1000 Programmable Peripheral Controller

Design and Applications Handbook

PAC1000
Programmable Peripheral Controller

Design and Applications Handbook

1992

47280 Kato Road, Fremont, California 94538
Tel: 510-656-5400 Facsimile: 510-657-5916 Telex: 289255

Printed in U. S. A.

General Information

1

Section Index

General Information

For additional information,
call 800-TEAM-WSI (800-832-6974).
In California, Call 800-562-6363.

Table of Contents

Company Profile

Company Description

WSI is a market leading producer of high-performance programmable peripheral integrated circuits. The company was founded in 1983 to serve the needs of system designers who need to achieve higher system performance, reduce the size and power consumption of their systems, and shorten their product development cycles in order to achieve faster market entry.

WSI produces an innovative portfolio of Programmable Peripherals as well as a broad line of high-performance non-volatile programmable PROM and EPROM memory products, both based on its patented self-aligned split-gate CMOS EPROM technology. The new Programmable Peripherals enable rapid system design of high-performance application specific controllers and related products. These devices are the first to integrate high-performance EPROM, SRAM and user-configurable logic and deliver a performance and integration breakthrough to the programmable peripherals market.

WSI's Programmable Peripherals and non-volatile memory products enable electronic designers to reduce their system size, shorten product development cycles and bring new system products to market in less time. As a result, WSI has established itself as a leading supplier of high-performance programmable solutions to a broad customer base that includes some of the world's largest and most technologically advanced electronics companies.

Technology

WSI's patented self-aligned, split-gate EPROM technology enables higher performance and greater memory densities per chip area than the traditional stacked-gate method. By developing significantly higher read current, the WSI EPROM cell has enabled the development of several memory devices that are the fastest of their type on the market. This core NVM technology is further leveraged by WSI's architecture and design innovations such as staggered virtual ground and contactless memory arrays resulting in dramatic die area savings. This high density memory capability enables WSI to provide cost-effective market leading products such as the smallest 4-Mbit EPROM on the market. WSI's proprietary NVM technology (licensed to Sharp Corporation and National Semiconductor Corporation) has enabled WSI to be first in the industry with numerous product breakthroughs in speed, high density, process innovations and packaging.

Markets and Applications

WSI's Programmable Peripheral and high-performance non-volatile memory products are used by the world's leading suppliers of advanced electronic systems in telecommunications, data processing, military, automotive and industrial markets.

Applications for the Programmable Peripherals include cellular telephones, disk drive controllers, modems, bus controllers, engine management computers, telecom switchers, motor controllers and others. High performance memory applications include digital signal processing, engineering workstations, high-speed modems, video graphics controllers, radar and others. By virtue of their high speed and programming capability, WSI products are ideally suited for these applications where designers are pushing the limits of system performance in highly competitive markets.

Product Groups

Programmable Peripherals

WSI's family of Programmable Peripherals represents a new class of programmable products. They enable system designers to reduce the size of their products, achieve lower operating power, optimize system performance and shorten product development cycles. They are the first devices to integrate high-speed EPROM, SRAM and programmable logic on a single chip. The Programmable Peripherals include the PSD3XX family, the MAP168 and the PAC1000.

PSD3XX Family: Microcontroller Peripherals with Memory

Each member of the PSD3XX family is a single-chip, field-programmable circuit that integrates all the required peripheral memory and logic elements for an embedded-control design. Programmable logic, page logic, programmable I/O ports, busses, address mapping, port address/data tracking, 256K to 1 Mb EPROM, and 16K SRAM are all on board. Advanced features such as memory paging, microcontroller port reconstruction, track mode, configuration security bit, and cascading further enhance the utility and value of the PSD3XX family. PSD3XX family devices are ideal for applications requiring high-performance, low power and very small form factors such as fixed disk control, cellular telephones, modems, computer peripherals, and automotive and military applications.

MAP168 User-Configurable Peripheral with Memory

Similar to the PSD3XX family, the high speed MAP168 integrates high-performance EPROM, SRAM, a PAD and user-configurable logic. Ideal for high-speed applications requiring expanded memory, system integration and increased data security, the 45 ns MAP168 is used with high speed digital signal processors, microprocessors and microcontrollers.

PAC1000 Peripheral Controller

The high speed PAC1000 sets a new standard for Programmable Peripheral performance, integration and functionality. The PAC1000 replaces up to 50 complex devices in high-end embedded controllers and microprocessor-based systems. Combining a CPU, 1K x 64 EPROM and extensive user-configurable logic, the PAC1000 assists its host processor with high rates of data manipulation and control, freeing the processor for other system functions. The 16 MHz PAC1000 has been designed into numerous high-performance applications such as work-station direct memory access controllers, video imaging digital signal processors, and VME bus LAN controllers.

Programmable Peripheral Development Tools

WSI's Programmable Peripheral products are supported with complete easy-to-use system development tools from both Data I/O and WSI. The Data I/O Unisite programmer can be used for production programming. The WSI tools include program development, simulation, and programming software, the IBM-PC hosted MagicPro™ Memory and Peripheral Programmer, a dial-in applications bulletin board and WSI's team of factory service and field application engineers. The menu-driven software tools run on popular customer owned computers and enable designers to rapidly configure and program the WSI part and try it in a prototype system. Additional design iterations are quickly accomodated. The system development tools increase the efficiency of the design process resulting in faster market entry for WSI's customers' products.

High-Performance Memory Products

WSI offers a broad product line of high-performance CMOS PROMs and EPROMs featuring architectures ranging from 2K x 8 to 512K x 8, plus several x16 products, with speeds ranging from 25 to 150 ns. Commercial, industrial and military products including MIL-STD-883C/SMD are available. A wide variety of package selections include plastic and hermetic, through-hole and surface mount types.

CMOS PROMs

As WSI's fastest family of products, Re-Programmable Read Only Memories (RPROMs) provide high-speed bipolar PROM pinout with matching speed and low power operation. The product family includes architectures ranging from 2K x 8 to 32K x 8 with speeds ranging from 25 to 90 ns. Commercial, industrial and military MIL-STD-883C/SMD configurations are available in a variety of hermetic and plastic package types.

"F" Family EPROMs

The high-speed "F" series EPROM family offers speeds ranging from 35 to 70 ns and architectures from 8K x 8 to 32K x 8, plus several x16 products. "F" family EPROMs are ideal for use in high-end engineering and scientific workstations, data communications and similar high-performance applications.

"L" Family Military EPROMs

WSI's "L" family military EPROM memory products feature high-density and high speed in popular JEDEC pinouts. With speeds ranging from 120 to 300 ns and architectures from 64K x 8 to 512K x 8 including several x16 products, the "L" family offers significant speed and high density benefits for developers of military avionics, communications, and control systems. The "L" family delivers world class densities from WSI's conservative 1.2 micron lithography CMOS process technology.

Manufacturing

WSI's manufacturing strategy includes utilizing multiple world-class manufacturing partners for each facet of the production process.

WSI has licensed its CMOS EPROM and logic process technology to Sharp Corporation in Japan and National Semiconductor Corporation in the USA. The Sharp facility in Fukuyama, Japan employs the most advanced sub-micron VLSI integrated circuit manufacturing equipment available including ion implantation, reactive ion etch, and wafer stepper lithographic systems. The world-class high volume National Semiconductor operation delivers low cost production of 1.2 micron CMOS technology product on 6" wafers. This low defect density manufacturing resource is capable of producing sub-micron technology product in the near future.

High-volume, low cost integrated circuit packaging and testing is performed for WSI by ANAM Electronics in Seoul, Korea, Fine Products in Hsinchu,Taiwan, National Semiconductor in Santa Clara, CA and at WSI in Fremont, CA. ANAM is the largest independent manufacturer of I.C. packaging and produces excellent product quality. Test capability ranges from simple logic devices to complex VLSI product. ANAM routinely processes a wide variety of high volume packages and enables WSI to leverage its materiel needs through ANAM's combined high-volume, low cost procurement activity. Commercial, industrial, and military grade product processing is available from ANAM.

Additional quality assurance and reliability testing are performed at WSI in Fremont, CA.

WSI's manufacturing strategy ensures the supply of double-sourced high quality, high-volume product with low variable cost and fast delivery.

Sales Network

WSI's international sales network includes several regional sales managers who direct the resources of the company to major market opportunities. Experienced technical field application engineers located in each field office assist WSI's customers during their advanced product development and match customer needs with WSI's product solutions. Over sixty manufacturer's representatives and leading national and regional component distributors in the United States, Europe and Asia round out the WSI sales network.

United States

Direct sales and field application engineering offices in Boston, Chicago, Huntsville, Philadelphia, Dallas, Los Angeles and Fremont, CA; More than 25 manufacturer's representatives for major national accounts; national distributors include Arrow/Schweber, Time Electronics and Wyle Laboratories; and regional distributors.

International

Direct WSI Sales management offices in Paris, Munich and Hong Kong; sales representatives and distributors in Germany, England, France, Italy, Sweden, Finland, Denmark, Norway, Spain, Belgium, Luxembourg, the Netherlands, and Israel. Sales representatives and distributors for the Asia/Pacific Rim region in Japan, Korea, Taiwan, Hong Kong, Singapore and Australia.

Management and Previous Affiliations:

Michael Callahan
President, CEO and
Chairman of the Board
(Advanced Micro Devices, Monolithic Memories, Motorola)

Robert J. Barker
V. P. Finance, CFO and Secretary
(Monolithic Memories, Lockheed)

John Ekiss
V. P. Marketing
(Intel, Motorola)

Thomas Branch
V. P. Worldwide Sales
(Monolithic Memories, Fairchild)

George Kern
V. P. Operations
(Advanced Micro Devices, Monolithic Memories)

Boaz Eitan
V. P. New Product and
Technology Development
(Intel)

Bob Buschini
Director of Human Resources
(General Electric, Raychem)

Financing

WSI is a privately held California corporation founded in August, 1983. The company has been financed by corporate investors, institutional investors, venture capital groups and private investors. Corporate investors are Sharp Corporation, National Semiconductor Corporation, Intergraph Corporation, and Kyocera Corporation. Venture capital investors include Accel Partners, Adler and Company, Bessemer Venture Partners, Genevest Consulting Group S. A., J. H. Whitney, Oak Investment Partners, Robertson Stephens and Co., Smith Barney Venture Corporation, and Warburg Pincus. The company has been audited annually since its inception by Ernst & Young (Arthur Young prior to 1989) and regularly reports financial information to Dunn & Bradstreet (Dunns number is 10-209-8167).

IPI-2 DISK CONTROL[illegible] DATA TRANSFERS

COMPUTER KEYBO[illegible]VE USERS MORE OPTIONS

ELECTRONIC DESIGN

A VNU BUSINESS PUBLICATION

OCTOBER 27, 1988

MICROPROGRAM AN EMBEDDED CONTROLLER FROM WAFERSCALE INTEGRATION, INC.

- ANALOG CAE GEARS UP FOR MIXED-MODE SIMULATION
- POWER-FACTOR CONTROL CIRCUITS RUN TWO STEPPERS FROM ONE BOARD

COVER FEATURE

PACKING ALL THE MAJOR BLOCKS OF A MICROPROGRAMMABLE SYSTEM, A CMOS IC EASES EMBEDDED CONTROLLER DESIGNS

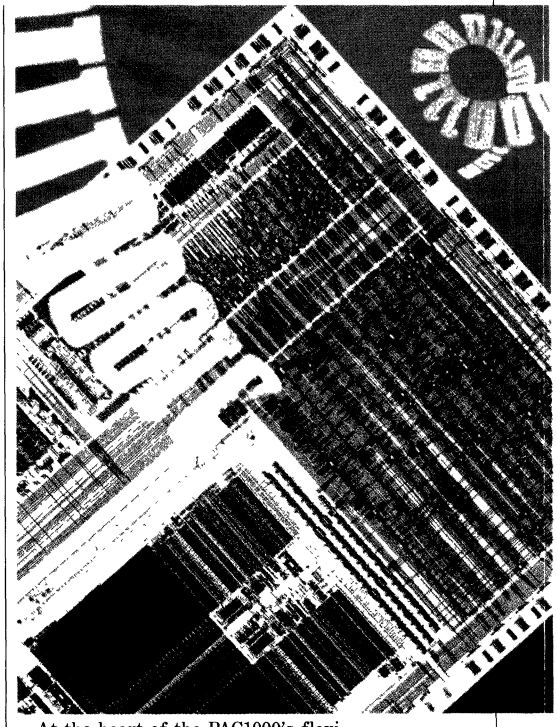
CONFIGURABLE CHIP EASES CONTROL-SYSTEM DESIGN

DAVE BURSKY

Anyone who has ever designed a high-performance controller subsyst›m using high-speed microprogrammed building blocks, programmable logic devices, gate arrays, or discrete logic realizes the difficulties in integrating the complete solution. In such a system, the chip count escalates, the operating power rises, and the development schedule lengthens.

By integrating all these functions and resources onto one high-speed CMOS chip—the PAC1000 microcontroller—WaferScale Integration Inc. has drastically reduced the chip count from the typically required 50 or so ICs to just one. At the same time, the PAC1000 slashes the power consumption from tens of watts to less than 1.5 W and cuts development time.

The PAC1000 can solve many high-end embedded control applications and is the only available circuit that can tackle system, data, and event control tasks. A C-like language and PC-hosted system-development tools simplify the creation of the control software. Users can configure the circuit as a microprocessor peripheral or as a standalone controller to meet the unique requirements of high-performance system, data, or event controllers. Each of the chip's two bidirectional 16-bit buses, its individual I/O lines, and interrupt inputs can, if necessary, be redefined during each 50-ns instruction cycle.



At the heart of the PAC1000's flexibility lies an internal microprogrammable architecture, including a 16-bit CPU, a fast 10-bit microsequencer, a 32-word-by-16-bit register file, and a 1kword-by-64-bit high-speed EPROM. As product planning manager Yoram Cedar explains, since the circuit executes any of its instructions in one clock cycle, the controller delivers a raw throughput of

COVER: USER-CONFIGURABLE CONTROLLER

20 MIPS.

Every instruction of the PAC1000 can perform as many as three simultaneous operations: program control, CPU functions, and output control, with all possible combinations allowed. Cedar claims the more powerful instruction format, combined with the higher clock speed, yields a five- to tenfold performance improvement, compared with other one-chip microcontrollers. The high throughput suits many tasks well. It has already found homes in radar, communications, video-graphics, I/O subsystems, bus and DMA controllers, and disk-drive-controllers.

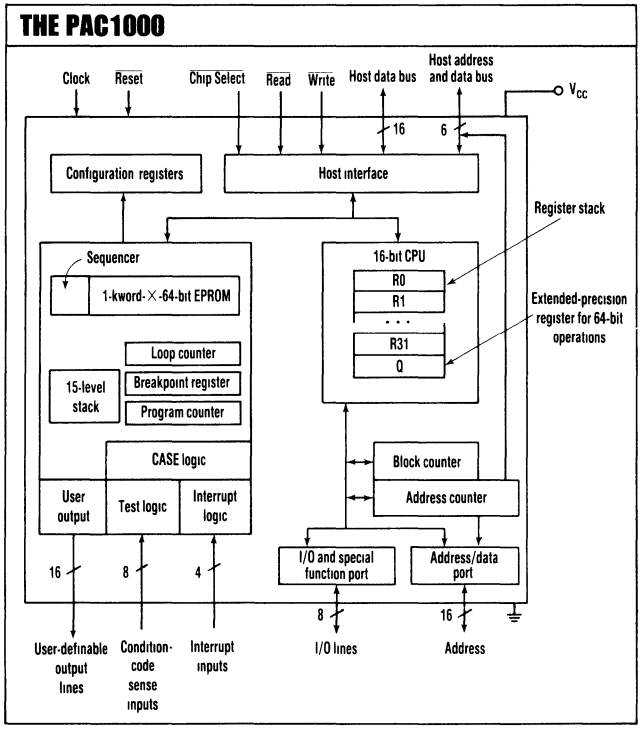
Besides the CPU, register file, and sequencer, the chip includes an auxiliary Q-register for double-word operations, an 8-input interrupt controller, 16 output control lines, 8 bidirectional I/O lines, scan-test and CASE program test logic, and a 22-bit external address bus *(Fig. 1, top)*.

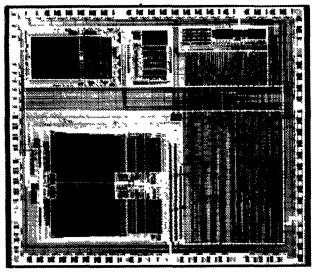
Also, Cedar emphasizes, the circuit deals much more rapidly with interrupts than most controllers do, and that serves embedded control applications well. The chip changes program flow in either of two ways. First, it has four user-definable interrupt input lines plus four dedicated internal interrupts that require just 100 ns, at most, to alter the program flow. Second, another set of input lines—22 condition-code inputs (8 external and 14 internal)—let the processor alter the program flow with condition calls and program jumps in just one 50-ns instruction cycle.

And if on-chip resources don't quite match an application's requirements, chip modifications can be done for large-volume users. The circuit was designed with the company's standard-cell library, and many of the chip's sections are actually cells in WaferScale's library *(Fig. 1, left)*. Noticeable on the chip's left side are the large cells that include the 64-kbit EPROM block on the bottom and the 16-bit CPU on the upper left. On the chip's right side, random logic performs the control and interface functions; small standard cells are used to create those circuits.

For every instruction, a dedicated field specifies the bit pattern on the output lines. Also, designers can individually program eight I/O lines as inputs or outputs or to perform special functions under the control of the chip's mode and I/O registers. The special functions turn the I/O lines into control signals that allow various features and flags to indicate several status conditions. In addition to the eight I/O lines, the circuit has two 16-bit bidirectional buses that go on and off the chip: One links with the host; the other is the upper 16 bits of the address/data bus. Another 16 lines are dedicated, user-programmable latched output lines. These can be changed on a cycle-by-cycle basis.

Thanks to all its buses and control signals, the PAC1000 microcontroller operates as either a memory-





1. PACKING A 16-bit microprogrammable central processor with a 32-word register file, a 1-kword-by-64-bit microcode UV EPROM, sequencer, and other configurable resources, the PAC1000 user-configurable microcontroller from WaferScale Integration delivers a raw instruction throughput of 20 MIPS at 20 MHz (top). Designers can add or alter various blocks to customize versions for high-volume users (left).

COVER: USER-CONFIGURABLE CONTROLLER

mapped peripheral to a microprocessor to offload the CPU *(Fig. 2a)* or as a standalone controller running from its own internally or externally stored program *(Fig. 2b)*. As a peripheral, the chip ties into the host with a straightforward bus interface—a 16-bit data bus and a 6-bit address bus to access the internal resources of the PAC1000—and the standard Chip Select, Read, and Write control lines. In the standalone mode, the chip typically runs the application program from its internal memory and uses its 16-bit output bus and 8-bit I/O port to control the application and communicate to a host system.

To handle multiple operations in parallel, the chip internally takes advantage of a long—64-bit—microcode word so that each word can control multiple sections of the circuitry. The on-chip microcode storage area consists of a fast, reprogrammable UV EPROM, organized as 1 kword by 64 bits. Since the EPROM is read only by the on-chip logic, it doesn't need high-current output buffers, which slow down the memory access. Thus, the EPROM contents can be read very quickly—the chip's 20-MHz version accesses memory in just 30 ns, well within the CPU's 50-ns instruction cycle time. The memory is also secure. Users can program a security bit to prevent an external system from extracting the code from the memory array.

Besides its own program memory, the chip also has a separate address/data bus that can be programmed for either 16 or 22 address lines (with 64-kword or 4-Mword off-chip addressing ranges, respectively). The address generator for the bus is separate from the sequencer that addresses the program memory. The PAC1000 can therefore execute a program while it's using the address bus to move data from memory into the on-chip register file or to an externally controlled device.

The address bus, in fact, can serve as a simple direct-memory-access controller when used with the on-chip 22-bit address counter and 16-bit block counter. This DMA controller can transfer data from external memory to the on-chip register file or to an external device.

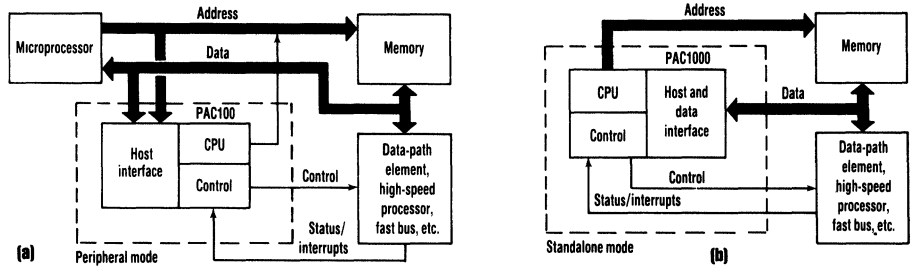
An eight-word FIFO register lets a host microprocessor asynchronously load commands or data into the controller. The 22-bit word length in the FIFO register is employed, so that if data values are to be loaded into the register file, the lower 16 bits of the 22-bit word sent over the host data bus represent the data, and the next five bits—the lower five bits of the host-interface address bus—represent the register location into which the data will be loaded (R0 to R31). The sixth bit of the host-interface address bus signifies whether the word loaded into the FIFO register is a command or data word. If it's a command, the lower 10 bits of the host-data bus are used as a branch address to one of the 1024 memory locations in the EPROM.

The 10-bit sequencer addresses the 1,024 words of program memory and has a 15-level stack that permits multiple subroutine calls to occur without forcing the program to go back to a higher level before calling the next subroutine. Besides having more levels in the stack than WaferScale's 5910 microsequencer, the enhanced sequencer block has a 10-bit loop counter that cuts overhead in programs for loops and nested loops. The application program can load the counter with a constant or a value calculated in the CPU.

Because programming fast, embedded controllers can get complicated, the company includes on-chip programming and test features to ease system development. For starters, a 10-bit breakpoint register simplifies real-time debugging. It can be loaded from either of two sources—a value stored in a CPU register or a constant value specified in the program memory. When the program memory address matches the register contents, the register issues an interrupt, which a service routine in memory could then react to.

Test and CASE logic on the chip also aids program and hardware testing. The condition-code logic responds to 22 different program test conditions that can be tested for true

PERIPHERAL OR STANDALONE



2. MULTIPLE BUSES, AN ON-CHIP ADDRESS GENERATOR, and sequencer blocks let the microcontroller operate as a memory-mapped peripheral to offload the host microprocessor (a). Or it can be operated as a standalone controller (b).

COVER: USER-CONFIGURABLE CONTROLLER

SAMPLE PROGRAM FOR PAC1000 MICROCONTROLLER

```
/* control memory read/write based on CC0 */
segment memcon,
  enmem   equ   h'0002',                  /* output control constants          */
  dismem  equ   h'0040',
  wr      equ   h'0000',
  rd      equ   h'1000',
start
  IF CC0, OUT enmem,                      /* enable memory                     */
    FOR 6, AOR  = R0 + R1, OUT wr,        /* store begin addr in AOR and loop  */
           AOR  = AOR + 4, OUT rd,        /* inc addr by 4 and do rd/wr        */
    ENDFOR, OUT wr,                       /* end loop body                     */
  ELSE, OUT dismem,                       /* disable mem if CC0 is not true    */
  ENDIF,
end,
```

3. THE HIGH-LEVEL LANGUAGE developed by WaferScale employs C-language-like structures to let designers easily develop complex configuration microcode.

or not-true results. Up to four conditions can be tested simultaneously. Tests can check for the state of various flags or register contents.

The processor handles two types of CASE operations: standard and priority. A CASE group consists of a combination of four test conditions that can be tested in a single cycle. In that same cycle, the PAC1000 branches to any one of 16 locations, depending on the status of the four inputs to the CASE group being tested. The priority CASE instruction operates on internal and external interrupt conditions and treats interrupts as prioritized test conditions. The priority encoder generates a branch to the highest-priority condition.

Thanks to all its on-chip resources, the PAC1000 is a powerful one-chip controller, housed in a windowed, 88-lead pin-grid-array package or an 84-lead ceramic leaded chip carrier. An 84-lead plastic leaded chip carrier package (the one-time-programmable version) is also available. Because the chip employs an EPROM to hold the program, revisions to the code are no more difficult than reprogramming a standard EPROM. Prototype systems and production products can benefit from the ability to revise the code at the last minute.

To alleviate the complexity of microcode program development, WaferScale has assembled a series of PC-hosted system-development tools (PAC-SDT). These make the PAC1000 as easy to program as any one-chip microcontroller. A simple example of a multiple-command expression in the C-like language lets designers combine operations such as FOR6,AOR=R0+R1,OUT WR (loop for six cycles, add the contents of registers R0 and R1 and store the result in the AOR register, output the value WR) in one word *(Fig. 3)*.

The toolset has a system-entry language, a functional simulator, and a device programmer (MagicPro). The system-entry, language software is the most critical part. The high-level language uses a structure similar to C's and practically eliminates writing routines in machine or assembly code. But designers who are more comfortable working on that level can write machine-code routines.□

PROGRAMMABLE PERIPHERALS

SINGLE-CHIP CMOS USER-CONFIGURABLE PERIPHERAL WITH MEMORY – COMMERCIAL & MILITARY

Part No.	Description	Speed (ns) Comm'l	Speed (ns) Military	Availability Samples	Availability Prodn	Package Selection J	L	Q	X
PSD301	Programmable Microcontroller Peripherals with Memory, x8/x16, 256Kb – 1Mb EPROM; 16K SRAM, PAD, System Features.	120		NOW	Q1 '92	•	•	•	
		150-200		NOW	NOW	•	•	•	•
			200		NOW		•		•
PSD311		120		NOW	Q1 '92	•	•	•	
		150-200		NOW	Q1 '92	•	•	•	•
			200		Q1 '92		•		•
PSD302		120		NOW	Q1 '92	•	•		
		150-200		NOW	Q1 '92	•	•		
PSD312		120		NOW	Q1 '92	•	•		
		150-200		NOW	Q1 '92	•	•		
PSD303		120		Q1 '92	Q1 '92				
		150-200		Q1 '92	Q1 '92				
PSD313		120		Q1 '92	Q1 '92				
		150-200		Q1 '92	Q1 '92				
MAP168	DSP Peripheral with Memory Features. 128K Bits EPROM, 32K Bits SRAM Programmable Address Decoder (PAD) Configurable. x8 or x16	45-55		NOW	NOW	•	•	•	•
			55		NOW	•		•	

HIGH-PERFORMANCE CMOS USER-CONFIGURABLE EMBEDDED CONTROLLER – COMMERCIAL & MILITARY

Part No.	Description	Speed (ns) Comm'l	Speed (ns) Military	Availability Samples	Availability Prodn	Package Selection Q	X	V
PAC1000	Programmable Peripheral Controller optimized for High-Performance Control Systems Key Features Include. 16-Bit CPU, 16-Bit Address Port, 16-Bit Output Control, 8-Bit I/O Port and Configuration Registers	12MHz		NOW	NOW	•	•	•
			12MHz	NOW	NOW		•	•
		16MHz		NOW	NOW	•	•	

HIGH-PERFORMANCE CMOS USER-CONFIGURABLE MICROSEQUENCER/STATE MACHINE – COMMERCIAL & MILITARY

Part No.	Description	Speed (ns) Comm'l	Speed (ns) Military	Availability Samples	Availability Prodn	Package Selection J	L	S	T
SAM448	User-Programmable Microsequencer for Implementing High-Performance State Machines. Includes EPROM integrated with Branch Control Logic, Pipeline Register, Stack and Loop Counter and 768 Product Terms	20-25MHz		NOW	NOW	*	•	*	•
			20MHz	NOW	NOW		•		•

*J and S packages not available in 25MHz

SOFTWARE DEVELOPMENT TOOLS †

Part No.	Includes	Availability
PSD - GOLD	Contains PSD301/MAP168 Software, Users Manual, WS6000 MagicPro (PC Based Programmer), WS6014(J/L) or WS6015(X) Adapter and 2 Sample Devices	NOW
PSD - SILVER	Contains PSD301/MAP168 Software and Users Manual	NOW
PAC1000 - GOLD	Contains PAC1000 Software, Users Manual, WS6000 MagicPro (PC Based Programmer), WS6010 (X) Adapter and 2 Sample Devices	NOW
PAC1000 - SILVER	Contains PAC1000 Software and Users Manual	NOW
SAM448 - GOLD	Contains SAM448 Software, Users Manual, WS6000 MagicPro (PC Based Programmer), WS6008(T) or 6009(C,J,L) Adapter and 2 Sample Devices	NOW
SAM448 - SILVER	Contains SAM448 Software and Users Manual	NOW
MEMORY - SILVER††	Contains WSI EPROM/RPROM Programming Software and Users Manual	NOW

† 1) All Development Systems include: 12 Month Software Update Service, access to WSI's 24 Hour Electronic Bulletin Board.
2) Package adaptor must be specified when ordering any "Gold" system

†† 1) Memory-Silver is included in all development systems.

NON-VOLATILE MEMORY

CMOS PROMs – COMMERCIAL

				Package Selection					
Part No.	Architecture	Description	Speed (ns)	D	J	L	P	S	T
WS57C191B	2K x 8	16K CMOS PROM	35-55	•	•		•		
WS57C291B	2K x 8	16K CMOS PROM	35-55					•	•
WS57C45	2K x 8	16K CMOS Reg. PROM	25-35					•	•
WS57C43B	4K x 8	32K CMOS PROM	35-70	•	•			•	•
WS57C49B	8K x 8	64K CMOS PROM	35-70	•	•			•	•
WS57C49C	8K x 8	64K CMOS PROM	35-70	•	•			•	•
WS57C51C	16K x 8	128K CMOS PROM	35-70	•	•	•			•
WS57C71C	32K x 8	256K CMOS PROM	45-70	•	•	•			•

CMOS PROMs – MILITARY

				DESC	Package Selection						
Part No.	Architecture	Description	Speed (ns)	SMD	C	D	F	H	K	T	Z
WS57C191B	2K x 8	16K CMOS PROM	45-55	•	•	•	•				•
WS57C291B	2K x 8	16K CMOS PROM	45-55	•					•	•	
WS57C45	2K x 8	16K CMOS Reg PROM	35-45	•	•		•	•	•	•	
WS57C43B	4K x 8	32K CMOS PROM	45-70		•					•	
WS57C49B	8K x 8	64K CMOS PROM	45-70	•	•	•	•			•	
WS57C49C	8K x 8	64K CMOS PROM	45-70	•	•	•	•			•	
WS57C51C	16K x 8	128K CMOS PROM	45-70		•	•				•	
WS57C71C	32K x 8	256K CMOS PROM	55-70		•	•				•	

NON-VOLATILE MEMORY (Cont.)

HIGH-SPEED CMOS EPROMs – COMMERCIAL

Part No.	Architecture	Description	Speed (ns)	Package Selection D	J	L	T
WS57C64F	8K x 8	High-Speed 64K CMOS EPROM	55-70	•	•		
WS57C128F	16K x 8	High-Speed 128K CMOS EPROM	55-70	•			
WS57C128FB	16K x 8	High-Speed 128K CMOS EPROM	35-45	•	•	•	
WS57C256F	32K x 8	High-Speed 256K CMOS EPROM	45-70	•	•	•	•

HIGH-SPEED CMOS EPROMs – MILITARY

Part No.	Architecture	Description	Speed (ns)	DESC SMD	Package Selection C	D	T	L
WS57C64F	8K x 8	High-Speed 64K CMOS EPROM	70	•	•	•		
WS27C64F	8K x 8	Low-Power 64K CMOS EPROM	90	•	•	•		
WS57C128F	16K x 8	High-Speed 128K CMOS EPROM	70	•	•	•		
WS57C128FB	16K x 8	High-Speed 128K CMOS EPROM	45-55		•	•		
WS27C128F	16K x 8	Low-Power 128K CMOS EPROM	90	•	•	•		
WS57C256F	32K x 8	High-Speed 256K CMOS EPROM	55-70	•	•	•	•	
WS27C256F	32K x 8	Low-Power 256K CMOS EPROM	90	•	•	•		•

CMOS EPROMs – COMMERCIAL

Part No.	Architecture	Description	Speed (ns)	Package Selection D	J	L
WS27C010L	128K x 8	Low-Power 1 Meg CMOS EPROM	120-150	•	•	•
WS27C210L	64K x 16	Low-Power 1 Meg CMOS EPROM	100-200	•	•	•

CMOS EPROMs – MILITARY

Part No.	Architecture	Description	Speed (ns)	DESC SMD	Package Selection C	D	L	T
WS27C256L	32K x 8	Low-Power 256K CMOS EPROM	120-250	•	•	•		•
WS27C512L	64K x 8	Low-Power 512K CMOS EPROM	120-200	•	•	•		•
WS27C010L	128K x 8	Low-Power 1 Meg CMOS EPROM	150-200	•	•	•	•	
WS27C210L	64K x 16	Low-Power 1 Meg CMOS EPROM	150-200		•	•	•	

CMOS BIT SLICE AND LOGIC

Part No.	Description	Speed		Package Selection							
		Comm'l	Military	B	G	J	K	L	P	S	Y
WS5901	4-Bit CMOS Bit Slice Processor	32,43 MHz	32,43MHz						•		•
WS59016	16-Bit CMOS Bit Slice Processor	15 MHz	12 5MHz	•		•		•			
WS59032	32-Bit CMOS Bit Slice Processor	26 4,33 MHz	23.6,29 MHz		•						
WS5910	CMOS Microprogram Controller	20,30 MHz	20,30 MHz						•		•
WS59510	16K x 16 CMOS Multiplier-Accum	30-50 ns			•	•			•		
WS59520	CMOS Pipeline Register	Tpd = 22ns	Tpd = 24ns				•			•	
WS59521	CMOS Pipeline Register	Tpd = 22ns	Tpd = 24ns				•			•	
WS59820	CMOS Bi-Directional Register	Tpd = 23ns	Tpd = 25ns		•	•					

WSI PACKAGE DESCRIPTIONS

Package Code	Description	Window	Surface Mount	Plastic/OTP
B/R	Ceramic Sidebrazed Dip	N/Y	N	–
C	Ceramic Leadess Chip Carrier (CLLCC)	Y	Y	–
C/Z	Ceramic Leadless Chip Carrier (CLLCC)	Y/N	Y	–
D/Y	0 600" Ceramic Dip	Y/N	N	–
F/H	Ceramic Flatpack	Y/N	Y	–
J	Plastic Leaded Chip Carrier (PLDCC)	N	Y	Y
L/N	Ceramic Leaded Chip Carrier (CLDCC)	Y/N	Y	–
P	Plastic Dip	N	N	Y
Q	Plastic Quad Flatpack (PQFP)	N	Y	Y
S	0 300" Plastic Dip	N	N	Y
T/K	0 300" Ceramic Dip	Y/N	N	–
V	Ceramic Quad Flatpack (CQFP)	Y	Y	–
X/G	Ceramic Pin Grid Array (CPGA)	Y/N	N	–

47280 Kato Road
Fremont, California 94538-7333
Tel· 510-656-5400 Fax: 510-657-5916
800-TEAM-WSI (800-832-6974)
In California 800-562-6363

WSI REGIONAL HOTLINES

USA Northwest:	Tel: 510-656-5400	Fax: 510-657-5916
USA Southwest:	Tel: 714-753-1180	Fax: 714-753-1179
USA Midwest:	Tel: 708-882-1893	Fax: 708-882-1881
USA Southeast:	Tel: 214-680-0077	Fax: 214-680-0280
USA Mid-Atlantic:	Tel: 215-638-9617	Fax: 215-638-7326
USA Northeast:	Tel: 508-685-6101	Fax: 508-685-6105
Europe (France):	Tel: 33 (1) 69-32-01-20	Fax: 33 (1) 69-32-02-19
Europe (Germany)	Tel: (49) 89.23 11.38.49	Fax: (49) 89.23.11.38.11
Asia (Hong Kong)	Tel: 852-575-0112	Fax: 852-893-0678

Printed in U S A 2/92 • Revision 8 5

Ordering Information
High-Performance CMOS Products

PAC1000 **–12** **D** **I** **B**

Basic Part Number

Manufacturing Process:

(Blank) = WSI Standard Manufacturing Flow

B = MIL-STD-883C Manufacturing Flow

Operating Temperature Range:

(Blank) = Commercial: 0° to +70°C
V_{CC}: +5V ± 5%

I = Industrial: –40° to +85°C
V_{CC}: +5V ± 10%

M = Military: –55° to +125°C
V_{CC}: +5V ± 10%

Package:

		Window
A	= PPGA Plastic Pin Grid Array	No
B	= 0.900″ Size Brazed Ceramic DIP	No
C	= CLLCC Ceramic Leadless Chip Carrier	Yes*
D	= 0.600″ CERDIP	Yes
F	= Ceramic Flatpack	Yes*
G	= CPGA Ceramic Pin Grid Array	No
H	= Ceramic Flatpack	No*
J	= Plastic Leaded Chip Carrier	No*
K	= 0.300″ Thin CERDIP	No
L	= CLDCC Ceramic Leaded Chip Carrier	Yes*
N	= CLDCC Ceramic Leaded Chip Carrier	No*
P	= 0.600″ Plastic DIP	No
Q	= Plastic Quad Flatpack	No*
R	= Ceramic Side Brazed	Yes
S	= 0.300″ Thin Plastic DIP	No
T	= 0.300″ Thin CERDIP	Yes
V	= CQFP Ceramic Quad Flatpack	Yes
W	= Waffle Packed Dice	–
X	= Ceramic Pin Grid Array	Yes
Y	= 0.600″ CERDIP	No
Z	= CLLCC	No

Speed:

– 12 = 12 MHz
– 16 = 16 MHz
Etc.

*Surface Mount

PAC1000

Section Index

For additional information,
call 800-TEAM-WSI (800-832-6974).
In California, Call 800-562-6363.

Programmable Peripheral
PAC1000 Introduction
Programmable Peripheral Controller

Overview

The PAC1000 Programmable Peripheral Controller is the first of a generation of products intended for applications in high-end embedded control where high-speed data processing, interface or control is needed. The PAC1000 replaces a board full of discrete components such as standard logic, FIFO, EPROM for microcode store, ALU, SEQUENCER, register files and PAL/PLD/PGA. To shorten the time-to-market for the system designer, a high-level software development language is used. This contrasts with the myriad state-machine entry, schematic entry, and place and route tools that would be needed for a discrete design using PAL, PLD, PGA or gate arrays.

The PAC1000 architecture is flexible and enables the system designer to customize the PAC1000 to optimize application performance. The PAC1000 is composed of three basic sections: a CPU for data processing, a programmable instruction control unit that determines the next address to the microcode store through polling condition codes or responding to interrupts, and a host interface to asynchronously load data from the host. Registered input/outputs are used to synchronize with the system.

As a result of integrating logic and EPROM memory into the PAC1000 and defining a high-level language for programming both, time-to-market and board space is reduced and reliability increased. The PAC1000 is currently used in applications such as Intelligent DMA controller, FDDI buffer controller, Frame buffer controller, LAN communications controller, disk controller, and I/O controller. For further details on the PAC1000 see Application Note 10.

Contents

Programmable Peripheral

PAC1000

Programmable Peripheral Controller

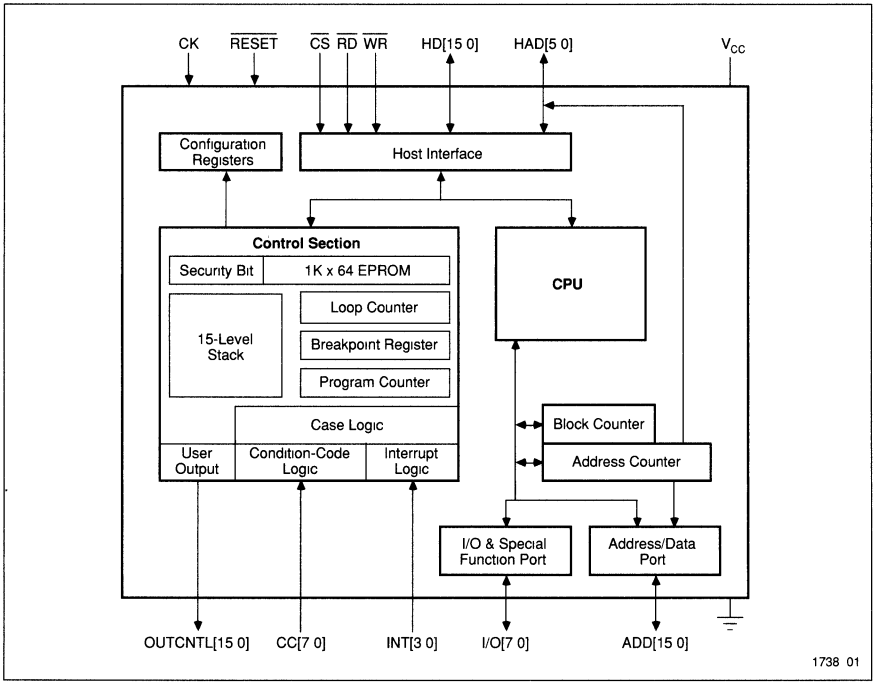
Preliminary

Features

- High-Performance Programmable Peripheral Controller
  - 16 MHz Instruction Execution, Output Port, and Address Bus
- Single-Cycle Control Architecture
  - One Cycle Per Instruction
- 16-bit CPU
  - Arithmetic Operations, Logic Operations, 33 General-Purpose Registers
- Address Generation
  - Up To 4 Mbytes Address Space
- High-Level Development Tools – System Entry Language, Functional Simulator, and Device Programmer
- Re-Programmable Program Store
  - 1K x 64-Bit EPROM for CPGA Package
  - 1008 x 64-Bit EPROM for PQFP Package
- Re-Programmable Program Store
  - On-Board 1K x 64-Bit EPROM
- Two Operating Modes
  - Host Processor Peripheral or Stand-Alone Controller
- Security
  - For EPROM Program Memory
- Package Availability
  - 88-Pin Ceramic PGA and 100-Pin PQFP

2

Figure 1. PAC1000 Block Diagram



General Description

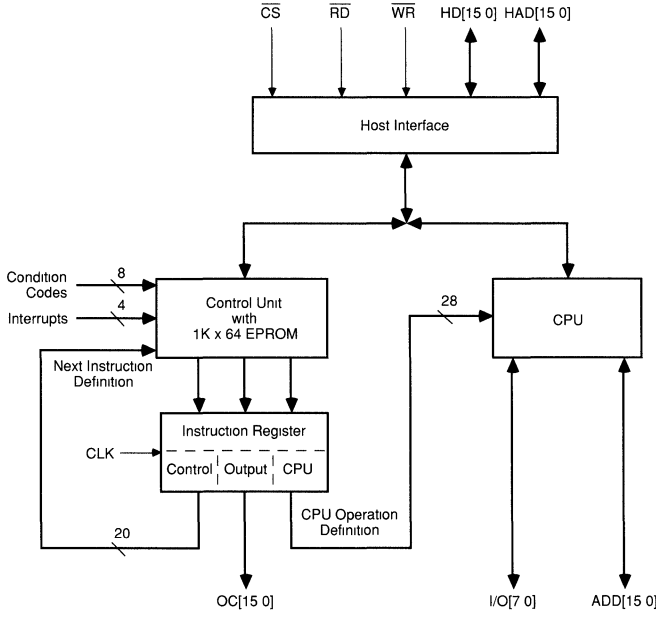
The PAC1000 Programmable Peripheral Controller is based upon an architecture that enables it to execute complex instructions in a single clock cycle. Each PAC1000 instruction can perform three simultaneous operations: Program Control, CPU functions, and Output Control, as shown in Figure 2. The PAC1000 can also perform address generation or event counting simultaneously with instruction execution. The PAC1000 is also capable of performing a conditional test on up to four separate conditions and multi-way branching in a single cycle.

The PAC1000, with its System Development Tools, matches the development cycle and ease of use of any standard microcontroller.

The high performance and flexibility of the PAC1000 were previously available only to designers who could afford the long development cycle, high cost, high power, and large board space requirements of a building-block solution (i.e., Sequencer, Microcode Memory, ALU, Register File, PALs, etc.)

The unique capabilities of PAC1000 are easily utilized with System development tools, which include a PACSEL C-like System Entry Language, a PACSIM Functional Simulator, and a MagicPro™ Device Programmer. All System Development Tools are PC-based and will operate on an IBM-XT, AT, PS2 or compatible machine. For more information, contact your nearest WSI sales office or representative.

Figure 2. Single-Cycle Control Architecture



Important Features:

- One cycle per instruction
- 16 MHz instruction execution rate
- Every instruction executes 3 parellel operations (Control, Output, CPU)

1738 02

Table 1. Pin Description

Signal	I/O	Description
HD[15:0]	I/O	*Host Data.* PAC1000 Data I/O Port via the Host Interface. Can also be configured to generate 16-bit address or status. Can serve as a general-purpose Data I/O Port.
HAD[5:0]	I/O	*Host Address.* Can be configured to output the lower six bits of the 22-bit Address Counter; can be used as a Host Interface function address, or as a general-purpose 16-bit port.
$\overline{CS}$	I	*Chip Select (active low).* Used with $\overline{RD}$ and $\overline{WR}$ to access the device via the Host Interface.
$\overline{RD}$	I	*Read Enable (active low).* Used with $\overline{CS}$ to output Program Counter, Status Register, or Data Output Register to HD[15:0] bus lines.
$\overline{WR}$	I	*Write Enable (active low).* Used with $\overline{CS}$ to write HD Bus data via the Host Interface into the PAC1000 FIFO.
CK	I	*Clock.*
CC[7:0]	I	*Condition Codes.* Condition-code inputs for use with Call, Jump, and Case instructions.
INT[3:0]	I	*Interrupts.* General-purpose, positive-edge-triggered interrupt inputs.
$\overline{RESET}$	I	*Asynchronous Reset (active low).* Resets Input/Output registers and counters, tri-states all I/O, and sets the Program Counter to 0.
OUTCNTL[15:0]	O	*Output Control.* User-defined Output Port. May be programmed to change value every cycle.
ADD[15:0]	I/O	*Address Port.* Outputs data from Address Counter or Address Output Register when configured as an output. When configured as an input, reads data to Address Input Register.
I/O[7:0]	I/O	*Input or Output Port.* Individually configurable bidirectional bus. As simple I/O, outputs come from the I/O Output Register, and inputs appear in the I/O Input Register. As special I/O functions, provides status, handshaking, and serial I/O. Alternatively, these signals can be used to extend the OUTCNTL or ADD lines.

Architectural Overview

The PAC1000 is a programmable peripheral controller optimized for high-performance control systems. The primary architectural elements, shown in Figure 3, are the Control Section, 16-bit CPU, Host Interface, 16-bit Address Port, 16-bit Output Control, 8-bit I/O Port, and Configuration Registers.

The PAC1000 can be used as a stand-alone embedded controller or as a peripheral to a host. In the latter case, the Host Data (HD) and Host Address (HAD) buses, together with the $\overline{CS}$, $\overline{RD}$, and $\overline{WR}$ pins allow for direct connection to a host bus. User-defined commands to the Control Section or data to the CPU can be loaded through the Host Interface.

In the stand-alone mode, the Host Interface ports can be used as additional address, data or I/O ports using the Data Output Register (DOR) and Data Input Register (DIR). The ADD port can be used to generate addresses through the Address Output Register (AOR) or the Address Counter. A DMA channel can be formed on the Host Interface using these and the Block Counter (BC) register. In addition, the ADD port can be used as a data bus or an I/O port, depending on how the chip is configured. Each pin in the I/O port can be configured individually as input, output, or special function. The special functions allow the control of internal PAC1000 elements (counters, I/O buffers) by other board elements.

The 16-bit CPU is highly parallel and can operate on operands from the 32x16-bit register file, miscellaneous register (AOR, AIR, DOR, DIR, Q, etc.), or constants loaded from the internal program-store EPROM.

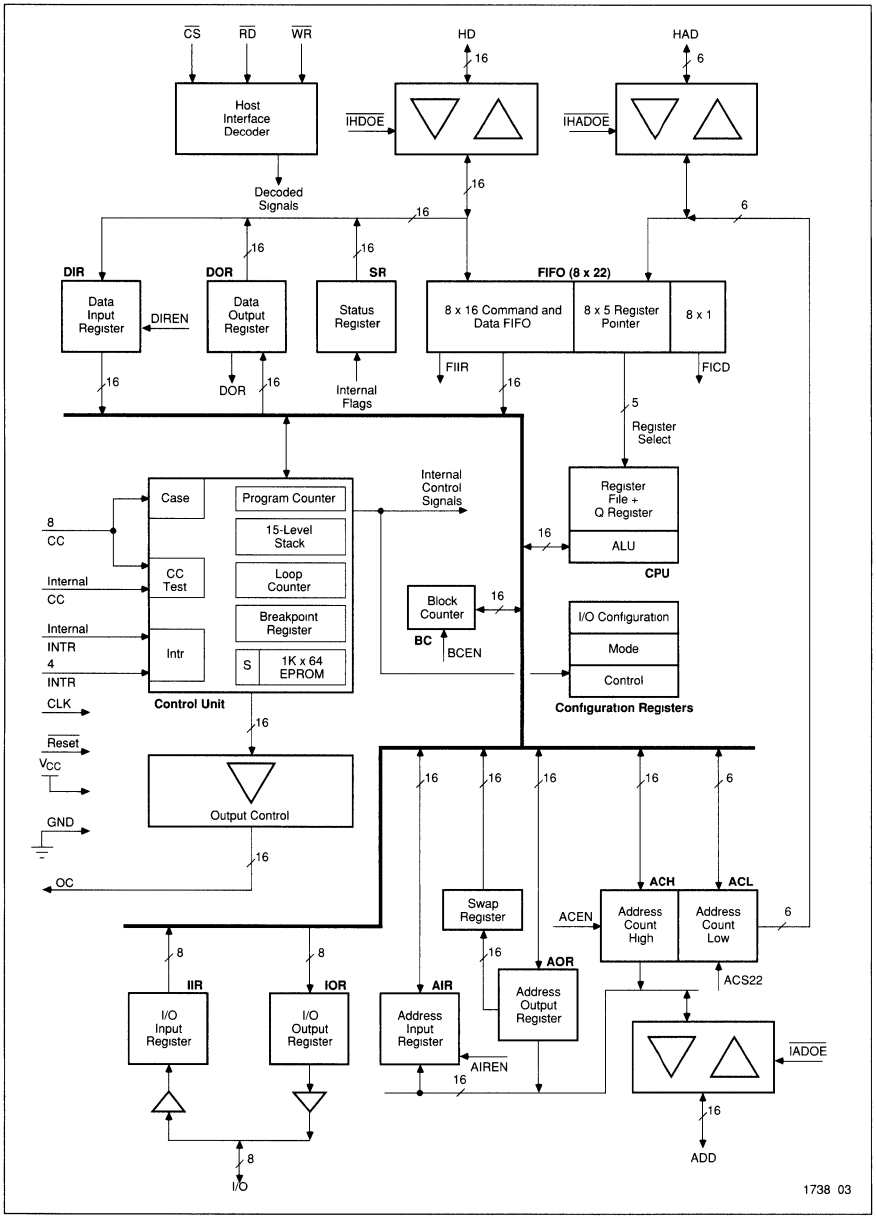
The internal and external operations of the PAC1000 are controlled by the Control Section. The 16 Output Control (OC) lines are general-purpose outputs. Each of them can be changed independently every clock cycle. They provide a very fast means to control various processes outside the chip.

In every clock cycle, one instruction is executed. Each instruction consists of up to three operations in parallel:

- Instruction Fetch—the next instruction is fetched from the 1Kx64 EPROM by the Program Control.
- Execution—the CPU executes an instruction.
- Output—placed on the Output Control (OC) lines.

Program flow can be changed through the condition-code inputs in one clock cycle or through the interrupt inputs after two clock cycles. Single-cycle 16-way branches can be done using the Case instruction, which samples four condition codes per cycle. Nested loops and subroutines can be carried out with the 15-level stack and the loop counter. The chip configuration can be changed in any cycle by loading the Configuration Register using the Program Control instruction portion.

Figure 3. Detailed Block Diagram



Operational Modes

The two basic modes of operation for the PAC1000 are either as a memory-mapped peripheral (Figure 4) or as a stand-alone controller (Figure 5).

In the peripheral mode, the host processor can asynchronously interface with the PAC1000.

Figure 4. Peripheral Mode

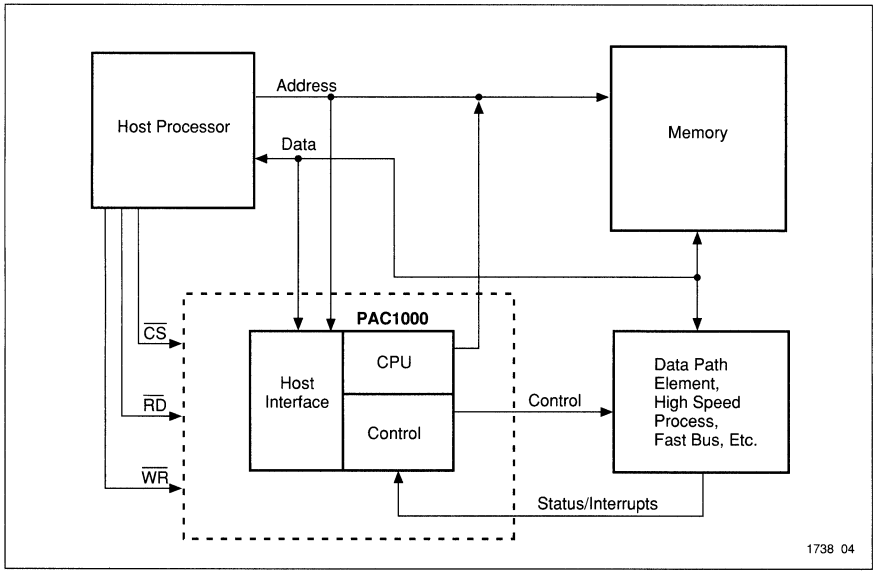
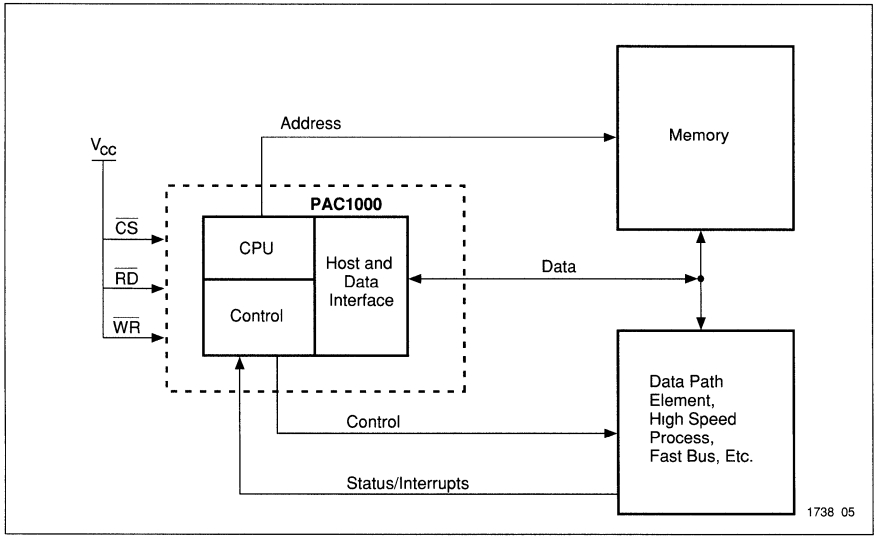


Figure 5. Stand-alone Mode



Host Interface

The Host Interface section of the PAC1000, shown in Figure 6, includes the Input Command/Data FIFO, Input/Output Data Registers, and the Status Register.

FIFO

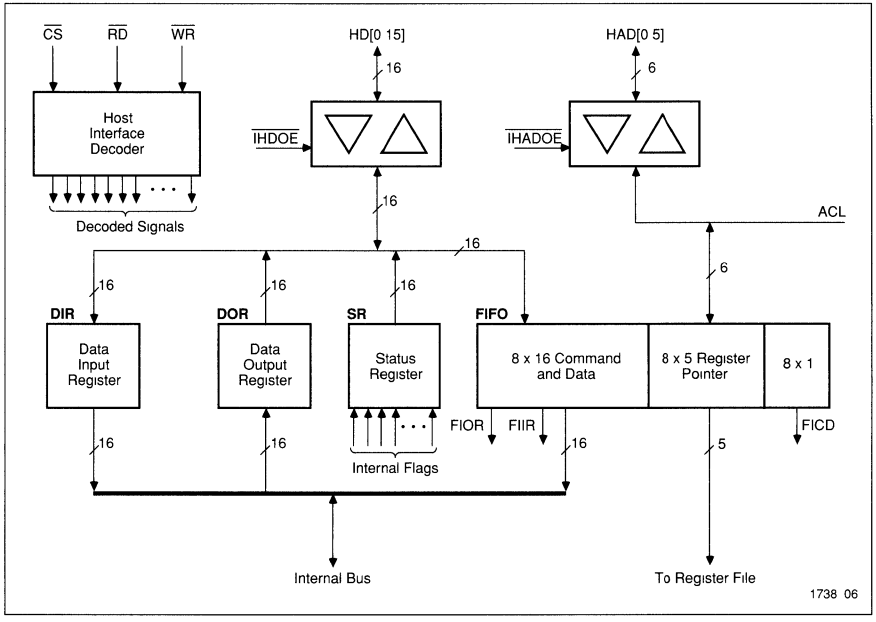
When the PAC1000 serves as a peripheral to a host, the FIFO is used to asynchronously load commands or data into the PAC1000. In order to write into the FIFO, $\overline{CS}$ and $\overline{WR}$ must have low-to-high transitions. The information written into the FIFO is specified by the 16-bit Interface Data bus (HD) and the 6-bit Host Address bus (HAD). Since the FIFO is used only to buffer data and commands from a host, it is inoperative when the PAC1000 is in stand-alone mode.

Bit five of the HAD bus specifies whether the input to the FIFO is command (HAD5=1) or data (HAD5=0). HAD5 is connected to the FICD internal Condition Code that can be sampled by the Control Section. If a command is written, then the lower 10 bits of the HD bus are used as the branch address for one of the 1024 locations in the Program Memory EPROM. At that location a user defined command or subroutine should exist which executes the needed operation. If the information is data, then the lower 5 bits of the HAD bus specify which CPU register is to be loaded from the HD bus.

This method of operation allows the host to access the PAC1000 as a memory-mapped peripheral.

2

Figure 6. Host Interface Architecture



Host Interface (Con't)

An example of FIFO usage is shown in Figure 7. When command or data information is available in the FIFO, the FIFO Output Ready (FIOR) interrupt (interrupt 5) triggers. If the FIOR interrupt is masked, then the FIOR status may be polled under program control. If HAD5 equals 1, the branch address location specified by MOVE is the Program Memory Address which contains the user specified instruction or sub-routine which executes the command. A JUMP or CALL FIFO control instruction performs a jump or call to the location specified by MOVE. If HAD5 equals 0, an RDFIFO instruction can transfer the FIFO contents into the register specified by HAD[4:0].

For further explanation, refer to the diagram below. Beginning at the location specified by MOVE, a user defined program exists which is going to load data into CPU registers 0,1,2, and 3 in four consecutive cycles from the next four FIFO locations. If one of the four FIFO locations contains a command (FICD=1), interrupt level 7 occurs (highest level). Loading a command into a CPU or other data register is not allowed. If this occurs, FIXP (FIFO exception) will be generated.

Following the execution of this routine, the Control Section is ready for its next instruction.

The FIFO drives three internal flags which can also be programmed to interrupt the PAC1000. They are:

- $\overline{FIIR}$ (FIFO full) and FIXP (FIFO exception), which drive INT7.
- FIOR (FIFO output ready), which drives INT5.

Table 2. Host Interface Functions

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	HAD5	HAD[4:0]	HD[15:0]	Function
0	1	0	0	Register Address	Data	Write data to FIFO
0	1	0	1	X	Command	Write command to FIFO
0	0	1	0	00100	X	Reset FIFO
0	0	1	0	00011	X	Reset status register
0	0	1	0	00010	Data	Read program counter
0	0	1	0	00001	Data	Read status register
0	0	1	0	00000	Data	Read data output register

Host Interface (Con't)

Data I/O Registers

Input and Output Data Registers are used to communicate with the Host Data (HD) bus. CPU Registers may be loaded directly from the Data Input Register (DIR) without passing through the FIFO. Similarly, the PAC1000 may be read via the Data Output Register (DOR).

Program Counter

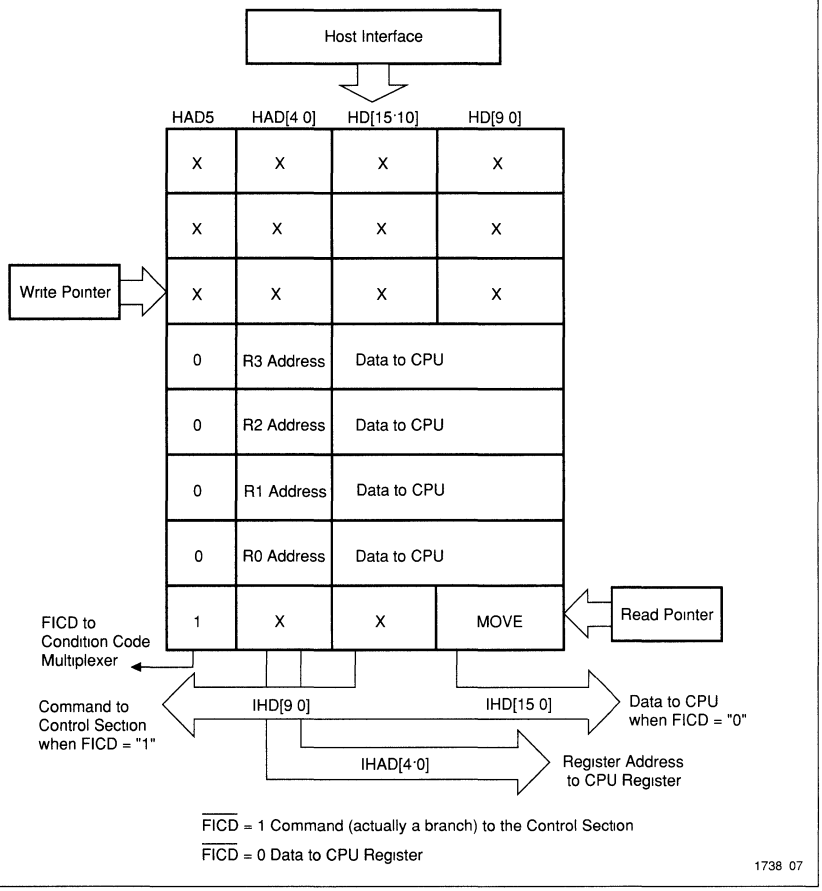
The Program Counter may be read via the Host Data bus. This allows a host to monitor the Program Memory address bus. It can also be used to drive external memory devices for expansion of the Control Port.

Status Register

The Status Register (SR), shown in Figure 8, monitors all internal status. Status bits can be set only by program execution. The SR can be read or cleared as specified in the Host Interface Functions table.

All SR flags are active high (1) and are latched at the rising edge of the clock.

Figure 7. Example of FIFO Block Diagram and Usage



Host Interface (Con't)

STAT11—(DBB) *Security Bit*, set when security is active:

1= Security active.

0= No security.

STAT10—*WSI Reserved.*

STAT9—(FIXP) *FIFO Exception*, set when the CPU receives a command or Control Section receives data:

1= Command or data received.

0= No exception occurred.

STAT8—(FIIR) *FIFO-Input Ready*, set when there is at least one vacant location in the FIFO:

1= FIFO ready for input.

0= FIFO not ready for input.

STAT7—(CY) *Carry Flag*, set when a carry (addition) or borrow (subtraction) occurs in CPU operations:

1= Carry occurred.

0= No carry occurred.

STAT6—(Z) *Zero Flag*, set when the result of a CPU operation is zero:

1= Zero occurred.

0= No zero occurred.

STAT5—(O) *Overflow Flag*, set when an overflow occurs during a two's complement operation:

1= Overflow occurred.

0= No overflow occurred.

STAT4—(S) *Sign Bit*, set when the most significant bit of the result of the previous CPU operation is negative:

1= Result is negative.

0= Result is positive.

STAT3—(STKF) *Stack Flag*, set when the stack is full:

1= Stack is full.

0= Stack is not full.

STAT2—(BRKPNT) *Breakpoint Flag*, set when the address in the breakpoint register is equal to the EPROM address:

1= Breakpoint occurred.

0= No breakpoint occurred.

STAT1—(BCZ) *Block Counter Zero*, set when the counter decrements to all 0s:

1= Block Counter reached zero.

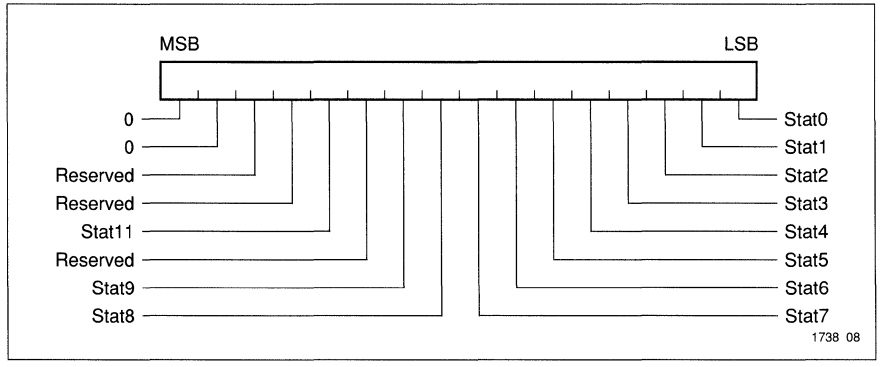
0= Block Counter is not zero.

STAT0—(ACO) *Address Counter Ones*, set when the counter increments to all 1s:

1= Address Counter reached all ones.

0= Address Counter is not all ones.

Figure 8. Status Register



Control Section

The control section, shown in Figure 9, consists of a number of blocks which are concerned with the sequencing of the control programs in the PAC1000. These are:

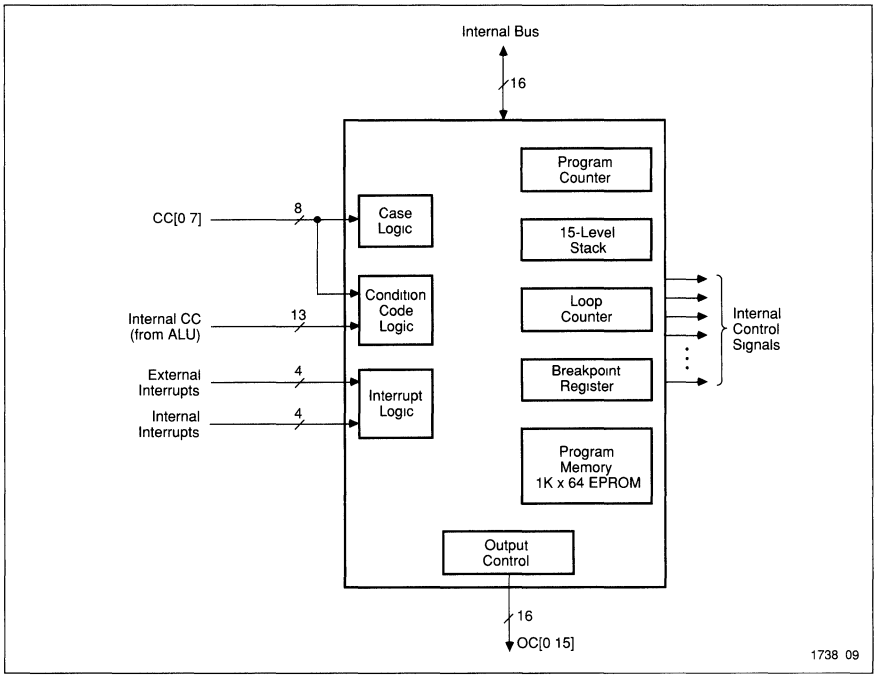
- ❑ Program Memory
- ❑ Security
- ❑ 15-Level Stack
- ❑ Program Counter
- ❑ Loop Counter
- ❑ Breakpoint Register
- ❑ Condition Codes
- ❑ Case Logic
- ❑ Interrupt Logic
- ❑ Output Control

Each block is described in detail below.

Parallel Operations

The PAC1000 can perform three simultaneous operations within a single instruction cycle, as shown in Figure 10. The ability to fetch an instruction from the Program Memory, execute it, and output a result within 50 nsec is due to a highly parallel structure.

Figure 9. Control Architecture



Control Section (Con't)

Program Memory

The Program Memory is a 1Kx64 high-speed EPROM. This on-board-memory allows the PAC1000 to operate in embedded control applications and eliminates the need for external memory components. Using an erasable memory allows program code to be modified for debug and/or field upgrades. The Program Memory is easily programmed using the WSI MagicPro™ (Memory and PSD Programmer).

Only sixteen Program Memory locations are reserved. The rest of the 1024 locations are available for applications.

Program memory is segmented as follows:

Address	Function
000H	Reset pointer program to here
000H–007H	User Defined Initialization Routine
008H–00FH	Interrupt Vector Locations
010H–3FFH	User-Defined Application Programs

Upon receiving a reset, the Program Counter is forced to address 000H. This location may contain a jump or call which branches to an initialization routine. Alternatively, the first eight locations of memory may be used as an initialization/configuration routine.

Security

User programs may be protected by setting a security bit during EPROM programming. Thereafter, the EPROM contents cannot be read externally. When the EPROM is erased, the security bit is cleared.

15-Level Stack

The 15-level Stack stores the return address following subroutine calls, interrupt service routines and the contents of the Loop Counter inside nested loops. When the stack is full, the STKF condition becomes true, and an interrupt (INT7) will occur. The interrupt service routine will overwrite the top of the stack.

Popping from an empty stack produces the previous top of stack value; pushing on a full stack overwrites the top of the stack.

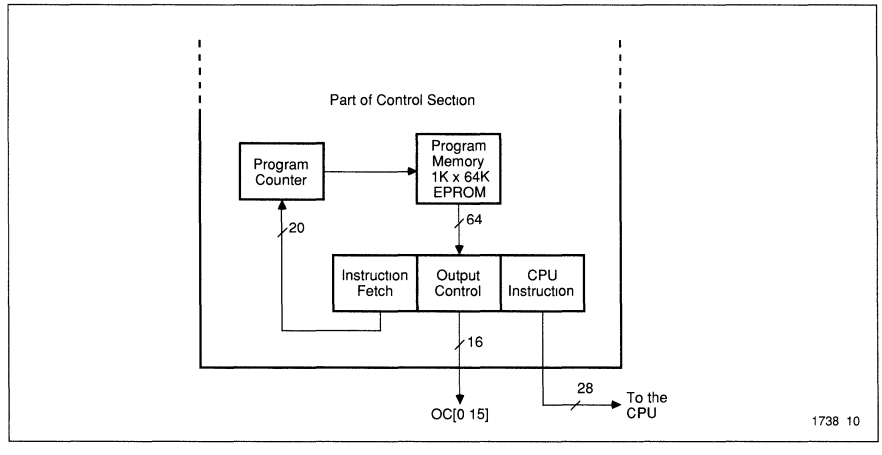
Program Counter

The 10-bit Program Counter (PC) generates sequential addressing to the 1K word Program Memory. Upon reset the PC is loaded with a 000H. From this point the value of the Program Counter is determined by program execution or interrupts.

Any JUMP or Case instruction that is executed loads the Program Counter with the destination address. CALL instructions or interrupts cause PC + 1 to be pushed onto the stack. The RETURN instruction loads the Program Counter from the stack with the value of the return address. This value may have previously been placed on the stack by a CALL or interrupt.

The PC can also be loaded from the Command/Data FIFO causing program execution to commence at an address provided by the host.

Figure 10. Parallel Operations



Control Section (Con't)

Loop Counter

The Loop Counter (LC) has two functions:

- 10-bit down counter that supports the LOOP instruction.
- Branch Register that can be loaded from the CPU Register File or Program Memory and used as an additional source of branching to Program Memory.

The LC can be loaded with values up to 1023. Loop initialization code places a value in LC. Loop termination code tests the counter for a zero value and then decrements LC. The loop count can be a constant, or it can be computed at execution time and loaded into LC from the CPU. The LC register can also be used as a CALL or JUMP execution vector. The content of the LC is automatically saved on (or retrieved from) the Stack when the program enters (or leaves) a nested loop.

A loop count will be loaded into the LC when a FOR instruction is encountered. This count can be a fixed value or it can be calculated and loaded from the CPU. The ENDFOR instruction will test the Loop Counter for a zero value. If this condition is not met, then the LC will be decremented by one. The program loop will continue until the count value equals zero. In a nested loop, the FOR instruction will load a new value to the LC and push the previous value to the stack.

Debug Capabilities

The PAC1000 provides breakpoint and single step capabilities for debugging application programs.

Breakpoint Register

The Breakpoint Register (BR) is a 10-bit register used for real time debug of the PAC1000 application program.

The Breakpoint Register can be loaded from one of two sources, either a constant value specified in the Program Memory or a calculated value loaded from the CPU. When the Program Memory address matches the contents of the Breakpoint Register an interrupt (INT 6) occurs. A service routine should exist in Program Memory which then performs the required procedure.

Single Step

Single step is a debugging mode in which the currently-executing program is interrupted by interrupt 6 after the execution of every instruction. The interrupt 6 service routine should reside in Program Memory.

Bit 8 in the Mask Register determines whether the PAC1000 is in a breakpoint mode (mask-bit 8 equals 0) or in a single step mode (mask-bit 8 equals 1).

Both breakpoint and single step use interrupt 6. The interrupt 6 service routine will typically dump the contents of the PAC1000 internal registers into external SRAM devices for examination by the user.

Condition Codes

The Condition Code (CC) logic operates on 21 individual program test conditions. Each condition can be tested for true or not true. The PAC1000 can also test up to four conditions simultaneously. For this feature refer to the section titled *Case Logic*.

Control Section (Con't)

User-Specified Conditions

User-Specified Conditions are treated in the same manner as internally generated test conditions. CC0—CC7 should be connected directly to the corresponding PAC1000 input pins. These signals must satisfy the required setup time to be serviced in the next cycle.

CPU Flags

CPU flags are internally generated. They reflect the status of the previous CPU arithmetic operation. These signals are internally latched and are valid only for one instruction (the instruction following their generation). The flags for arithmetic operations are defined as follows:

Zero (Z)—The result of the previous CPU operation is zero (Z=1).

Carry (CY)—The result of the previous CPU operation generated a carry (addition) or borrow (subtraction) (CY=1).

Overflow (O)—The previous two's complement CPU operation generated an overflow (O=1).

Sign (S)—The most significant bit of the result of the previous CPU operation is negative (S=1).

FIFO Flags

FIFO flags allow the user to synchronize and monitor the operations that are performed on the FIFO by the host or by user's program.

Upon reset the FIFO flags are cleared, signifying an empty state. The meaning of the flags are as follows:

FIFO Output Ready (FIOR)—There is at least one word in the FIFO (FIOR=1).

FIFO Input Ready (FIIR)—FIFO is not full (FIIR=1). This flag can also be connected to the host through I/07.

FIFO Command/Data (FICD)—This flag indicates if the contents of the FIFO is a command or a data. This flag is generated directly from HAD5 (FICD=1 command, FICD=0 data).

FIFO Exception (FIXP)—This flag indicates that one of two events occurred: (a) FIFO data has been read as a command, or (b) a command has been read as data.

Stack-Full Flag

STACK FULL flag (STKF=1) indicates that the stack is 15 levels full. This condition will also generate an interrupt (INT7) if not masked.

Interrupt Flag

INTERRUPT flag (INTR =1) indicates that there is a masked interrupt pending. This flag is cleared when the interrupt is cleared.

Data Register Read Flag

DATA REGISTER READ flag (DOR) is a handshake flag between the host and the PAC1000, accessible only to the PAC1000. The flag is reset (DOR=0) when the PAC1000 writes into the Data Output Register. The flag is set (DOR=1) after the host has performed a read on the Data Output Register.

Counter Flag

Counter flags reflect the status of their respective counters. The PAC1000 utilizes two counters; the Address (A) counter is a 16/22-bit auto-incrementing up counter; the

Table 3. Condition-Code Logic

Test Group	Source	Conditions and Flags
User-Specified	External	CC0–CC7
CPU	Internal	Carry (CY), Zero (Z), Overflow (O), Sign (S)
FIFO	Internal	FIFO Command/Data (FICD), FIFO Output Ready (FIOR), FIFO Input Ready (FIIR), FIFO Exception (FIXP)
Counters	Internal	Address Counter Ones (ACO), Block Counter Zero (BCZ)
Stack	Internal	Stack Full (STKF)
Interrupt	External/Internal	Interrupt (INTR) is pending
Data register read	Internal	Data Output Register(DOR) has been read

Control Section (Con't)

Block (B) counter is an auto-decrementing 16-bit down counter. The counters' clock input signal is the same as the PAC1000's clock signal. Each counter can be individually enabled or disabled. When disabled, the output retains the last count. The counter flags are defined as follows:

ACO—*A Counter Ones*, set when the A counter has reached the value FFFFH, in the 16-bit mode, or the value 3FFFFFH in the 22-bit mode.

BCZ—*B Counter Zero*, set when the B counter has reached the value 0000H.

Case Logic

THE PAC1000 hardware implements two basic types of Case instructions: Case and Priority Case.

Case Instructions

Case instructions operate on any one of four different Case groups. Each Case group consists of a combination of four test conditions which can be tested in a single cycle. In that same cycle the PAC1000 will branch to one of the addresses contained in the sixteen memory locations following the instruction, depending on the status of the four inputs to the Case group being tested.

There are four Case Groups (sets of Case Conditions):

Case Group 0 (CG0): CC0–CC3.

Case Group 1 (CG1): CC4–CC7.

Case Group 2 (CG2):

- Z—Zero
- O—Overflow
- S—Sign
- CY—Carry

Case Group 3 (CG3):

- INTR—Interrupt
- BCZ—B Counter Zero
- FIOR—FIFO output Ready
- FICD—FIFO Command/Data

(The FIXP, ACO, STKF, FIIR, and DOR condition codes do not fall into any of the four Case groups.)

Priority Case Instructions

Priority Case instructions operate on the four internal and the four external interrupt inputs. In this mode of operation, interrupts are treated as prioritized test conditions and the priority encoder is used to generate a branch to the highest priority condition. The branch address is located in one of the nine memory locations following the Priority Case instruction. Priorities in this mode of operation are the same as in the Interrupt mode of operation. Once a Priority Case instruction is executed, the occurrence of a higher priority condition will not affect program execution until another Priority Case instruction is executed. For a Priority Case instruction to be executed, MODE0 of the Mask Register must be equal to zero (MODE0=0).

Interrupt Logic

The Interrupt Logic accepts eight inputs, four of them are generated externally and four are dedicated for internal conditions. The four external, user defined, inputs (INT0–INT3) are connected to pins INT0, INT1, INT2, and INT3. These are positive, rising-edge-triggered signals that have a maximum latency of two cycles. Each interrupt has a reserved area in memory that should contain a branch to an interrupt service routine.

Table 4. Interrupt Assignments

Interrupt	Priority	Effect	Trigger Condition	Reserved Address
INT7	Highest	Internal	FIXP+ACO+STKF+$\overline{\text{FIIR}}$	00FH
INT6		Internal	BRKPT	00EH
INT5		Internal	FIOR	00DH
INT4		Internal	Software Interrupt (SWI)	00CH
INT3		External	INT3	00BH
INT2		External	INT2	00AH
INT1		External	INT1	009H
INT0	Lowest	External	INT0	008H

Control Section (Con't)

Clearing a serviced interrupt is performed automatically. When the interrupt is serviced, the internally generated vector is decoded to clear the serviced interrupt. In addition, the user can clear any pending interrupt by using the Clear Interrupt Instruction (CLI).

Interrupt Mask Register

The Interrupt Mask Register, shown in Figure 11, allows individual interrupts to be masked. Setting a Mask Register bit to a 1 masks the associated interrupt. To unmask an interrupt, the appropriate Mask Register bit must be reset to 0.

When the PAC1000 is reset,the Mask Register will mask all interrupts and the Mode Register will select the non-interrupt mode. To select the interrupt mode the MODE0 bit (see Configuration Register section in this document) should be set to 1 (MODE0=1).

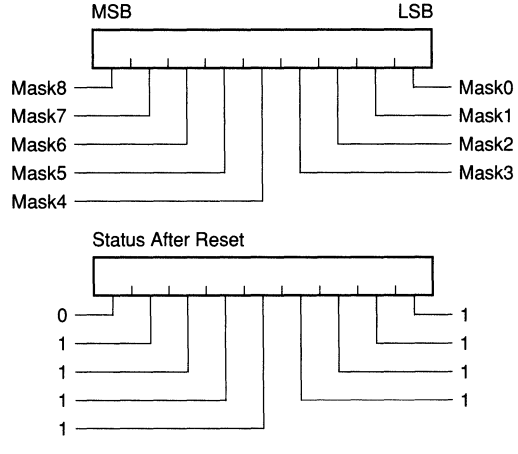
Mask8 is used to select INT6 to be either a single-step interrupt (when Mask8=1) or a breakpoint interrupt (when Mask8=0) .See the section on Debug Capabilities for further details.

Table 5. Interrupt Definitions

Interrupt	Triggered By
INT7[1]	FIFO Exception (FIXP)
	Address Counter contains all Ones (ACO)
	Stack Full (STKF)
	FIFO Full (Not FIFO Input Ready, $\overline{\text{FIIR}}$)
INT6[2]	Breakpoint or Single Step occurrence
INT5	FIFO Output Ready (FIOR)
INT4	Always pending; triggers when unmasked by program execution
INT3	User-defined
INT2	User-defined
INT1	User-defined
INT0	User-defined

Notes:
1. The INT7 interrupt handler checks the source of the interrupt by testing the condition code.
2. See Interrupt Mask Register, Mask8.

Figure 11. Interrupt Mask Register



Control Section (Con't)

Output Control

The Output Control bus (OUTCNTL) consists of 16 latched Output Control signals. These signals can be changed on a clock to clock basis. For every Program Memory location there is a dedicated field which specifies the value of the Output Control bus. The OUTCNTL Operation places this value on the Output Control bus. The OUTCNTL Operation can be performed in parallel with any other PAC1000 instructions.

The OUTCNTL bus can be used to control external events on a clock to clock basis.

Counters

The PAC1000 contains a 16 or 22-bit Address Counter and a 16-bit Block Counter. Each of these counters can change count on a clock to clock basis or can be internally or externally enabled or disabled on a clock to clock basis. These counters are in addition to the Loop and Program Counters of the Control Section.

Address Counter

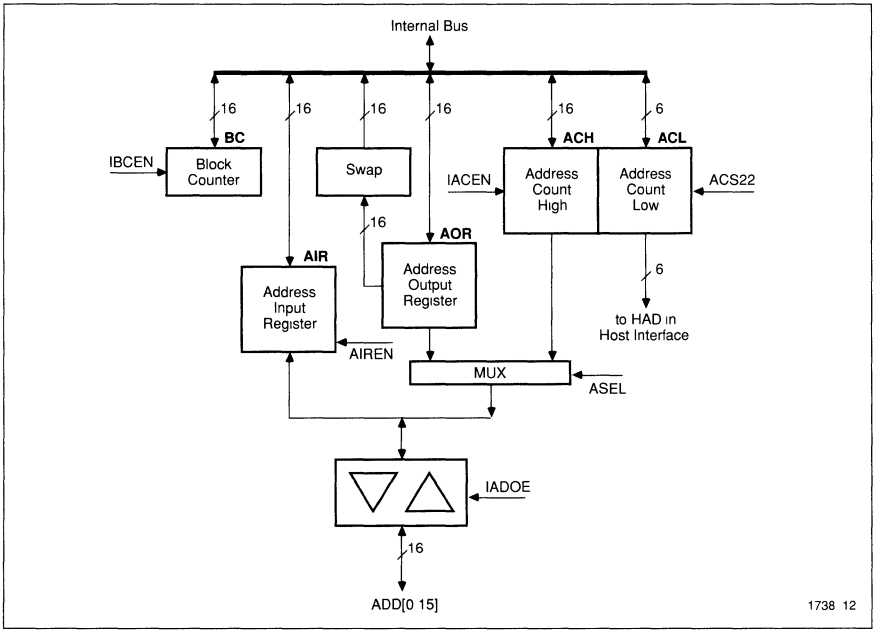
The Address Counter (AC), shown in Figure 12, is a 16- or 22-bit ascending counter that can be loaded or read by the CPU and enabled/disabled with the ACEN bit of the Control Register. (This control is also available externally through the I/01 pin; see I/O and Special Functions). While enabled, the counter will increment by one every rising edge of the clock.

The ACO flag indicates that the value of the counter is all ones. This flag stays latched until the counter is loaded with a new value. The counter will continue to count until disabled. ACO is a condition code and a member of a Case Group; see the Control Section description for more details. ACO can also generate an internal interrupt 7, if enabled.

In the 16-bit mode, the counter outputs (ACH) are available through the ADD bus. The count is gated to the ADD bus by setting the ASEL bit (CTRL9) of the Control Register.

In the 22-bit mode, the higher 16 bits (ACH) are available through the ADD bus and the six low order bits (ACL) are available through the Host Address (HAD) bus. These low order bits are multiplexed with the host address lines. The address lines from the host which drives the HAD bus must be placed in the high impedance state before the lower 6-bits (ACL) of the Address Counter can be read.

Figure 12. Address and Block Counter



Counters (Con't)

Selecting the 16- or 22-bit count mode is performed by setting or resetting the ACS22 bit in the I/O Configuration Register.

The address Output Register is an alternate source of address outputs; it is selected by resetting the ASEL bit of the Control Register. In this mode the CPU can be used to provide address generation and the Address Counter can be used as an event counter.

Block Counter

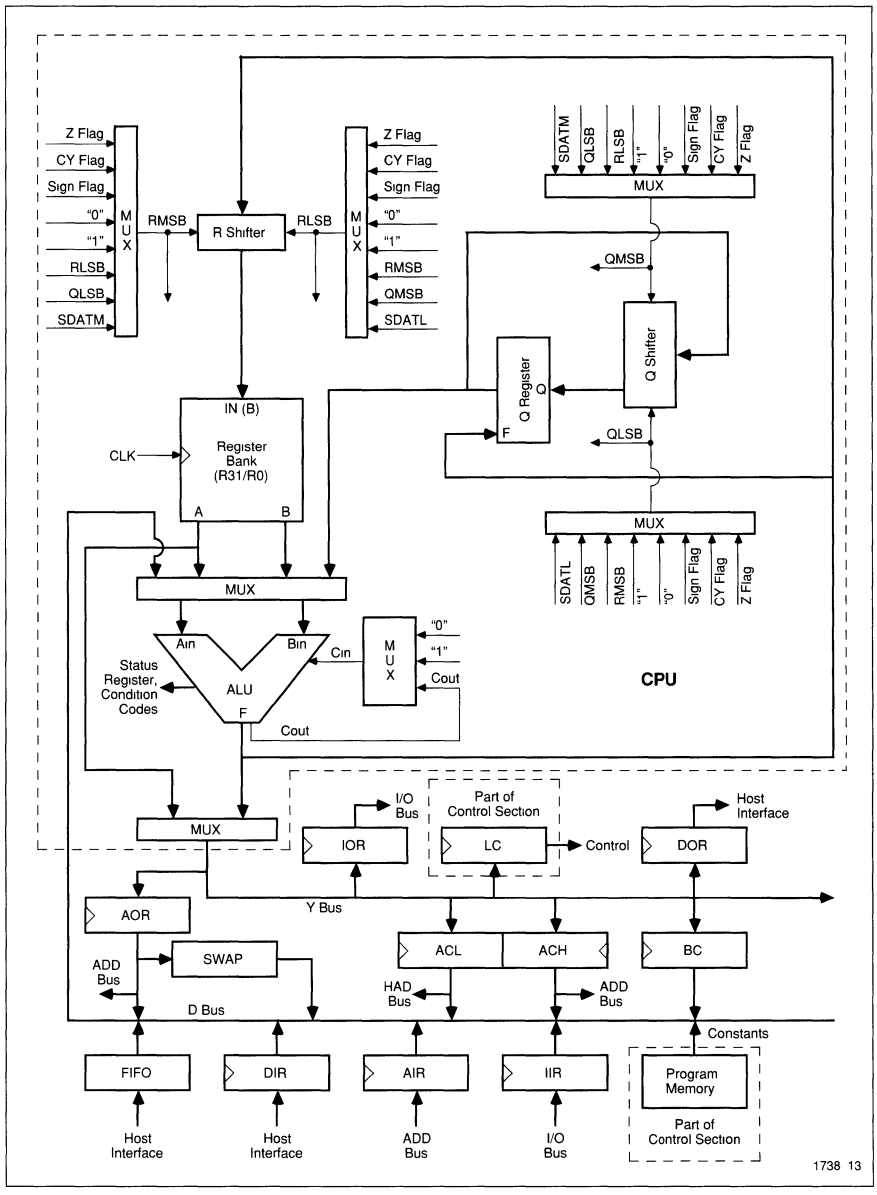
The Block Counter (BC) is a 16-bit down counter. It is enabled by the BCEN bit of the Control Register. It is useful as a counter for DMA transfers. The BCEN signal is (optionally) available externally through the I/O0 bit (see I/O and Special Functions). While enabled, the counter will decrement by one every rising edge of the clock. The BCZ flag indicates that the counter reached the zero value. After the occurrence of an all 0s condition the Block Counter will continue down counting until disabled. The flag is latched and can be cleared by loading a new value into the Block Counter. BCZ is a condition code and a member of a Case Group; see the Control Section description for more details.

Both counters may be read without disabling the count operation and loaded via the CPU.

Central Processing Unit

The CPU, shown in Figure 13, performs 16-bit operations in a single clock cycle. It contains 33 general purpose registers (R0...R31, and Q). The Q register can be used in conjunction with any of the R0...R31 registers to perform double precision shift operations. The main building blocks are the register bank (R0...R31), Q register, ALU, Y-bus devices, and D-bus devices. The register bank supplies up to two 16-bit registers, one of which is always the destination register.

Figure 13. CPU Block Diagram



Central Processing Unit (Con't)

The ALU operates on up to two external operands that are selected by its input MUX. In every instruction, 1 of the 10 D-bus devices (AOR, SWAP, ACL, ACH, BC, FIFO, DIR, AIR, IIR, and Program Store) or a member of the register bank or the Q register outputs, can be selected as an operand source to the ALU. The possibilities are shown in Figure 14. During ALU operations, three options can be selected to provide the carry-in (Cin) input: 0, 1, or the previous latched carry-out (adequate for multiple precision operations).

The ALU's output or a selected register can be loaded into one of the seven Y-bus devices (IOR, AOR, LC, DOR, ACL, ACH, or BC) every instruction cycle. This can happen in parallel with the feedback path from the ALU's output that is directed either to the Q register or to the destination register of the register bank.

Figure 14. CPU Sources and Destinations

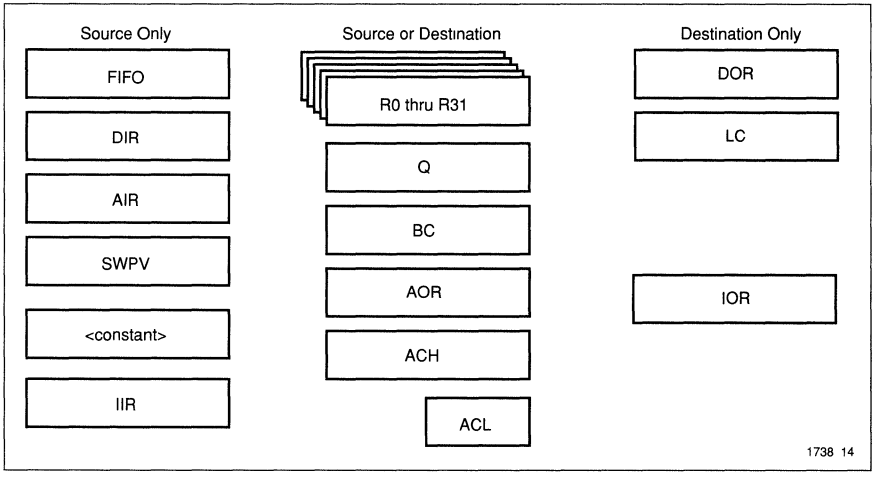


Table 6. CPU Operand Mnemonics

Mnemonic	Description
ACH or ACH/ACL	16- or 22-bit Auto-incrementing Counter, or General Purpose Registers
AIR	Address Input Register
AOR	Address Output Register
BC	Block Counter (16-bit auto-decrementing), or General Purpose Register
<constant>	Constant values in Program Storage
DIR	Data Input Register
DOR	Data Output Register
FIFO	Input Data from FIFO
IIR	I/O Input Register
IOR	I/O Output Register
LC	Program Loop Counter
Q	16-bit CPU Register
R0–R31	16-bit CPU Registers
SWPV	Byte Swap version of AOR

Central Processing Unit (Con't)

CPU operations can be performed on one, two or three operands. Each operation is performed in a single clock cycle. In two- or three-operand instructions, one of the operands must be a CPU internal register (R0...R31, or Q).

CPU operations are performed independently of operations in the counters, Host Interface, Output Control, and Program Control.

Arithmetic Operations

The CPU can perform the following arithmetic operations:

- ❑ Addition
- ❑ Subtraction
- ❑ Increment
- ❑ Decrement
- ❑ Compare

Logic Operations

The CPU can perform the following logic operations:

- ❑ AND
- ❑ OR
- ❑ Invert
- ❑ Exclusive OR
- ❑ Exclusive NOR

Shift Operations

Single shift operations, shown in Figure 15, can occur either to the left or to the right, with or without the Q register. Shift instructions specify the sources that are shifted into the corresponding registers.

All shift operations can be executed in the same clock cycle as an arithmetic or logic operation. The arithmetic or logic operation is executed first; the result is shifted and then stored in the register file. The shift can be either left or right.

The CPU can perform the following shift operations:

- ❑ Single-precision, left or right, within a general-purpose register (R0...R31, or Q).
- ❑ Double-precision, left or right, between an R0...R31 register and the Q register.

The LSB and MSB of the general-purpose registers are each fed by an eight-to-one multiplexer.

The sources and destinations for shift operation are given below:

Shift Right

Zero Flag (Z)

Carry Flag (CY)

Sign Flag (S)

Binary 0 (0)

Binary 1 (1)

Least-significant bit of this register (RLSB)

Least-significant bit of the Q register (QLSB)

Serial I/O port (SDATM)

Shift Left

Zero Flag (Z)

Carry Flag (CY)

Sign Flag (S)

Binary 0 (0)

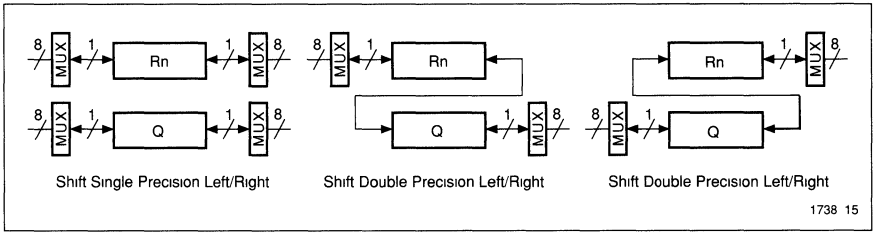
Binary 1 (1)

Most-significant bit of this register (RMSB)

Most-significant bit of the Q register (QMSB)

Serial I/O port (SDATL)

Figure 15. Shift Operations



Central Processing Unit (Con't)

Rotate Operations

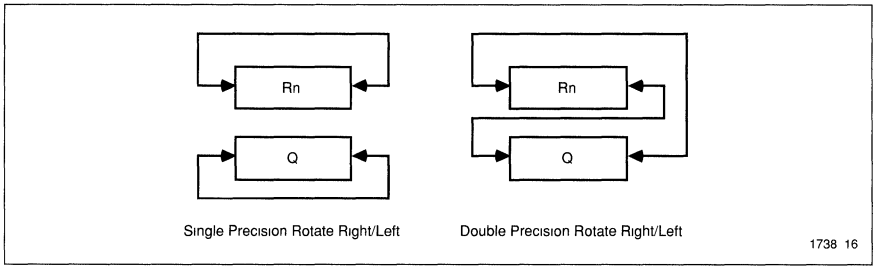
The CPU can perform the following rotate operations, as shown in Figure 16:

- Single-precision, left or right, within a general-purpose register (R0...R31, or Q).
- Double-precision, left or right, between an R0...R31 register and the Q register.

Multiple Precision Operations

The carry-out in each instruction can be used in the next instruction for multiple precision operations (e.g., ADDC). This feature enables the user to implement complex arithmetic operations such as division or multiplication in several clock cycles.

Figure 16. Rotate Operations

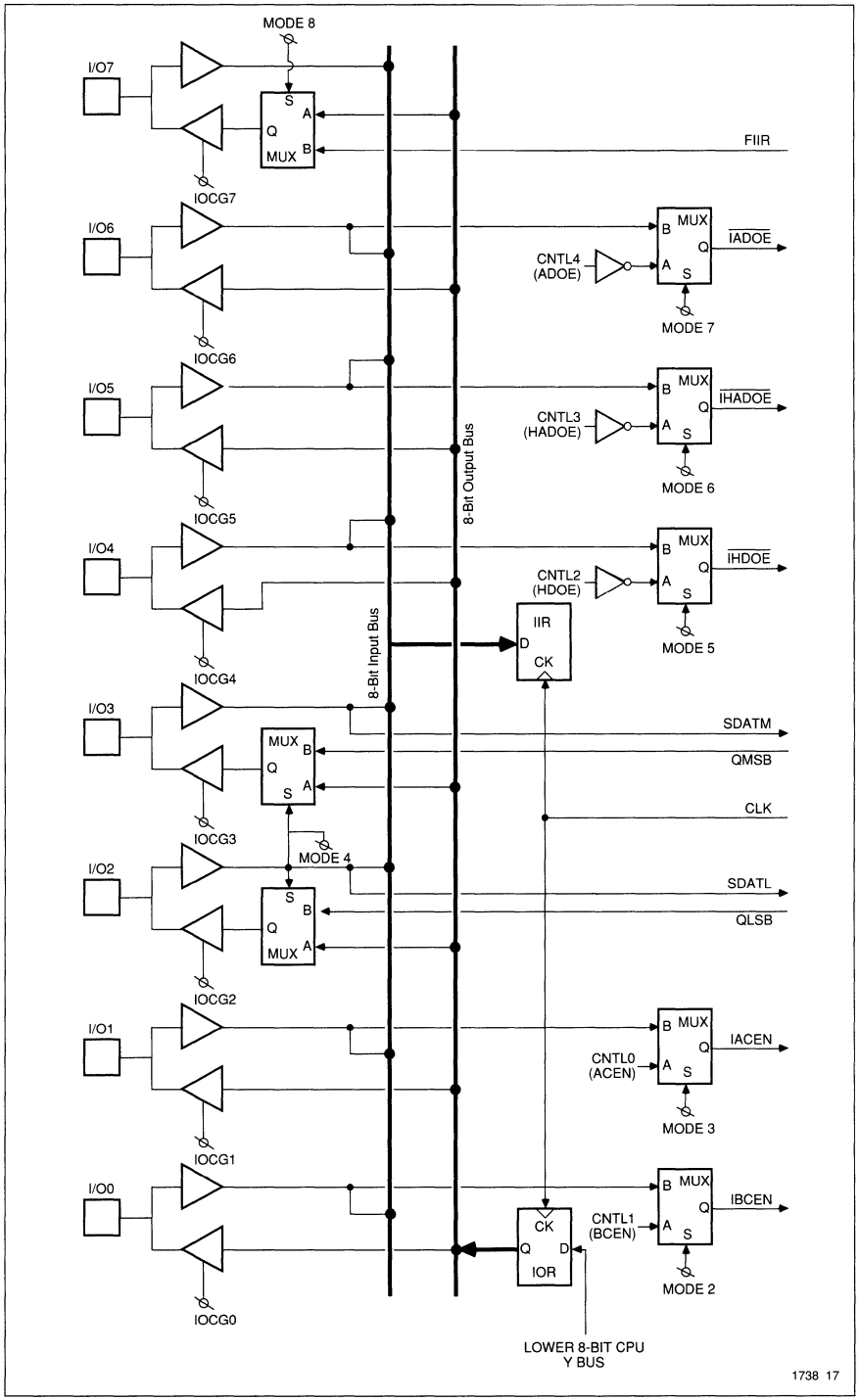


I/O and Special Functions

The I/O bus, shown in Figure 17, consists of eight lines which can be individually programmed as inputs or outputs. These lines can also be programmed to perform Special Functions. The functions of these pins are defined by the Mode Register and I/O Configuration Register (see Configuration Register Section). The I/O and Special Functions map according to the table. The I/O lines must first be configured as inputs or outputs via the I/O Configuration Register; the Special Function option can then be enabled via the Mode Register. Individual special function control is shown in the accompanying table.

Once a Special Function has been enabled, the corresponding internal control function is automatically disabled. Conversely, when a Special Function is disabled, control of the corresponding internal control function is returned to the Control Register (see Configuration Register). Because the Inputs in the I/O Register are clocked on each cycle, the status of the special function can also be read to the CPU.

Figure 17.
I/O and Special Function Bus



Configuration Registers

The Configuration Registers allow the user to control and configure different operating modes of the PAC1000. The three 10-bit Configuration Registers are the Control Register, I/O Configuration Register, and Mode Register. Each register has an associated instruction which allows individual register bits to be modified.

Control Register

The Control Register, shown in Figure 18, provides for internal control of key functions within the PAC1000 . Several of these functions can alternatively be controlled externally through the I/O bus (see I/O and Special Functions). The Control Regıster is modified on the falling edge of the clock.

Table 7. I/O Pins and Special Functions

Pin	Special Function	Direction	Description
I/O7	FIIR	output	FIFO Input Ready. FIFO not full.
I/O6	ADOE	input	Address Output Enable
I/O5	HADOE	input	Host Address Output Enable
I/O4	HDOE	input	Host Data Output Enable
I/O3	QMSB	bidirectional	Q Register MSB
I/O2	QLSB	bidirectional	Q Register LSB
I/O1	ACEN	input	Address Counter Enable
I/O0	BCEN	input	Block Counter Enable

Table 8. Special-Function Control

Special Function	Pin Name	I/O Configuration	Mode
FIIR	I/O7	IOCG7=1 (output)	MODE8=1
ADOE	I/O6	IOCG6=0 (input)	MODE7=1
HADOE	I/O5	IOCG5=0 (input)	MODE6=1
HDOE	I/O4	IOCG4=0 (input)	MODE5=1
QMSB	I/O3	IOCG3=1 (output)	
		IOCG3=0 (input)	MODE4=1
QLSB	I/O2	IOCG2=1 (output)	
		IOCG2=0 (input)	MODE4=1
ACEN	I/O1	IOCG1=0 (input)	MODE3 =1
BCEN	I/O0	IOCG0=0 (input)	MODE2 =1

Configuration Registers (Con't)

ASEL (CTRL9)—*Address Select.* Selects the source that will write to the Address bus:

1= Address Counter.

0= Address Output Register (AOR).

AIREN (CTRL8)—*Address Input Register Enable.* Enables and disables writing to the Address Input Register from the ADD Port:

1= Enable writing to Address Input Register (AIR).

0= Disable writing to Address Input Register (AIR).

DIREN (CTRL7)—*Data Input Register Enable.* Enables and disables writing to the Data Input Register (DIR) from the HD Port:

1= Enable writing to Data Input Register (DIR).

0= Disable writing to Data Input Register (DIR).

HDSEL1 (CTRL6) and HDSEL0 (CTRL5)—*Host Data Select.* Select the source to be connected to Host Data (HD) bus:

HDSEL1 (CTRL6)	*HDSEL0 (CTRL5)*	*Selection*
0	0	FIFO—Peripheral Mode
0	1	Data Output Register
1	0	Status Register
1	1	Program Counter

ADOE (CTRL4)—*Address Output Enable.* Selects direction of Address bus (ADD) for next clock cycle:

1= Output (see ASEL).

0= Input (see AIREN).

HADOE (CTRL3)—*Host Address Output Enable.* Selects direction of Host Address (HAD) bus for next clock cycle:

1= Output (driven from ACL Register).

0= Input (into the FIFO).

HDOE (CTRL2)—*Host Data Output Enable.* Selects Direction of Host Data (HD) bus for next clock cycle:

1= Output (See HDSEL0 and HDSEL1).

0= Input (See DIREN).

BCEN (CTRL1)—*Block Counter Enable.* Enables and disables Block Counter:

1= Enable Counting on next rising clock edge.

0= Disable Counting on next rising edge.

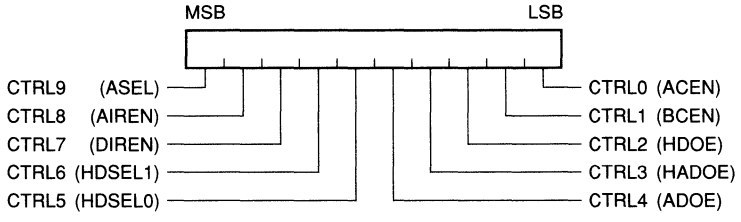
ACEN (CTRL0)—*Address Counter Enable.* Enables and disables Address Counter:

1= Enable Counting on next rising clock edge.

0= Disable Counting on next rising clock edge.

2

Figure 18. Control Register



Note: After Reset, All Bits Are Cleared to Zero.

1738 18

Configuration Registers (Con't)

I/O Configuration Register

The I/O Configuration Register, shown in Figure 19, controls the direction of the individual lines of the I/O bus as well as configuring the Address Counter. Each I/O pin can be configured independently to be a general purpose input or output, or each can serve a special function (see I/O and Special Function). The I/O Configuration Register is also used to configure the Address Counter as a 16-bit counter with a maximum count of FFFFH or as a 22-bit counter with a maximum count of 3FFFFFH. The I/O Configuration Register is modified on the falling edge of the clock.

ACS22 (IOCG9)—Configures Address Counter as a 22- or 16-bit counter:

1= 22-bit counter.

0= 16-bit counter.

I/O7 (IOCG7)—Selects direction of I/O7 pin:

1= Output.

0= Input.

I/O6 (IOCG6)—Selects direction of I/O6 pin:

1= Output.

0= Input.

I/O5 (IOCG5)—Selects direction of I/O5 pin:

1= Output.

0= Input.

I/O4 (IOCG4)—Selects direction of I/O4 pin:

1= Output.

0= Input.

I/O3 (IOCG3)—Selects direction of I/O3 pin:

1= Output.

0= Input.

I/O2 (IOCG2)—Selects direction of I/O2 pin:

1= Output.

0= Input.

I/O1 (IOCG1)—Selects direction of I/O1 pin:

1= Output.

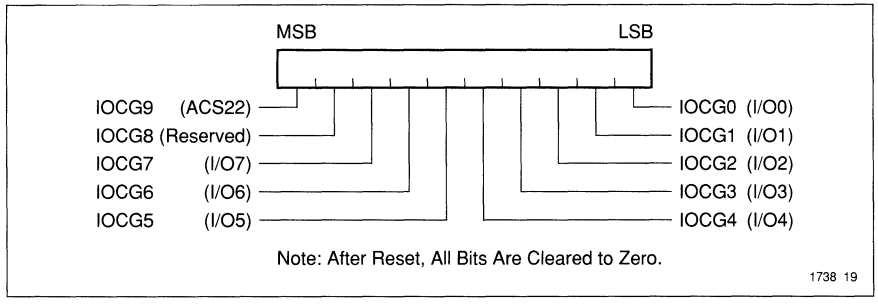
0= Input.

I/O0 (IOCG0)—Selects direction of I/O0 pin:

1= Output.

0= Input.

Figure 19. I/O Configuration Register



Note: After Reset, All Bits Are Cleared to Zero.

1738 19

Configuration Registers (Con't)

Mode Register

The Mode Register, shown in Figure 20, allows the user to externally control and monitor key elements within the PAC1000 which would (alternatively) be controlled internally through the Control Register. Enabling a Special Function in the Mode Register disables the corresponding function in the Control Register. The Special Function input pins are shared with the general purpose I/O pins. The direction of the appropriate pin must be set in the I/O Configuration Register prior to programming the Mode Register.

The Mode Register can also be used to reset the FIFO as well as program the interrupt controller to generate either interrupts or Priority Test Conditions. See the discussion on "Priority Case" in the *Condition Code* section, above.

After Reset, all Mode Register bits equal zero. The Mode Register is modified on the falling edge of the clock.

The use of the Mode Register and I/O Configuration register for Special Functions is shown in the Special Function Settings table.

FIRST (MODE9)—*FIFO Reset.* (If held high, FIFO cannot receive information):

1= Initiate FIFO Reset (FIRST).

0= Complete FIFO Reset (FINRST).

FIIR (MODE8)—*FIFO Input Ready:*

1= I/O7 becomes output for the FIFO Input Ready (FIIR) flag.

0= I/O7 becomes general purpose I/O (IO7).

ADOE (MODE7)—*Address Output Enable:*

1= I/O6 becomes input for the Address Output Enable (AOE).

0= I/O6 becomes general purpose I/O (IO6).

HADOE (MODE6)—*Host Address Output Enable:*

1= I/O5 becomes input for Host Address Output Enable (HADOE).

0= I/O5 becomes general purpose I/O (IO6).

HDOE (MODE5)—*Host Data Output Enable:*

1= I/O4 becomes input for Host Data bus Output Enable HDOE).

0= I/O4 becomes general purpose I/O (IO4).

SIOEN (MODE4)—*Serial I/O Enable:*

1= I/O3 and I/O2 become MSB and LSB (respectively) of the CPU's Q register (SIO).

0= I/O3 and I/O2 become general purpose I/O ACEN(MODE3).

ACEN (MODE3)—*Address Counter Enable:*

1= I/O1 becomes input for Address Counter Enable (ACEN).

0= I/O1 becomes general purpose I/O.

BCEN (MODE2)—*Block Counter Enable:*

1= I/O0 becomes input for Block Counter Enable (BCEN).

0= I/O0 becomes general purpose I/O.

Reserved (MODE1)

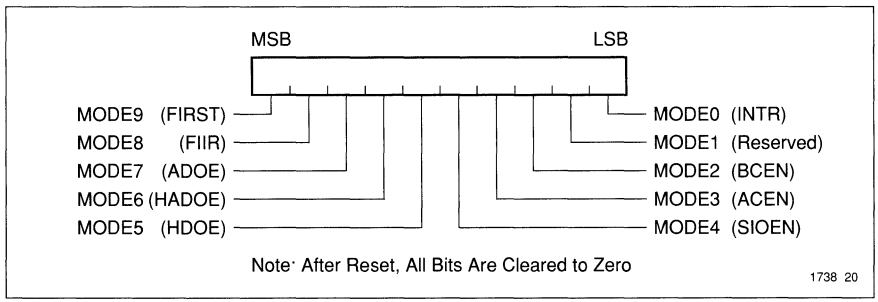
INTR (MODE0)—*Interrupt/Priority-Case Mode:*

1= Select Interrupt mode (INTR).

0= Selects Priority Case mode (PCC).

2

Figure 20. Mode Register



State Following Reset

Whenever the PAC1000 $\overline{\text{RESET}}$ input is driven low for at least two processor clocks, the chip goes through reset. The next two tables describe the PAC1000 signal and internal register states following reset.

Table 9. Special Function Settings

Mode Bit	I/O Configuration Bit	Function
MODE8=1	IOCG7=1	FIIR flag output on I/O7
MODE7=1	IOCG6=0	$\overline{\text{ADOE}}$ provided by I/O6
MODE6=1	IOCG5=0	$\overline{\text{HADOE}}$ provided by I/O5
MODE5=1	IOCG4=0	$\overline{\text{HDOE}}$ provided by I/O4
MODE4=1	IOCG3=1	MSB of Q register output on I/O3
MODE4=1	IOCG3=0	I/O3 can be shifted into MSB of Q register or destination register
MODE4=1	IOCG2=1	LSB of Q register output on I/O2
MODE4=1	IOCG2=0	I/O2 can be shifted into LSB of Q register or destination register
MODE3=1	IOCG1=0	ACEN provided by I/O1
MODE2=1	IOCG0=0	BCEN provided by I/O0

Table 10. Signal States Following Reset

Signal	Condition
HAD[5:0]	Input
HD[15:0]	Input
IO[7:0]	Input
ADD[15:0]	Input
OC[15:0]	0000H

Table 11. Internal States Following Reset

Component	Contents
ACH Register	0
ACL Register	0
AOR Register	0
AIR Register	0
DOR Register	0
DIR Register	0
IOR Register	0
IIR Register	0
STATUS Register	0
I/O Configuration Register	0
CONTROL Register	0
Breakpoint Register	0
Mode Register	0
PC Register (Program Counter)	0
MASK Register	011111111B
BC Register	FFFFH
R31–R0 Registers	Unknown
Q Register	Unknown
LC Register	Unknown
FIFO Locations	Unknown
FIFO Flags	Empty

Electrical and Timing Specifications

Table 12. Absolute Maximum Ratings

Storage Temperature	–65°C to +150°C
Voltage to any pin with respect to GND	–0.6V to +7V
V_{PP} with respect to GND	–0.6 V to +14.0V
ESD Protection	>2000V

Stresses above those listed here may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods of time may affect device reliability.

Table 13. Operating Range

Range	Temperature	V_{CC}
Commercial	0°C to +70°C	+5V ± 5%
Industrial	–40°C to +85°C	+5V ± 10%
Military	–55°C to +125°C	+5V ± 10%

Table 14. DC Characteristics

Over operating range with $V_{PP}=V_{CC}$

Parameter	Symbol	Test Conditions	Min	Max	Units
Output Low Voltage	V_{OL}	I_{OL}=8 mA		0.4	V
Output High Voltage	V_{OH}	I_{OH}=–4 mA	2.4		V
V_{CC} Standby Current CMOS	I_{SB1}	note 1		65	mA
V_{CC} Standby Current TTL	I_{SB2}	note 2		65	mA
Active Current (CMOS)	I_{CC1}	notes 1, 3			
—Commercial				130	mA
—Military				150	mA
Active Current (TTL)	I_{CC2}	notes 2, 3			
—Commercial				160	mA
—Military				180	mA
V_{PP} Supply Current	I_{PP}	$V_{PP}=V_{CC}$		100	µA
V_{PP} Read Voltage	V_{PP}	notes 1, 2	V_{CC}–0.4	V_{CC}	V
Input Load Current	I_{LI}	V_{IN}=5.5V or GND	–10	10	µA
Output Leakage Current	I_{LO}	V_{OUT}=5.5V or GND	–10	10	µA

Notes:
1. CMOS inputs: GND ± 0.3V or V_{CC} ± 0.3V.
2. TTL inputs: $V_{IL} \leq 0.8V$, $V_{IH} \geq 2.0V$.
3. Active current is an AC test and uses AC timing levels.

Table 15. AC Timing Levels

Inputs:	0 to 3V Reference 1.5V
Outputs:	0.4 to 2.4V

Table 16. AC Characteristics

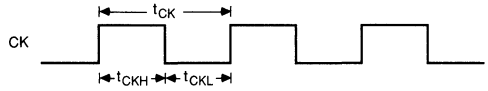
Parameter	Symbol	12MHz[1] Min	12MHz[1] Max	16MHz[2] Min	16MHz[2] Max
CLOCK CYCLE					
Clock Time	t_{CK}	84		62.5	
Clock Pulse Width High	t_{CKH}	29		25	
Clock Pulse Width Low	t_{CKL}	29		25	
HOST READ CYCLE					
Read Cycle Time	t_{RC}	45		35	
Address to Data Valid	t_{ACC}		40		30
$\overline{CS}$ to Data Valid	t_{CS}		40		30
$\overline{CS}$ to Tristate	t_{CSZ}	0	45	0	35
HOST WRITE CYCLE					
Pulse width to $\overline{CS}$ and $\overline{WR}$ LOW	t_{PWL}	23		18	
Pulse width to $\overline{CS}$ and $\overline{WR}$ High	t_{PWH}	12		10	
Data setup to $\overline{WR}$	t_{SD}	5		5	
Data hold to $\overline{WR}$	t_{HD}	12		10	
RESET CYCLE					
$\overline{RESET}$ setup	t_{SR}	10		10	
$\overline{RESET}$ to tristate of ADD, HAD, HD, I/O	t_{RZ}	35		35	
$\overline{RESET}$ clocked to OUTCNTL low	t_{ROL}	35		35	
ADDRESS TIMING					
Address/Data setup	t_{SADD}	0		0	
Address/Data hold	t_{HADD}	8		8	
Clocked Counter to Address output	t_{CADD}		30		22
Clocked Address Register to Address	t_{RADD}		40		30
ADOE enable to data valid	t_{ADOE}		40		30
HADOE enable to data valid	t_{HADOE}		40		30
Address output disable	t_{CKZ}	0	45	0	35

Table 16. AC Characteristics (Con't)

Parameter	Symbol	12MHz[1] Min	12MHz[1] Max	16MHz[2] Min	16MHz[2] Max
DATA AND I/O TIMING					
Clock to I/O Output Valid	t_{CKIO}		30		25
Clock to HD Output	t_{CKHD}		35		30
I/O data setup	t_{SIO}	5		5	
I/O data hold	t_{HIO}	5		5	
HD data setup	t_{SHD}	5		5	
HD data hold	t_{HHD}	12		9	
HDOE enable to data valid	t_{HDOE}		40		30
Bus Output Disable	t_{CKZ}	0	45	0	35
TEST AND INTERRUPT TIMING					
Condition Code setup	t_{SCC}	60		50	
Condition code hold	t_{HCC}	0		0	
Clock to OUTCNTL Valid	t_{COV}	0	25	5	20
Minimum Interrupt pulse for acceptance	t_{IPWA}	15		10	
SPECIAL FUNCTION TIMING (I/O Bus)					
SQ15 Setup	t_{SSQ15}	15		12	
SQ15 hold	t_{HSQ15}	5		5	
SQ0 setup	t_{SSQ0}	15		12	
SQ0 hold	t_{HSQ0}	5		5	
Clock to Q0 output	t_{CKQ0}		35		30
Clock to Q15 output	t_{CKQ15}		35		30
Address counter enable setup	t_{SACEN}	15		10	
Address Counter enable hold	t_{HACEN}	0		0	
Block Counter enable setup	t_{SBCEN}	15		10	
Block Counter enable hold	t_{HBCEN}	5		5	
External output enable to data valid	t_{SFV}		30		25
External output enable to high impedance	t_{SFZ}		30		25

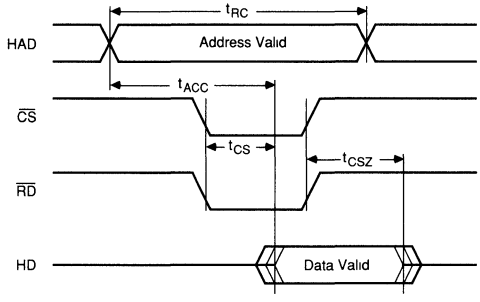
Notes:
1. Operating temperature range: Commercial, Industrial, Military
2. Operating temperature range: Commercial

Figure 21. Clock Cycle Timing



1738 21

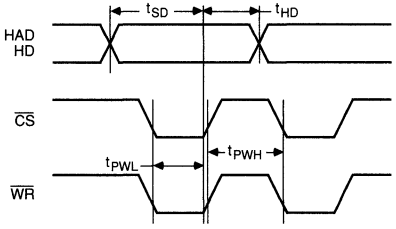
Figure 22. Host Read Cycle Timing



Note t_{CS} is referenced from $\overline{RD}$=0 and $\overline{CS}$=0

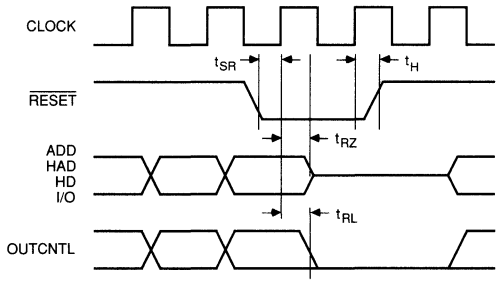
1738 22

Figure 23. Host Write FIFO Cycle Timing



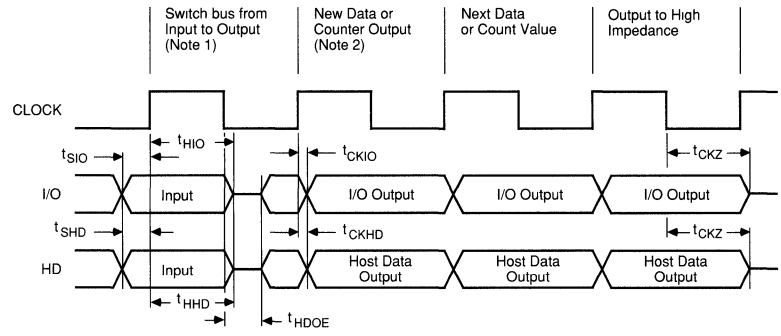
1738 23

Figure 24. Reset Cycle Timing



1738 24

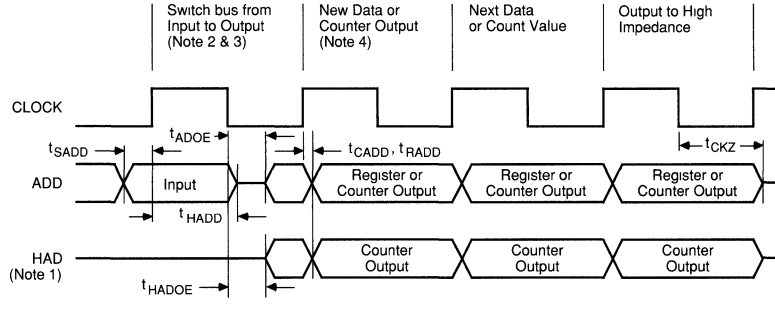
Figure 25. Data and I/O Timing



Notes 1 A bus directional change (input-to-output or output-to-input) takes place on the falling edge of the clock
2 New data or count value is latched on the rising edge of the clock

1738 25

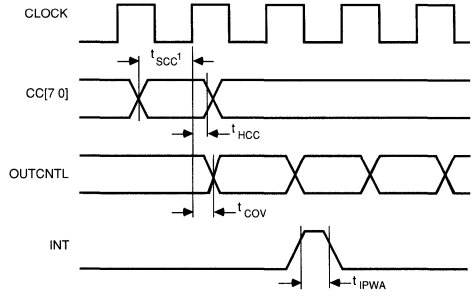
Figure 26. Address Timing



Notes 1 The Host Address (HAD) bus is used to output the lower six bits of the 22-bit counter
2 A bus directional change takes place on the falling edge of the clock (input-to-output or output-to-input)
3 Selection of the source to be output on a bus occurs on the falling edge of the clock (i e , counter or address register)
4 New data or count value is latched on the rising edge of the clock

1738 26

Figure 27. Test and Interrupt Timing



Note 1 Since condition codes are not latched, they should be stable t_{SCC} prior to being tested

1738 27

Figure 28. Special Function Timing

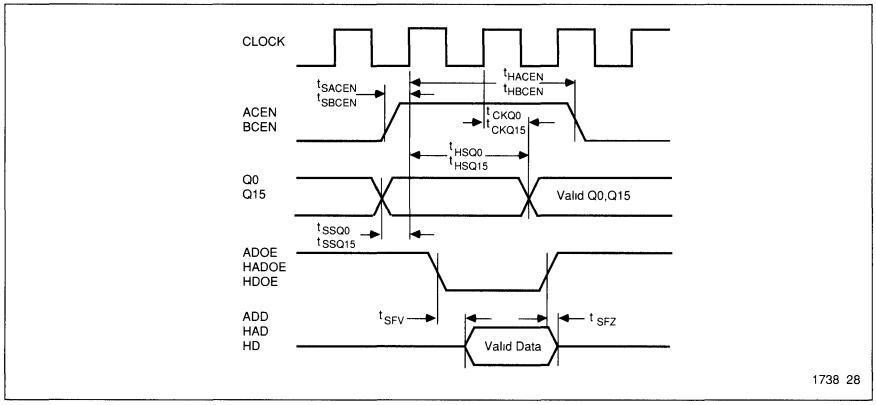


Figure 29. 100-Pin PQFP Pin Assignments

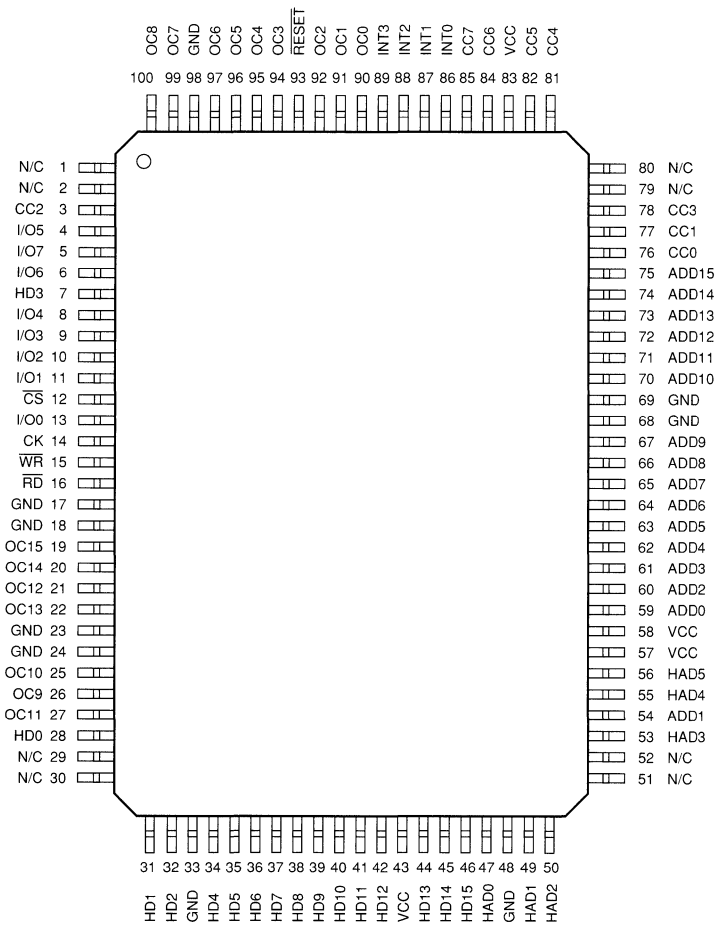
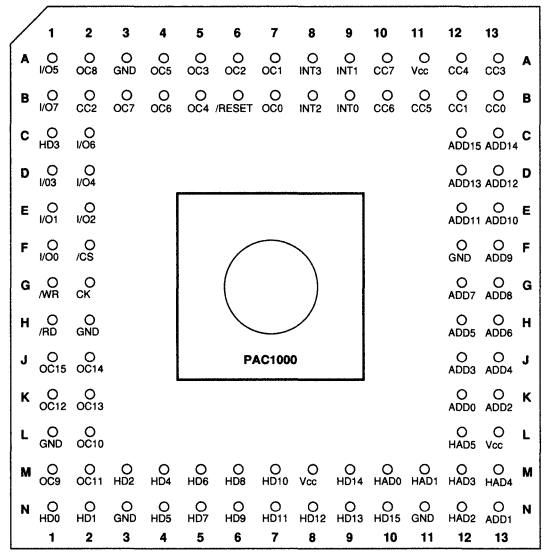
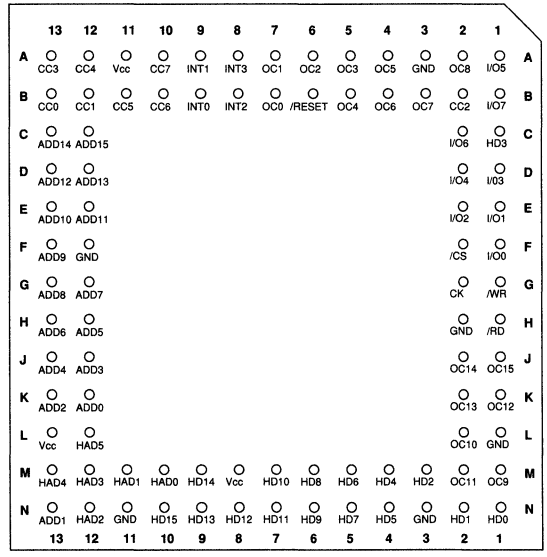


Figure 30. 88-Pin Ceramic PGA Pin Assignments



TOP (THROUGH PACKAGE) VIEW



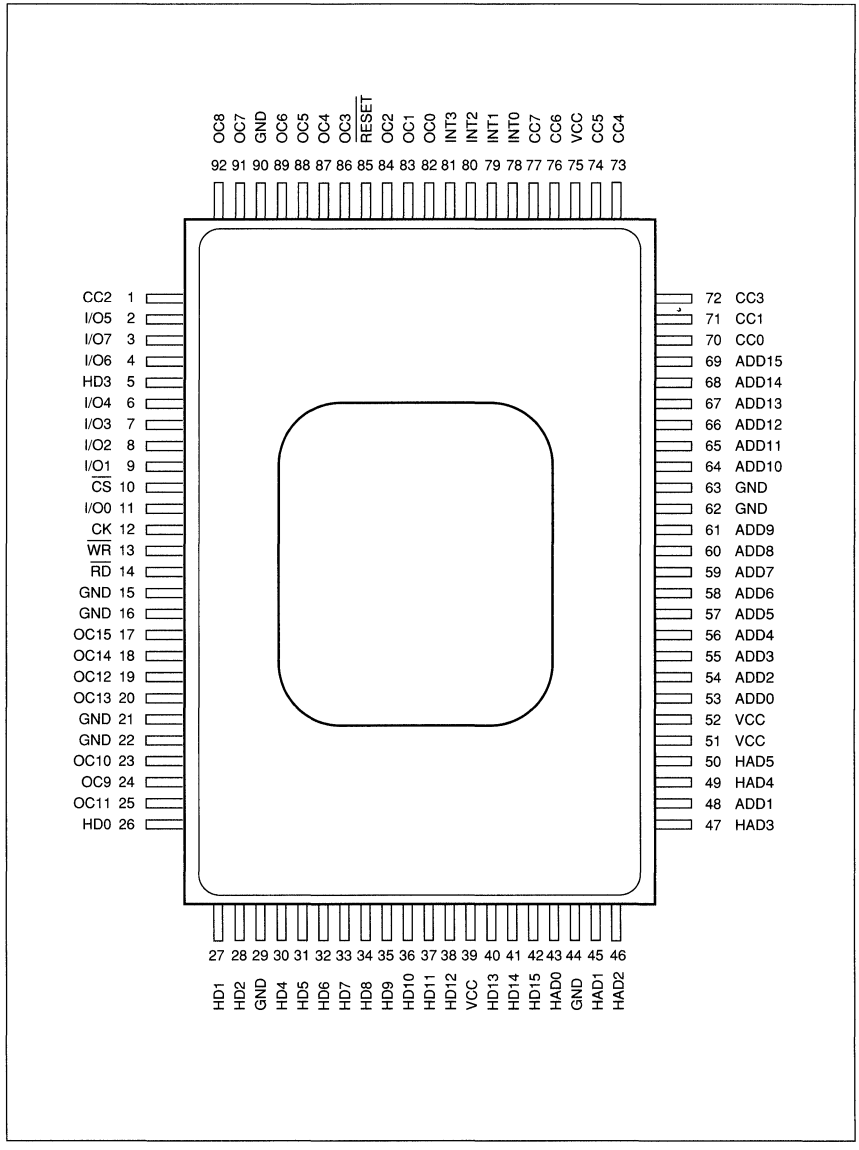
BOTTOM VIEW

1738 29

Table 17. PGA Pin Assignments

Name	Pin	Name	Pin	Name	Pin
$\overline{CS}$	F2	GND	H2	I/O0	F1
$\overline{RD}$	H1	GND	L1	I/O1	E1
$\overline{RESET}$	B6	GND	A3	I/O2	E2
$\overline{WR}$	G1	GND	F12	I/O3	D1
ADD0	K12	GND	N3	I/O4	D2
ADD1	N13	GND	N11	I/O5	A1
ADD10	E13	HAD0	M10	I/O6	C2
ADD11	E12	HAD1	M11	I/O7	B1
ADD12	D13	HAD2	N12	INT0	B9
ADD13	D12	HAD3	M12	INT1	A9
ADD14	C13	HAD4	M13	INT2	B8
ADD15	C12	HAD5	L12	INT3	A8
ADD2	K13	HD0	N1	OC0	B7
ADD3	J12	HD1	N2	OC1	A7
ADD4	J13	HD10	M7	OC10	L2
ADD5	H12	HD11	N7	OC11	M2
ADD6	H13	HD12	N8	OC12	K1
ADD7	G12	HD13	N9	OC13	K2
ADD8	G13	HD14	M9	OC14	J2
ADD9	F13	HD15	N10	OC15	J1
CC0	B13	HD2	M3	OC2	A6
CC1	B12	HD3	C1	OC3	A5
CC2	B2	HD4	M4	OC4	B5
CC3	A13	HD5	N4	OC5	A4
CC4	A12	HD6	M5	OC6	B4
CC5	B11	HD7	N5	OC7	B3
CC6	B10	HD8	M6	OC8	A2
CC7	A10	HD9	N6	OC9	M1
CK	G2			VCC	A11
				VCC	L13
				VCC	M8

Figure 31. 92-Pin CQFP Pin Assignments



Instruction Set Overview

The PAC1000 architecture can perform three operations simultaneously in each instruction cycle. The operations are specified in the System Entry Language (PACSEL) using a single statement. PACSEL instructions can perform three operations:

- Program Control (PROGCNTL)
- CPU
- Output Control (OUTCNTL)

Each *instruction* is executed in a single cycle; the three *operations* are executed in parallel.

The syntax of a PACSEL statement has a label and three components:

```
[label:] PROGCNTL, CPU,
OUTCNTL;
```

The PROGCNTL component determines program flow and determines the next statement to be executed; the CPU component determines which operation is to be performed by the CPU; the OUTCNTL component determines the state of the control outputs.

A comma (,) is used to separate the instructions and a semi-colon marks the end of a statement. In general, each statement is executed in a single cycle.

In PACSEL statements, the PROGCNTL, CPU, OUTCNTL components can come in any order or any combination of Macro or Assembler operators. That is, you may mix Assembler operators among Macro operators. Tables at the end of this section summarize the Macro and Assembler operators.

In some cases, the same mnemonic is used to specify identical operations in both Macro and Assembler level.

You may:

- Specify all the components in the same statement in order to perform the operations in parallel:

  ```
  PROGCNTL, CPU, OUTCNTL;
  ```

- Specify components one at a time:

  ```
  CPU;
  PROGCNTL;
  OUTCNTL;
  ```

- Use components in any combination:

  ```
  PROGCNTL, CPU;
  PROGCNTL, OUTCNTL;
  CPU, OUTCNTL;
  ```

WSI recommends that the user adhere to a specific ordering of these components and specific groupings of assembler-level and macro operators, e.g. in separate files. This manual uses the PROGCNTL, CPU, OUTCNTL ordering.

When PROGCNTL is omitted, the implied instruction is CONTinue, that is, proceed to the next control instruction. When CPU is omitted, the implied instruction is NOP. When OUTCNTL is omitted, the implied instruction is MAINTain, that is, maintain the most recent OUTCTL, in Assembler order.

There is a class of supplemental CPU (sCPU) instructions which can follow certain primary CPU instructions with one or more spaces as a delimiter:

```
PROGCNTL,  CPU sCPU,  OUTCNTL;
```

An sCPU instruction must follow a valid CPU instruction and can not stand alone.

Table 18. PACSEL Assembler Language Summary

Mnemonic	Arguments	Meaning
The PROGCNTL Operators		
ACSIZE	<16/22>	Set A Counter Size
AI		Allow Interrupts
CALL	<LABEL | LCPTR | FIFO>	Uncond Branch Subrtn
CALLC	<COND> <LABEL | FIFO>	Cond Branch Subrtn
CALLNC	<COND> <LABEL | FIFO>	Inv Cond Bran Subrtn
CCASE	<CG> <VALUE>	Branch Subrtn Caseblk
CLI	<MASK>	Clear Interrupt
CONT		Continue
CPI	<VALUE>	Prioritized Subrtn
DI	<MASK>	Disable Interrupt
DSS		Disable SSM
EI	<MASK>	Enable Interrupt
ESS		Enable SSM
JCASE	<CG> <VALUE>	Uncond Branch Caseblk
JMP	<LABEL | LCPTR | FIFO | TOS>	Uncond Branch
JMPC	<COND> <LABEL>	Cond Branch
JMPNC	<COND> <LABEL>	Inv Cond Branch
JPI	<VALUE>	Prioritized Branch
LDBP	<VALUE | LABEL>	Load BP Reg
LDBPD		Load BP Comp Value
LDLC	<VALUE | LABEL>	Load Counter
LDLCD		Load Ctr Comp Value
LOOPNZ	<LABEL>	Repeat Branch CNTRNZ
PLDLC	<VALUE | LABEL>	Push VALUE & LDCTR
PLDLCD		Push Comp Val & LDCTR
POP		Pop Stack
POPLC		Pop Stack to Cntr
PUSHLC		Push Cntr
RESTART		Branch to 0
RET	[<LABEL>]	Return
RC	<COND> [<LABEL>]	Conditional Return
RNC	<COND> [<LABEL>]	Inv Cond Return
RSTCON	<MASK>	Reset Control Reg
RSTIO	<MASK>	Reset I/O Config Reg
RESTMODE	<MASK>	Reset Mode Reg
SETCON	<MASK>	Set Control Reg
SETIO	<MASK>	Set I/O Config Reg
SETMODE	<MASK>	Set Mode Reg
TWB	<COND> <LABEL>	Three-way Branching
TWBC	<COND> <LABEL>	Converse Three-way Branching

Table 18. PACSEL Assembler Language Summary (Cont.)

Mnemonic	Arguments	Meaning
The CPU Operators		
ADC	<ARG1> <ARG2> [<ARG3>] [sCPU]	Add with Carry
ADD	<ARG1> <ARG2> [<ARG3>] [sCPU]	Add
AND	<ARG1> <ARG2> [<ARG3>] [sCPU]	Bitwise AND
CLR	<REG>	Clear Register
CMP	<ARG1> [<ARG2>	Compare
DEC	<ARG1> [<ARG2>] [sCPU]	Decrement
INC	<ARG1> [<ARG2>] [sCPU]	Increment
INV	<ARG1> [<ARG2>] [sCPU]	Invert
MOV	<DEST> <SRC> [sCPU]	Move SRC to DEST
NOP		No Operation
OR	<ARG1> <ARG2> [<ARG3>] [sCPU]	Bitwise or
RDFIFO		Read FIFO Data to Reg
SBC	<ARG1> <ARG2> [<ARG3>] [sCPU]	Sub with Carry
SHLRQ	<REG> <RARG> <QARG>	Shift Left Reg & Q
SHLR	<REG> <RARG>	Shift Left Reg
SHRRQ	<REG> <RARG> <QARG>	Shift Right Reg & Q
SHRR	<REG> <RARG>	Shift Right Reg
SUB	<ARG1> <ARG2> [<ARG3>] [sCPU]	Subtract
XOR	<ARG1> <ARG2> [<ARG3>] [sCPU]	Exclusive OR
XNOR	<ARG1> <ARG2> [<ARG3>] [sCPU]	Exclusive NOR
The sCPU Operators		
ARDREG	<ARG1> <ARG2>	Read Reg to 2nd Dest
ASHLR	<SOURCE>	Shift Reg Left
ASHLRQ	<RSOURCE> <QSOURCE>	Shift Q & Reg Left
ASHRR	<SOURCE>	Shift Reg Right
ASHRRQ	<RSOURCE> <QSOURCE>	Shift Q & Reg Right
AWREG	<ARG1>	Write to 2nd Dest
The MACRO Operators		
DIV	<ARG1> <ARG2> <ARG3>	Divide
MUL	<ARG1> <ARG2> <ARG3>	2'S Comp Multiply
The OUTCNTL Operators		
MAINT		Maintain Prev Value
OUT	<VALUE | EXPRESSION	OUTPUT

Table 19. PACSEL Macro Language Summary

The PROGCNTL Operators

```
CALL <label | LCPTR | FIFO> [ON] [NOT] [<cond>]
CASE N, PROGCNTL, CPU, OUTCNTL;
CLEAR <int level> [...<int level>]
CONFIGURE <pm1> [<pm2>...<pm10>]
CONT
DISABLE <int level> [<int level>...<int level>]
ELSE
ENABLE <int level> [<int level>...<int level>]
ENDFOR
ENDIF
ENDPSWITCH
ENDSWITCH
ENDWHILE
FOR <value>
GOTO <label | LCPTR |FIFO | TOS> [ON] [NOT] [<cond>]
IF [NOT] <cond>
INPUT <i/o pin> [<i/o pin>...<i/o pin>]
LOADBP <value>
OUTPUT <i/o pin> [<i/o pin>...<i/o pin>]
PRIORITY m, PROGCNTL, CPU, OUTCNTL;
PSWITCH
RESET <p1> [<p2>...<p10>]
RETURN [ON] [NOT] [<cond>]
SET <p1> [<p2>...<p10>]
SWITCH <case group>
WHILE [NOT] <cond>
```

The CPU-Operator Assignment

```
move
   <dest> := <src>
arithmetic expression
   <dest> := <arg1> <+/-> <arg2> <+/-><arg3>
logical expression
   <dest> := <arg1> <logical operator> <arg2>
increment, decrement, invert, unary minus
   <dest> := <opr> <src>
macro expression
   <dest> := <arg1> [* | /] <arg2>
shift RAM
   <Rx> = Rx <shft opr> <shft arg>
shift RAM and q
   <QRX> = Q <shft opr> <shft arg> <shft opr> <shft arg>
```

The OUTCNTL Operator

```
OUT <arg1> [<arg2>...<arg16>]
```

System Development Tools

PAC1000 System Development Tools are a complete set of PC-based development tools. They provide an integrated easy-to-use software and hardware environment to support PAC1000 development and programming.

The tools run on an IBM-XT, AT, PS2 or compatible computer running MS-DOS version 3.1 or later. The system must be equipped with 640 Kbytes of RAM and a hard disk.

Hardware

The PAC1000 System Programming Hardware consists of:

- ❑ WS6000 MagicPro Memory and PSD Programmer (XT, AT only)
- ❑ WS6010 Package Adaptor (88-Pin Ceramic Pin Grid Array) for the MagicPro Remote Socket Adaptor Unit
- ❑ WS6013 Package Adaptor (100-Pin PQFP) for the MagicPro Remote Socket Adaptor Unit

The MagicPro Programmer is the common hardware platform for programming all WSI programmable products. It consists of the IBM-PC plug-in Programmer Board and the Remote Socket Adaptor Unit

Software

The PAC1000 System Development Software consists of the following:

- ❑ WISPER Software—PSD Software Interface
- ❑ IMPACT Software—Interface Manager for PAC1000
- ❑ PACSEL Software—System Entry Language
- ❑ PACSIM Software—Functional Simulator
- ❑ PACPRO Software—Device Programming Software

WISPER and IMPACT software provide a menu-driven user interface enabling other tools to be easily invoked by the user.

The system design is entered into PACSEL source program files using an editor chosen by the user. PACSEL supports assembly-level and high-level Macro programming.

The PACSEL Assembler produces object code format in single or multiple modules, which are then linked by the PACSEL Linker into a single load file with a format suitable for PACSIM and PACPRO.

The PACSIM functional simulator enables the user to test and debug programs by examining the state of PAC1000 internal registers before and during a complete functional simulation of the device.

PACPRO software programs PAC1000 devices by using the MagicPro hardware and the socket adapter.

The programmed PAC1000 is then ready to be used.

Support

WSI provides a complete set of quality support services to registered owners. These support services include the following:

- ❑ 12-month Software Updates.
- ❑ Hotline to WSI Application Experts—For direct design assistance.
- ❑ 24-Hour Electronic Bulletin Board—For design assistance via dial-up modem.

Training

WSI provides in-depth, hands-on workshops for the PAC1000 and the System Development Tools. Workshop participants will learn how to develop and program their own high-performance microcontrollers. Workshops are held at the WSI facility in Fremont, California.

Programming/ Erasing

Refer to the PAC1000 Users Manual found with the PAC1000-Gold and the PAC1000-Silver.

Ordering Information – PAC1000

Part Number	Speed (MHz)	Package Type	Package Drawing	Operating Temperature	Manufacturing Procedure
PAC1000-12Q	12	100-Pin Plastic Quad Flat Package	Q1	Commercial	Standard
PAC1000-12V	12	92-Pin Ceramic Quad Flatpack	V1	Commercial	Standard
PAC1000-12VI	12	92-Pin Ceramic Quad Flatpack	V1	Industrial	Standard
PAC1000-12VM	12	92-Pin Ceramic Quad Flatpack	V1	Military	Standard
PAC1000-12VMB	12	92-Pin Ceramic Quad Flatpack	V1	Military	MIL-STD-883C
PAC1000-12X	12	88-Pin Ceramic Pin-Grid Array	X1	Commercial	Standard
PAC1000-12XI	12	88-Pin Ceramic Pin-Grid Array	X1	Industrial	Standard
PAC1000-12XM	12	88-Pin Ceramic Pin-Grid Array	X1	Military	Standard
PAC1000-12XMB	12	88-Pin Ceramic Pin-Grid Array	X1	Military	MIL-STD-883C
PAC1000-16X	16	88-Pin Ceramic Pin-Grid Array	X1	Commercial	Standard

Ordering Information – System Development Tools

Part Number	Contents
PAC1000-GOLD	WISPER Software IMPACT Software PACSEL Software PACSIM Software PACPRO Software User's Manual WSI-Support WS6000 MagicPro Programmer One Socket Adaptor and Two PAC1000 Product Samples
PAC1000-SILVER	WISPER Software IMPACT Software PACSEL Software PACSIM Software PACPRO Software User's Manual WSI-Support
WS6000	MagicPro Programmer IBM PC Plug-in Adaptor Card Remote Socket Adaptor
WS6010	88-Pin CPGA Adaptor Used with the WS6000 MagicPro Programmer
WS6013	100-Pin PQFP Adaptor Used with the WS6000 MagicPro Programmer
WSI-Support	Support Services, including: ❑ 12-month Software Update Service ❑ Hotline to WSI Application Experts ❑ 24-hour Access to WSI Electronic Bulletin Board
WSI-Training	Workshops at WSI, Fremont, CA For details and scheduling, call PSD Marketing, (510) 656-5400

PAC1000 Instruction Set

Section Index

PACSEL Language

Programmable Peripheral
PAC1000 Instruction Set
PACSEL Language

Overview

PACSEL, the PAC1000 System Entry Language, is an assembly-level language with macro constructs. While it is not a true macro assembler (i.e. you cannot write your own macros), it does provide a very convenient set of pre-constructed high-level macros for many common programming needs.

This section gives an overview of PACSEL operations, directives, and development rules. Consult the reference sections in this chapter for specific information on how PACSEL constructs control the PAC1000 Programmable Peripheral Controller.

Operations

Each PACSEL instruction performs three operations:

- ❑ Program Control (PROGCNTL)
- ❑ CPU
- ❑ Output Control (OUTCNTL)

Each instruction is executed in a single cycle; the three operations are executed in parallel. In conventional peripheral controllers, separate instructions are required to perform each of these operations: Program Control operations (jumps, calls, and returns) to control the program flow; CPU operations to do logical, arithmetic, and shift tasks; and various forms of Output Control operations.

Each PACSEL statement has an optional label and three components:

```
[label:]PROGCNTL,CPU,OUTCNTL;
```

The PROGCNTL component determines which statement is to be executed next; the CPU component determines which operation is to be performed by the CPU; and the OUTCNTL component determines the state of the control outputs.

Commas (,) separate the components and a semicolon (;) marks the end of the statement.

The PROGCNTL, CPU, and OUTCNTL components can come in any order. Assembler operators and macro operators can be used together in the same statement. The available operators are summarized at the end of this section. In some cases, the same mnemonic is used to specify identical operations at both the assembler and the macro level.

You may:

- ❑ Specify all the components in the same statement in order to perform the operations in parallel:

```
PROGCNTL, CPU, OUTCNTL;
```

- ❑ Specify components one at a time:

```
CPU;
PRGCNTL;
OUTCNTL;
```

- ❑ Use components in any combination:

```
PROGCNTL, CPU;
PROGCNTL, OUTCNTL;
CPU, OUTCNTL;
```

Operations (Cont.)

WSI recommends that the user adhere to a specific ordering of these components and specific groupings of assembler-level and macro operators, e.g. in separate files. This manual uses the PROGCNTL, CPU, OUTCNTL ordering.

When PROGCNTL is omitted, the implied instruction is CONTinue, that is, proceed to the next control instruction. When CPU is omitted, the implied instruction is NOP. When OUTCNTL is omitted, the implied instruction is MAINTain, that is, maintain the most recent OUTCTL, in Assembler order.

There is a class of supplemental CPU (sCPU) instructions which can follow certain primary CPU instructions with one or more spaces as a delimiter:

```
PROGCNTL,  CPU sCPU,  OUTCNTL;
```

An sCPU instruction must follow a valid CPU instruction and can not stand alone.

Summary of PACSEL Assembler Operators

Mnemonic	Arguments	Meaning
The PROGCNTL Operators		
ACSIZE	<16/22>	Set A Counter Size
AI		Allow Interrupts
CALL	<LABEL | LCPTR | FIFO>	Uncond Branch Subrtn
CALLC	<COND> <LABEL | FIFO>	Cond Branch Subrtn
CALLNC	<COND> <LABEL | FIFO>	Inv Cond Bran Subrtn
CCASE	<CG> <VALUE>	Branch Subrtn Caseblk
CLI	<MASK>	Clear Interrupt
CONT		Continue
CPI	<VALUE>	Prioritized Subrtn
DI	<MASK>	Disable Interrupt
DSS		Disable SSM
EI	<MASK>	Enable Interrupt
ESS		Enable SSM
JCASE	<CG> <VALUE>	Uncond Branch Caseblk
JMP	<LABEL | LCPTR | FIFO | TOS>	Uncond Branch
JMPC	<COND> <LABEL>	Cond Branch
JMPNC	<COND> <LABEL>	Inv Cond Branch
JPI	<VALUE>	Prioritized Branch
LDBP	<VALUE | LABEL>	Load BP Reg
LDBPD		Load BP Comp Value
LDLC	<VALUE | LABEL>	Load Counter
LDLCD		Load Ctr Comp Value
LOOPNZ	<LABEL>	Repeat Branch CNTRNZ
PLDLC	<VALUE | LABEL>	Push VALUE & LDCTR
PLDLCD		Push Comp Val & LDCTR
POP		Pop Stack
POPLC		Pop Stack to Cntr
PUSHLC		Push Cntr
RESTART		Branch to 0
RET	[<LABEL>]	Return
RC	<COND> [<LABEL>]	Conditional Return
RNC	<COND> [<LABEL>]	Inv Cond Return
RSTCON	<MASK>	Reset Control Reg
RSTIO	<MASK>	Reset I/O Config Reg
RESTMODE	<MASK>	Reset Mode Reg
SETCON	<MASK>	Set Control Reg
SETIO	<MASK>	Set I/O Config Reg
SETMODE	<MASK>	Set Mode Reg
TWB	<COND> <LABEL>	Three-way Branching
TWBC	<COND> <LABEL>	Converse Three-way Branching

Summary of PACSEL Assembler Operators (Cont.)

Mnemonic	Arguments	Meaning
The CPU Operators		
ADC	<ARG1> <ARG2> [<ARG3>] [sCPU]	Add with Carry
ADD	<ARG1> <ARG2> [<ARG3>] [sCPU]	Add
AND	<ARG1> <ARG2> [<ARG3>] [sCPU]	Bitwise AND
CLR	<REG>	Clear Register
CMP	<ARG1> [<ARG2>	Compare
DEC	<ARG1> [<ARG2>] [sCPU]	Decrement
INC	<ARG1> [<ARG2>] [sCPU]	Increment
INV	<ARG1> [<ARG2>] [sCPU]	Invert
MOV	<DEST> <SRC> [sCPU]	Move SRC to DEST
NOP		No Operation
OR	<ARG1> <ARG2> [<ARG3>] [sCPU]	Bitwise or
RDFIFO		Read FIFO Data to Reg
SBC	<ARG1> <ARG2> [<ARG3>] [sCPU]	Sub with Carry
SHLRQ	<REG> <RARG> <QARG>	Shift Left Reg & Q
SHLR	<REG> <RARG>	Shift Left Reg
SHRRQ	<REG> <RARG> <QARG>	Shift Right Reg & Q
SHRR	<REG> <RARG>	Shift Right Reg
SUB	<ARG1> <ARG2> [<ARG3>] [sCPU]	Subtract
XOR	<ARG1> <ARG2> [<ARG3>] [sCPU]	Exclusive OR
XNOR	<ARG1> <ARG2> [<ARG3>] [sCPU]	Exclusive NOR
The sCPU Operators		
ARDREG	<ARG1> <ARG2>	Read Reg to 2nd Dest
ASHLR	<SOURCE>	Shift Reg Left
ASHLRQ	<RSOURCE> <QSOURCE>	Shift Q & Reg Left
ASHRR	<SOURCE>	Shift Reg Right
ASHRRQ	<RSOURCE> <QSOURCE>	Shift Q & Reg Right
AWREG	<ARG1>	Write to 2nd Dest
The MACRO Operators		
DIV	<ARG1> <ARG2> <ARG3>	Divide
MUL	<ARG1> <ARG2> <ARG3>	2'S Comp Multiply
The OUTCNTL Operators		
MAINT		Maintain Prev Value
OUT	<VALUE | EXPRESSION	OUTPUT

Summary of PACSEL Macro Operators

The PROGCNTL Operators

```
CALL <label | LCPTR | FIFO> [ON] [NOT] [<cond>]
CASE N, PROGCNTL, CPU, OUTCNTL;
CLEAR <int level> [...<int level>]
CONFIGURE <pml> [<pm2>...<pm10>]
CONT
DISABLE <int level> [<int level>...<int level>]
ELSE
ENABLE <int level> [<int level>...<int level>]
ENDFOR
ENDIF
ENDPSWITCH
ENDSWITCH
ENDWHILE
FOR <value>
GOTO <label | LCPTR |FIFO | TOS> [ON] [NOT] [<cond>]
IF [NOT] <cond>
INPUT <i/o pin> [<i/o pin>...<i/o pin>]
LOADBP <value>
OUTPUT <i/o pin> [<i/o pin>...<i/o pin>]
PRIORITY m, PROGCNTL, CPU, OUTCNTL;
PSWITCH
RESET <pl> [<p2>...<p10>]
RETURN [ON] [NOT] [<cond>]
SET <pl> [<p2>...<p10>]
SWITCH <case group>
WHILE [NOT] <cond>
```

The CPU-Operator Assignment

```
move
   <dest> := <src>
arithmetic expression
   <dest> := <arg1> <+/-> <arg2> <+/-><arg3>
logical expression
   <dest> := <arg1> <logical operator> <arg2>
increment, decrement, invert, unary minus
   <dest> := <opr> <src>
macro expression
   <dest> := <arg1> [* | /] <arg2>
shift RAM
   <Rx> = Rx <shft opr> <shft arg>
shift RAM and q
   <QRX> = Q <shft opr> <shft arg> <shft opr> <shft arg>
```

The OUTCNTL Operator

```
OUT <arg1> [<arg2>...<arg16>]
```

Directives

Directives give PACSEL the information it needs to correctly assemble each program module. For instance, an assembler source file must have segment directives to declare a name for each relocatable code segment in the file.

A segment is the smallest unit of code which can be relocated by the linker. Normally, each assembler source module begins with a segment declaration. The directive gives the segment a name; the absolute address of the segment can also be specified. The syntax is:

```
segment < segment name > [ , abs ( <value> ) ];
```

A source file can contain more than one segment. Each source file must be terminated with the end directive:

```
end;
```

Labels from other segments which are referenced in the current segment must be declared with the external directive:

```
external <label1> , <label2> , ... <label1>;
```

Labels from the current segment which are also referenced by other segments must be declared with the entry directive:

```
entry <label1> , <label 2> , ... <label1n>;
```

Every label that is declared external in one segment must be declared as entry in some other segment. Local labels (labels referenced only in the current segment) need not be declared.

In summary, a source file looks like this:

```
  segment test 1;
  entry a1, a2;
  external x1, x2;
a1: ...
a2:
        .
        .
        .
  JMP x1;
        .
        .
        .
  CALL x2;
        .
        .
        .
  end;
```

One source file can be incorporated into another by means of the include directive:

```
include '<filename>';
```

Before assembly, this line will be replaced with the contents of the named source file.

The org and align directives are used to control the location of the instruction word which follows them. The org directive sets the current program counter to the location indicated by its argument, relative to the beginning of the current segment. For example, the directive

```
org h'10';
```

will place the next word at location h'10' relative to the beginning of the segment.

Directives (Cont.)

The align directive sets the program counter to the next-higher multiple of its argument. For example, if the program counter currently has the value h'155', the directive:

```
align 16;
```

will place the next word at location h'160', that is, at the next available location which is divisible by 16. The linker will preserve the specified alignment.

Symbolic constants can be defined with the equ directive, as follows:

```
<symbol> equ <value>;
```

The "value" can be a number or a previously defined symbol.

The "alias" directive can be used to alias signal and register names, to user defined names. Aliases are permitted for the following:

Q	1NT0 .. INT7
AOR	
ACH	IO0 .. IO7
ACL	CC0 .. CC7
BC	
DOR	R0 .. R31
IOR	AIR
IIR	
DIR	

The aliases can also be placed in a file with extension ".ALS". This file has to be included in every source file which uses aliased signal names. The statement syntax is:

```
alias <alias name> <signal name> ;
```

Remember that PACSEL is case-sensitive and hence "alias" keyword should be in lower case.

Example:

```
alias SVADD R3 ;
```

Wherever R3 is used in .mal file, the aliased name SVADD can be used. For example,

```
mov R3 5;
```

This can be written as

```
mov SVADD 5;
```

This directive can be put together in a ".ALS" file or can be used directly in ".mal" source file. It puts together in a ".ALS" file, then the same file can also be used in the PACSIM simulator.

Programming Guidelines

These are the guidelines for writing PACSEL programs:

❑ Source File naming

The assembler source filename must have a .mal extension.

❑ Case Sensitivity

PACSEL is case-sensitive. Observe the conventions given in the manual. In general, all the instructions and arguments are upper-case. All the directives are lower-case.

❑ Whitespace Requirements

PACSEL is whitespace-sensitive. In general, arguments and operators must be surrounded by blanks. Whenever in doubt, use a blank (or a tab).

Use whitespace freely to emphasize program structure.

❑ Comments

Any text enclosed by "/!" and "*/" is not processed by the PACSEL assembler. Such comments may span lines or pages. Comments may not be nested. For example:

```
/* This is a legal
                comment */
```

However,

```
/*
        /* This comment is nested; an error will result */
*/
```

Comments may also be used in Link Command Files.

❑ Special Characters

PACSEL source files may contain the standard set of printable ASCII characters, plus tabs, spaces, carriage returns, and linefeeds. No other characters are allowed.

In particular, some word processors, in document mode, set the hi-order bit of some ASCII characters in a file for internal purposes. Although these characters will display correctly within the word processor, they will not be accepted by PACSEL, and the resulting error messages may not indicate the cause.

❑ Operation Arguments

In general, arguments are names of registers or immediate, constant values. The allowed registers for a given argument are specified in the documentation for each instruction. A constant is a number value in the range

```
0 <= value <= h'FFFF'
0 <= value <= 65535
```

For example,

```
1289                  (a decimal number)
h'FA48'               (a hexadecimal number)
o'4777'               (an octal number)
b'0110111000000000'   (a binary number)
```

Symbolic constants, previously defined by an "equ statement, may be used in place of numeric values, for example,

```
SUCCESS equ 1 ;
R1 := SUCCESS
```

Note: When any constant is used as an argument in a CPU instruction, the only allowed PROGCNTL instruction is CONT.

Programming Guidelines (Cont.)

❑ Assembly-time expression evaluation

PACSEL supports the use of assembly-time variables and expressions to compute the values of constants which can then be used as arguments to the PACSEL instructions. This facility can be useful, for instance, in computing values to be loaded into the loop counter or the OUTCNTL field.

The operators supported are:

Unary:	-	2's complement
	~	1's complement
	--	decrement
	++	increment
Binary:	*	multiply
	/	divide
	%	remainder
	+	add
	-	subtract
	<<	shift left
	>>	shift right
	&	bitwise AND
	|	bitwise OR

Assignments are made using the 'set' and '=' operators.

These concepts are illustrated in the following example:

```
segment asmexp;
                integer A, B, C;
                et A = 5;
                set B = A + 2
                set C = A & B;
                R1 := R1 + $2, OUT C;
end;
```

For this example, the assembler generates just one line of code, treating the computed value of C as a constant:

```
000000: E000 0003 0422 0005
```

It is important to distinguish between run-time assignments (like "R1 := R1 + R2" in the example) and assembly-time assignments (like "set C = A & B"); the latter do not generate code.

❑ Restrictions on PROGCNTL, CPU, and OUTCNTL combinations

In a few cases, there are restrictions on what you can do in each of the sections of combined PROGCNTL, CPU, and OUTCNTL instructions. These are:

When any constant is used in the CPU operation, you may only use CONT as the PROGCNTL operation.

Configuration operations use the PROGCNTL bits, so you cannot do anything other than CONT. The CPU and OUTCTL instructions are not limited.

PACSEL Assembler Reference

ADC	
Instruction Type:	CPU
Operation:	dest = src1 + src2 + CY

Syntax:	`[label:] [PROGCNTL,] ADC dest/src src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] ADC dest src1 src2 [,OUTCNTL];`

Description:

In the first form, ADC adds the source register and the destination register, and the value of the CY bit, then places the result in the dest register. In the second form, two source registers and CY are added and stored in the destination register. One of the sources may be the same as the destination register.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF = Arithmetic Flags: Z, CY, S, O.

Example 1:

```
ADC R1 R1;          /* double r1 and add CY */
```

Example 2:

```
MOV R31 h'FFFF';    /* load immediate value */
ADD R31 h'1';       /* add immediate */
ADC R31 h'0';       /* R31 now is one; see ADD */
```

Example 3:

```
ADC BC R23;         /* BC = BC + R23 + CY */
```

Example 4:

```
ADC BC R23 R24;     /* BC = R23 + R24 + CY */
```

Example 5:

```
CONT ,
ADD BC R23 R24 ,    /* full instruction format */
OUT h'A5A5';
```

PACSEL Assembler Reference (Cont.)

ADD	
Instruction Type:	CPU
Operation:	dest = src1 + src2

Syntax:	`[label:] [PROGCNTL,] ADD dest/src src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] ADD dest src1 src2 [,OUTCNTL];`

Description:

In the first form, ADD adds the source and the destination registers and places the result in the destination. In the second form, two source registers are added and the result is stored in the destination register. One of the sources may be the same as the destination register.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

See Also: ADC

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

Example 1:

```
ADD      R1 R1;          /* double R1 */
```

Example 2:

```
MOV      R31 h'FFFF';    /* load immediate value */
ADD      R31 h'1';       /* add immediate */
ADD      R31 h'0';       /* R31 now is zero; see ADC */
```

Example 3:

```
ADD      BC R23;         /* add R23 to block counter */
```

Example 4:

```
ADD      BC R23 R24;     /* R23 + R24 to block counter */
```

Example 5:

```
CONT ,
ADD BC R23 R24 ,         /* full instruction format */
OUT h'A5A5';
```

PACSEL Assembler Reference (Cont.)

AI	
Instruction Type:	PROGCNTL
Operation:	Allow interrupts
Syntax:	`[label:] AI [,CPU] [,OUTCNTL];`

Description:

While an interrupt is being serviced, a hardware locking mechanism prevents any other interrupt from getting serviced. This lock remains in effect until it is explicitly cleared by means of the AI instruction. The AI instruction is normally placed in the interrupt service routine.

Condition Codes affected: None

Example 1:

```
INT3_SERV:    /* service routine for INT3 */
    .
    .           /* service the interrupt */
    .
   AI;          /* re-activate the service mechanism */
   RET;         /* return to main program */
```


AND	
Instruction Type:	CPU
Operation:	dest = src1 AND src2

Syntax:	`[label:] [PROGCNTL,] AND dest/src src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] AND dest src1 src2 [,OUTCNTL];`

Description:

In the first form, this operator ANDs the source and the destination and places the result in the destination register. In the second form, two sources are ANDed and stored in the destination register. One of the sources may be the same as the destination register.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF = Arithmetic Flags: Z, CY, S, O

PACSEL Assembler Reference (Cont.)

AND (Cont.)

Example 1:

```
AND       R1 R2;              /* R1 = ( R1 AND R2) */
```

Example 2:

```
AND       R31 h'0FFFF';       /* AND immediate mask */
```

Example 3:

```
MOV       R23 h'123';         /*load register with constant */
MOV       R24 h'FFFE';        /* load register with mask */
ANDBC     R23 R24;            /* Mask and load Block Counter */
```

Example 4:

```
CONT ,
AND IOR R23 ,                 /* full instruction format */
OUT h'A5A5';
```

ARDREG	
Instruction Type:	Supplementary CPU
Operation:	Read, store register independent of primary CPU operation
Syntax:	`[label:] [PROGCNTL,] CPU ARDREG arg1 arg2 [,OUTCNTL];`

Description:

This instruction reads the internal register (R0...R31) specified by "arg2" and stores the value in the external register specified by "arg1". Only internal registers may be transferred to external registers by this operation, and the destination register of the primary CPU operation may be an internal register only if this supplementary instruction is used. Otherwise, the primary CPU operation is unrestricted in type and arguments.

Condition Codes affected: (by primary CPU operation only)

Example 1:

```
ADD R3 BC ARDREG AOR R3; /* R3 <- R3+BC, parallely AOR <- R3 */
```

PACSEL Assembler Reference (Cont.)

ASHLR	
Instruction Type:	Supplementary CPU
Operation:	Shift Left Register after primary CPU operation
Syntax:	`[label:] [PROGCNTL,] CPU ASHLR src [,OUTCNTL];`

Description:

This instruction shifts the result of the primary CPU operation left one bit before the primary result is stored in the destination of the primary operation. The data shifted in depends on the source operand:

Z	– the Zero bit flag
CY	– the Carry bit flag
S	– the Sign bit flag
0	– a binary '0'
1	– a binary '1'
RMSB	– the Most Significant Bit of this register
QMSB	– the Most Significant Bit of the Q register
SDAT	– Serial Data port in/out.

This instruction is valid only if the primary CPU operation uses internal registers (R0...R31, Q) as sources and destinations.

Condition Codes affected: (by the primary CPU operation only)

Example 1:

```
CMP R0 R1 ;
ADD R4 Q R5 ASHLR Z ;    /* R4 <-- (Q+R5) shifted left with Z
                         flag into LSB */
```

PACSEL Assembler Reference (Cont.)

ASHLRQ	
Instruction Type:	Supplementary CPU
Operation:	Shift left register and Q after primary CPU operation
Syntax:	`[label:] [PROGCNTL,] CPU ASHLRQ rsource qsource [,OUTCNTL];`

Description:

This instruction shifts the result of the primary CPU operation left one bit before storing the primary result in the destination of the primary operation. In addition, the previous value of Q is shifted left one bit.

The data shifted in depends on the rsource and qsource operands:

Z	– the Zero bit flag
CY	– the Carry bit flag
S	– the Sign bit flag
0	– a binary '0'
1	– a binary '1'
RMSB	– the Most Significant Bit of this register
QMSB	– the Most Significant Bit of the Q register
SDAT	– Serial Data port in/out.

This supplementary CPU operation is valid only if the primary CPU operation operates on internal registers (R0...R31, Q) only.

Condition Codes affected: (by the primary CPU operation only)

Example 1:

```
MOV Q h'007A';
SUB R1 RS;
ADD R4 R5 ASHLRQ Z CY;  /* R4<- (R4+R5) shifted left with Z flag
                           into LSB; simultaneously, shift Q left 1
                           bit with CY entering LSB */
```

PACSEL Assembler Reference (Cont.)

ASHRR	
Instruction Type:	Supplementary CPU
Operation:	Shift Right Register after primary CPU operation
Syntax:	`[label:] [PROGCNTL,] CPU ASHRR source [,OUTCNTL];`

Description:

This instruction shifts the result of the primary CPU operation right one bit before the primary result is stored in e destination of the primary operation. The data shifted in depends on the source operand:

Z	– the Zero bit flag
CY	– the Carry bit flag
S	– the Sign bit flag
0	– a binary '0'
1	– a binary '1'
RLSB	– the Lease Significant Bit of this register
QLSB	– the Least Significant Bit of the Q register
SDAT	– Serial Data port in/out.

This instruction is valid only if the primary CPU operation uses internal registers (R0...R31, Q) as sources and destinations.

Condition Codes affected: CY, Z, S, O

Example 1:

```
CMP R0 R1;
ADD R4 Q R5 ASHRR CY;
```

PACSEL Assembler Reference (Cont.)

ASHRRQ	
Instruction Type:	Supplementary CPU
Operation:	Shift Right Register and Q after primary CPU operation
Syntax:	`[label:[ [PROGCNTL,] CPU ASHRRQ rsource qsource [,OUTCNTL];`

Description:

This instruction shifts the result of the primary CPU operation right one bit before storing the primary result in the destination of the primary operation. In addition, the previous value of Q is shifted right one bit.

The data shifted in depends on the rsource and qsource operands:

- Z – the Zero bit flag
- CY – the Carry bit flag
- S – the Sign bit flag
- 0 – a binary '0'
- 1 – a binary '1'
- RLSB – the Least Significant Bit of this register
- QLSB – the Least Significant Bit of the Q register
- SDAT – Serial Data port in/out.

This supplementary CPU operation is valid only if the primary CPU operation operates on internal registers (R0...R31, Q) only.

Condition Codes affected: (by the primary CPU operation only)

Example 1:

```
MOV Q h'007A';
SUB R1 R2;
ADD R4 R5 ASHRRQ Z CY;     /* R4 <-- (R4+R5) shifted right with Z
                           flag into MSB; simultaneously, shift Q
                           right 1 bit with CY entering MSB */
```

AWREG	
Instruction Type:	Supplementary CPU
Operation:	Write result of primary CPU operation to second destination
Syntax:	`[label:] [PROGCNTL,] CPU AWREG arg1 [,OUTCNTL];`

Description:

This instruction stores the result of the primary CPU operation in the internal register specified by "arg1". Only internal registers (R0...R31, Q) are allowed destinations. If this instruction is used, the primary operation should have an external register as the destination operand. Otherwise, the primary CPU operation is unrestricted in type and arguments.

Condition Codes affected: (by primary CPU operation only)

Example 1:

```
ADD BC R3 AWREG R2;    /* BC<-- + R3, simultaneously R2 <--
                       (BC+R3)  */
```

PACSEL Assembler Reference (Cont.)

CALL	
Instruction Type:	PROGCNTL
Operation:	Call to subroutine/thru pointer/via FIFO

Syntax:	`[label:] CALL label2 [,CPU] [,OUTCNTL];`
	or
	`[label:] CALL LCPTR [,CPU] [,OUTCNTL];`
	or
	`{label:] CALL FIFO [,CPU] [,OUTCNTL];`

Description:

CALL is an unconditional call to the subroutine. In the first form, the target is a program label.

The keyword LCPTR may be substituted for "label2", in which case the current value of LC (Loop Counter) is used as an execution pointer. The 10-bit LC value is loaded into the program counter. In this case, the LCPTR register must be explicitly loaded (see LDLC and LDLCD) prior to this operation.

In the third form, the keyboard FIFO may be substituted for "label2," in which case the top of the FIFO is used as the call target.

In all cases the next instruction address is pushed on the stack.

Condition Codes affected: FIFO flags (Form 3), STKF

Example 1:

```
CALL E1b,
 NOP,
 OUT h'46';    /* output ASCII F during CALL cycle*/

E1b:           RET;
```

Example 2:

```
LDLC 270;      /* call through pointer */
CALL LCPTR;
E2b:           RET;
```

Example 3:

```
CALL FIFO;     /* main program for host-driven PAC1000 */
JMP E3;
```

PACSEL Assembler Reference (Cont.)

CALLC	
Instruction Type:	PROGCNTL
Operation:	Call if Condition TRUE

Syntax:	`[label:] CALLC cc label 2 [,CPU] [,OUTCNTL];`
	or
	`[label:] CALLC cc FIFO [,CPU] [,OUTCNTL]`

Description:

The condition code specified by "cc" is evaluated. If it is TRUE, then control branches to the code at the specified label. In the second form, the top of the FIFO is the call target and only FIFO flags may be specified as the condition to test.

The next instruction address is pushed on the stack if the call is performed.

The condition Codes are:

INTR	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Condition Codes affected: FIFO flags (Form 2), STKF

Example 1:

```
CALLC Z E1b,          /* CALL on condition code 'zero' */
 NOP,
 OUT h'45';           /* output ASCII E during CALL cycle */
.
.
.
E1b:  CALLC CY E1c;   /* CALL on carry, default CPU and
                      OUTCNTL */
```

PACSEL Assembler Reference (Cont.)

CALLNC	
Instruction Type:	PROGCNTL
Operation:	Call if condition not TRUE

Syntax:	`[label:] CALLNC cc label2 [cCPU] [,OUTCNTL];`
	or
	`[label:] CALLNC cc FIFO [,CPU] [,OUTCNTL];`

Description:

The condition code specified by "cc" is evaluated. If it is FALSE, then control branches to the code at the specified label. In the second form, the top of the FIFO is the call target and only FIFO flags may be specified as the condition to test.

The next instruction address is pushed on the stack if the call is performed.

The condition Codes are:

INTR	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Condition Codes affected: FIFO flags (Form 2), STKF

Example 1:

```
CALLNC Z E1b,          /* CALL on condition code ' not zero' */
 NOP,
 OUT h'45';            * output ASCII E during CALL cycle */
.
.
.
E1b:  CALLNC CY E1c;   /* CALL on not carry, */
```

PACSEL Assembler Reference (Cont.)

CCASE	
Instruction Type:	PROGCNTL
Operation:	Call to Case Block
Syntax:	`[label:] CCASE casegroup address [,CPU] [,OUTCNTL];`

Description:

CCASE performs an unconditional CASE selection using the Case Block at the address specified. Case selection is based on the 4 bits of the Case Group specified:

```
'CGO' specifies [CC3, CC2, CC1, CC0]
'CG1' specifies [CC7, CC6, CC5, CC4]
'CG2' specifies [INTR, BCZ, FIOR, FICD]
'CG3' specifies [Z, O, S, CY]
```

One of 16 successive instructions starting at the Case Block address will be executed. If the instruction does not include a jump operation, the next sequential instruction will be executed. Ordinarily, then, the instructions in a Case Block will include jumps.

The Case Block must start at a location whose address contains zeros in the lower four bits. The address specified must be a numerical value or a symbolic constant whose value is previously defined in this module. The actual location of the Case Block must be resolved at link time, by specification in a link directive.

Condition Codes affected: STKF

Example 1:

```
/* This example outputs the binary value of Case Group 1 (CG1) on
the F outputs for one cycle, bracketed by the value -1. */

E1c equ h'160'      /* The case block should be located at h'160 by
                    means of a link directive */
CCASE CG1 E1C,
 NOP ,
 OUT h'FFFF';       /* output -1 during call cycle*/

/* The case block in general, will be in another segment: */
    segment CASE_EXAMPLE;
CB1:  JMP ECEND, NOP, OUT 0;
  JMP ECEND, NOP, OUT 1;
  JMP ECEND, NOP, OUT 2;
  JMP ECEND, NOP, OUT 3;
  JMP ECEND, NOP, OUT 4;
  JMP ECEND, NOP, OUT 5;
  JMP ECEND, NOP, OUT 6;
  JMP ECEND, NOP, OUT 7;
  JMP ECEND, NOP, OUT 8;
  JMP ECEND, NOP, OUT 9;
  JMP ECEND, NOP, OUT 10;
  JMP ECEND, NOP, OUT 11;
  JMP ECEND, NOP, OUT 12;
  JMP ECEND, NOP, OUT 13;
  JMP ECEND, NOP, OUT 14;
  JMP ECEND, NOP, OUT 15;
ECEND:
  RET, NOP, OUT h'FFFF';        /* all cases end up here */
```

The appropriate link directive is: `locate CASE_EXAMPLE h'160';`

PACSEL Assembler Reference (Cont.)

CLI	
Instruction Type:	PROGCNTL
Operation:	Clear Interrupt Mask bits
Syntax:	`[label:] CLI mask [,CPU] [,OUTCNTL];`

Description:

Use the mask to clear pending interrupts. Bits set to 1 in the mask clear the corresponding interrupt. Zero bits have no effect.

This is the mask format:

MASK7	MASK6	MASK5	MASK4	MASK3	MASK2	MASK1	MASK0

The CLI function is automatically performed by the hardware when the interrupt is serviced. However, if the interrupt is masked, or if interrupts are disabled, CLI must be used to clear pending interrupts.

Condition Codes affected: none

See Also: DI, EI

Example 1:

```
/* This example illustrates the use of CLI at the end of
an interrupt service routine */

   org h'8';              /* external interrupt 0 vector */
   JMP INTO_SERVICE;

   org   h'100';          /* arbitrary */
INT0_service:
.
.
.
   CLI   b'00000001';     /* clear interrupt 0 request */
   RET;                   /* return-from-interrupt */

/* This example illustrates the use of CLI as part of system
initialization procedures */
```

Example 2:

```
   org h'0';              /* start-up location */
   JMP INIT;

   org h'10' ;
INIT:       CLI h'FF';    /* clear any pending interrupts */
```

CLR	
Instruction Type:	CPU
Operation:	Reset Register
Syntax:	`[label:] {PROGCNTL,]  CLR reg [,OUTCNTL]`

Description:

CLR resets the specified register to 0. This instruction does not use the branch bits, and so allows a JMP or CALL instruction in the PROGCNTL section in the same cycle. The specified register can be any of the internal registers (R0..R31 or Q) or any of the external registers (AOR, ACH, ACL, BC, DOR, IOR).

Condition Codes affected: CY, Z

Example 1:

```
JMP label1, CLR R31;        /* branch to label1 and reset R31
                               at the same time */
```

PACSEL Assembler Reference (Cont.)

CMP	
Instruction Type:	CPU
Operation:	Compare src1 and src2
Syntax:	`[label:] [PROGCNTL,] CMP srcl src2 [,OUTCNTL];`

Description:

CMP compares the two sources and sets the status flags accordingly. Neither register is affected. The comparison is done by computing (src1 - src2) and discarding the numerical result.

Status of Flags following CMP A B:

Relation	Unsigned Numbers	2's-Complement Numbers
A=B	Z=1	Z=1
A≠B	Z=0	Z=0
A≥B	CY=1	S xnor O=1
A<B	CY=0	S xnor P=1
A>B	CY Z=1	(S xnor O) and Z=1
A≤B	CY or Z=1	(S xor O) or Z=1

Legend:

CY = carry
O = overflow
S = sign
Z = zero

The sources can be: IIR, AIR, SWPV, DIR, RO...R31, Q, AOR, ACH, ACL, BC, <const>

Condition Codes affected: CY, Z, S. O

Example 1:

```
/* This example illustrates using CMP to convert a single upper
case hex digit in R20 into binary, returning the result in R21  */
MOV       R21 1;          /* default value to result */
CMP       R20 h'46';      /* is the code <= ASCII F? */
RC        CY;             /* return with default if not */
CMP       R20 h'30';      /* is the code > ASCII SP? */
RC        CY;             /* return with default if not */
CMP R20 h'39';            /* is it greater than '9' ? */
JNC       CY HDTB1;       /* jump if not */
SUB       R20 h'31';      /* map 'A' --> 10, 'B' -> 11, etc. */
CMP       R20 9;          /* is it between? */
RC        CY;             /* return with default if so */
MOV       R21 R20;        /* else transfer value for return */
RET;
```

PACSEL Assembler Reference (Cont.)

CONT	
Instruction Type:	PROGCNTL
Operation:	Continue
Syntax:	`[label:] CONT [,CPU] [,OUTCNTL];`

Description:

CONT is the default PROGCNTL operation, that is, if no PROGCNTL operation is specified in an instruction, specified CPU and OUTCNTL operations will be performed and control will continue to the next sequential instruction. The use of CONT is optional.

Condition Codes affected: none

Example 1:

```
/* The following code will produce a 50% duty cycle on the F0
   output at a frequency determined by the system clock */

CONT,
   NOP,
   OUT 1 ;     /* output '1' for one cycle */
CONT,
   NOP,
   OUT 0;      /* output '0' for one cycle */
CONT,
   NOP,
   OUT 1;      /* output '1' for one cycle */
JMP E1,
   NOP,
   OUT 0;      /* output '0' during jump cycle */
```

CPI	
Instruction Type:	PROGCNTL
Operation:	Call on Prioritized Interrupt
Syntax:	`[label:] CPI address [,CPU] [,OUTCNTL];`

Description:

The current interrupt status is evaluated. If no interrupt source is active, control branches to the first instruction of a 16-instruction Case block. If at least one interrupt is active, control branches to one of the final eight successive instructions in the block. (The second through eighth instruction in the block are not used.)

The Priority Case Block must start at a location whose address contains zeros in the lower four bits. The address specified must be a numerical value or a symbolic constant whose value is previously defined in this module. The actual location of the Case Block must be resolved at link time, by specification in a link directive.

This instruction is effective only if the INTR bit of the MODE register is clear. Otherwise, the interrupts will be processed by the normal interrupt vector mechanism. CPI is useful in systems where interrupts are not used but in which prioritization of polled inputs is important.

If the selected instruction does not include a jump operation, the next sequential instruction will be executed. Usually, then, the instructions in a Priority Case Block will include jumps to avoid falling through to following cases.

The return address is pushed on the stack.

Condition Codes affected: STKF

See Also: JPI

PACSEL Assembler Reference (Cont.)

CPI (Cont.)

Example 1:

```
/* The following example shows how interrupt
  conditions might be processed by a CPI within a
  polling loop. Interrupts are not enabled anywhere
  in this implementation.*/
  E1CASES equ h'3F0';
.
.
.
  CPI E1CASES,
.
.
.
/* The case block, in general, will be in another
  segment:  */  segment CASE_EXAMPLE;
RET ;          /* arrive here if nothing pending */
JMP ERROR ;    /* error if control comes here */
JMP ERROR;
JMP ERROR;
JMP ERROR ;
JMP ERROR;
JMP ERROR;
JMP ERROR ;
JMP PINT0 ;    /* process int0 */
JMP PINT1 ;    /* process int1 */
JMP PINT2 ;    /* process int2 */
JMP PINT3 ;    /* process int3 */
JMP PINT4 ;    /* process int4 */
JMP PINT5 ;    /* process int5 */
JMP PINT6 ;    /* process int6 */
JMP PINT7 ;    /* process int7 */

PINT0:            /* framework for each routine */
.
.
.
  CLI b'00000001' ;     /* clear the interrupt */
  RET;                  /* since original entry via
                        CPI */
```

The appropriate link directive is:

```
locate CASE_EXAMPLE, h'3F0';
```

PACSEL Assembler Reference (Cont.)

DEC	
Instruction Type:	CPU
Operation:	dest = dest/src - 1

Syntax:	`[label:] [PROGCNTL,] DEC dest/src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] DEC dest src [,OUTCNTL];`

Description:

In the first form, the destination is decremented by one. In the second form, the source is decremented by one and stored in the destination.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>,DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and scr2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

See Also: INC

Example 1:

```
DEC    R1;     /* decrement R1 */
```

Example 2:

```
DEC R1 R3;     /* decrement R3, result to R1 */
```

PACSEL Assembler Reference (Cont.)

DI	
Instruction Type:	PROGCNTL
Operation:	Disable Interrupts in Mask Register
Syntax:	`[label:] DI mask [,CPU] [,OUTCNTL];`

Description:

Use the mask to disable interrupts through the interrupt Mask Register. Bits set to 1 in the mask disable the corresponding interrupt. Zero bits in the mask have no effect.

The Interrupt Mask Register:

MASK7	MASK6	MASK5	MASK4	MASK3	MASK2	MASK1	MASK0

Condition Codes affected: none

See Also: CLI, EI

Example 1:

```
/* This example illustrates the use of DI at the beginning of the
Breakpoint interrupt service routine */

  org h'E';              /* external interrupt 6 vector */
  JMP INT6_SERVICE;

  org h'10';             /* arbitrary */
  DI  b'01000000';       /* disable further breakpoint ints */
INT6_SERVICE:
.
.
.
        RET;             /* return-from-interrupt */
```

Example 2:

```
/* This example illustrates the use of DI as part of system
initialization procedures */
  org h'0';              /* start-up location */
  JMP INIT;

INIT:   org h'10';
  DI h'FF';              /* prevent (mask) all interrupts */
```

PACSEL Assembler Reference (Cont.)

DIV	
Instruction Type:	Macro
Operation:	dest-reg = src-reg1 / src-reg2
Syntax:	`[label:] DIV dest-reg/MSW-dividend LSW-dividend divisor;`

Description:

This Macro divides the 32-bit dividend supplied in the first two arguments by the divisor. The quotient is left in the Q register. The remainder is left in the destination register.

During execution of this code, OUTCNTRL is implied "MAINT".

Condition Codes affected: CY, Z, S. 0

Example 1:

```
DIV R2 R1 R3;    /* R2, R1 is divided by R3 */
```

DSS	
Instruction Type:	PROGCNTL
Operation:	Disable Single Step mode
Syntax:	`[label:] DSS [,CPU] [,OUT CNTL];`

Description:

Disable Single Step mode.

Condition Codes affected: none

See Also: ESS

Example 1:

```
/* This example illustrates a default single-step handler */

  org h'E'          /* external interrupt 0 vector */
JMP SS_SERVICE

  org h'100';       /* arbitrary */
SS-SERVICE
  DSS    ;          /* default single-step handler here */

  RET;              /* return-from-interrupt */

/* Here is the single-step enable */
  ESS;

ADD R1 R2;          /* the first single step occurs two
                    cycles from instruction */
```

PACSEL Assembler Reference (Cont.)

EI	
Instruction Type:	PROGCNTL
Operation:	Enable Interrupts
Syntax:	`[label:] EI mask [cCPU] [,OUTCNTL];`

Description:

Use the mask to enable interrupts through the Interrupt Mask Register. Bits set to "1" in the mask enable the corresponding interrupt. Zero bits in the mask have no effect.

The Interrupt Mask Register:

MASK7	MASK6	MASK5	MASK4	MASK3	MASK2	MASK1	MASK0

If Interrupt 4 is enabled, the device will immediately process an Interrupt 4, since this interrupt is always active.

Condition Codes affected: none

See Also: CLI, DI

Example 1:

```
/* This example illustrates the use of EI at the end of an
external interrupt (INT 2) service routine */

  org h'A';                  /* external interrupt 2 vector */
  JMP INT2_SERVICE

  org h'10';                 /* just past interrupt vectors */
INT2_SERVICE
  DI   b'00000100';          /* disable int2 briefly */
.
.
.
  EI   b'0000100';           /* re-enable int2 briefly */
  RET;                       /* return-from-interrupt */
```

Example 2:

```
/* This example illustrates the use the EI as part of system
initialization procedures */
  org h'0';                  /* start-up location */
  JMP INIT;

  org h'10';
INIT:      EI b'00000100';  /* allow (unmask) all int2 */
```

PACSEL Assembler Reference (Cont.)

ESS	
Instruction Type:	PROGCNTL
Operation:	Enable Single Step mode
Syntax:	`[label:] ESS [,CPU] [,OUTCNTL];`

Description:

Enable Single Step mode. Interrupt 6 will be generated after every subsequent instruction, if it is unmasked.

Condition Codes affected: none

See Also: DSS

Example 1:

```
  org h'E';            /* external interrupt 0 vector */
  JMP SS_SERVICE

  org    h'100';       /* arbitrary */
SS_SERVICE:
  DSS    ;             /* default single-step handler here */

  RET;                 /* return-from-interrupt */
/* Here is the single-step enable */
  ESS;

    ADD R1 R2;         /* the first single step occurs two
                       cycles from instruction */./
```

PACSEL Assembler Reference (Cont.)

INC	
Instruction Type:	CPU
Operation:	dest = dest/src +1

Syntax:	`[label:] [PROGCNTL,] INC dest/src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] INC dest src [,OUTCNTL];`

Description:

In the first form, the destination is incremented by one. In the second form, the source is incremented by one and stored in the destination.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,

a. at least one of the sources must be R0...R31 or Q
b. no more than two distinct registers among R0...R31 may be used and
c. src1 and src2 cannot reference the same member of R0...R31.

3. AF=Arithmetic Flags: Z, CY, S, O.

See Also: DEC

Example 1:

```
INC R1;          /* increment R1 */
```

Example 2:

```
INC R1 R3;       /* increment R3, result to R1 */
```

PACSEL Assembler Reference (Cont.)

INV	
Instruction Type:	CPU
Operation:	dest = NOT dest/src

Syntax	`[label:] [PROGCNTL,] INV dest/src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] INV dest src [,OUTCONTL];`

Description:

In the first form, the destination is bit-inverted. In the second form, the source is bit-inverted and stored in the destination.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF,DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q. AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 or Q
   and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

Example 1:

```
INV R1;        /* R1 <-- NOT (R1) */
```

Example 2:

```
INV R1 R3;     /* R1 <-- NOT (R3) */
```

PACSEL Assembler Reference (Cont.)

JCASE	
Instruction Type:	PROGCNTL
Operation:	Jump to Case Block
Syntax:	`[label:] JCASE casegroup address [,CPU] [,OUTCNTL];`

Description

JCASE performs an unconditional CASE selection using the Case Block at the address specified. Case selection is based on the 4 bits of the Case Group specified:

'CGO' specifies [CC3, CC2, CC1, CC0]
'CG1' specifies [CC7, CC6, CC5, CC4]
'CG2' specifies [INTR, BCZ, FIOR, FICD]
'CG3' specifies [Z, O, S, CY]

One of the 16 successive instructions starting at the Case Block label will be executed. If the instruction does not include a jump operation, the next sequential instruction will be executed. Ordinarily, then, the instructions in a Case block will include jumps.

The Case Block must start at a location whose address contains zeros in the lower four bits. The address specified must be a numerical value or a symbolic constant whose value is previously defined in this module. The actual location of the Case Block must be resolved at link time, by specification in a link directive.

Condition Codes affected: none

Example 1:

```
/* This example outputs the binary value of Case Group 1 (CG1) on
the F outputs for one cycle, bracketed by the value -1. */

  E1C equ h'200';
  JCASE CG1 E1C,
    NOP ,
    OUT h'FFFF';       /* output -1 during jump cycle */

/* The case block, in general, will be in another segment:
   */segment CASE_EXAMPLE;
  E1C:   JMP ECEND, NOP, OUT 0 ;
  JMP ECEND, NOP, OUT 1 ;
  JMP ECEND, NOP, OUT 2 ;
  JMP ECEND, NOP, OUT 3 ;
  JMP ECEND, NOP, OUT 5 ;
  JMP ECEND, NOP, OUT 6 ;
  JMP ECEND, NOP, OUT 7 ;
  JMP ECEND, NOP, OUT 8 ;
  JMP ECEND, NOP, OUT 9 ;
  JMP ECEND, NOP, OUT 10 ;
  JMP ECEND, NOP, OUT 11 ;
  JMP ECEND, NOP, OUT 12 ;
  JMP ECEND, NOP, OUT 13 ;
  JMP ECEND, NOP, OUT 14 ;
  JMP ECEND, NOP, OUT 15 ;
ECEND:
  CONT, NOP, OUT h'FFFF';      /* all cases end up here */
```

The appropriate link directive is: `locate CASE_EXAMPLE, h'200';`

PACSEL Assembler Reference (Cont.)

JMP	
Instruction Type:	PROGCNTL
Operation:	Jump to target

Syntax:	`[label:] JMP label2 [,CPU] [,OUTCNTL];`
	or
	`[label:] JMP LCPTR [,CPU] [,OUTCNTL];`
	or
	`[label:] JMP FIFO [,CPU] [,OUTCNTL];`
	or
	`[label:] JMP TOS [,CPU] [,OUTCNTL];`

Description:

JMP is an unconditional branch to the target address. The target, in the first form, is a program label.

The keyword LCPTR may be substituted, in which case the current value of LC (Loop Counter) is used an as execution pointer. The ten-bit LC value is loaded into the program counter. In this case, the LCPTR register must be explicitly loaded (see LDLC and LDLCD) prior to this operation.

In the third form the keyword FIFO may be substituted, in which case the top of the FIFO is used as the jump target.

In the fourth form the keyword TOS may be substituted, in which case the top of stack is used as the jump target without popping the stack.

Condition Codes affected: FIFO flags (Form 3)

Example 1:

```
JMP E1b,
 NOP,
 OUT h'A5A5';      /* output test pattern during JMP cycle */
E1b:
```

Example 2:

```
LDLC 37;           /* jump through pointer */
MP LCPTR;
E2b:
```

Example 3:

```
JMP FIFO;          /* main program for host-driven mode */
JMP E3;
```

PACSEL Assembler Reference (Cont.)

JMPC	
Instruction Type:	PROGCNTL
Operation:	Jump if Condition Code TRUE
Assembler Syntax:	`[label:] JMPC cc label2 [,CPU] [,OUT CNTL];`

Description:

The condition code specified by "cc" is evaluated. If it is TRUE, then control branches to the specified label.

The Condition Codes are:

INTR	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Condition Codes affected: none

Example 1:

```
JMPC Z E1b,    /* jump on condition code 'zero' */
NOP,
OUT h'45';     /* output ASCII E during jump cycle   */
.
.
.
E1b:    JMPC CY E1c;     /* jump on carry, use default CPU
                            and OUTCNTL */
```

JMPNC	
Instruction Type:	PROGCNTL
Operation:	Jump if Condition Code not TRUE
Syntax:	`[label:] JMPNC cc label2 [,CPU] [,OUTCNTL];`

Description:

The condition code specified by "cc" is evaluated. If it is FALSE, then the program branches to the specified dest.

The Condition Codes are:

INTR	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Condition Codes affected: none

Example 1:

```
JMPNC Z E1b,              /* jump on condition code 'nonzero' */
NOP,
OUT h'45';                /* output ASCII E during jump cycle */
.
.
.
E1b:    JMPNC CY E1c;     /* jump on not carry, default CPU,
                             OUTCNTL */
```

PACSEL Assembler Reference (Cont.)

JPI	
Instruction Type:	PROGCNTL
Operation:	Jump on Prioritized Interrupt
Syntax:	`[label:] JPI address [,CPU] [,OUTCNTL];`

Description:

The current interrupt status is evaluated. If no interrupt source is active, then the program branches to the first instruction of a 16-instruction Priority Case Block. If at least one interrupt is active, the program branches to one of the final eight successive instructions in the block. (The second through eighth instructions in the block are not used.)

The Priority Case block must start at a location whose address contains zeros in the lower four bits. The address specified must be a numerical value or a symbolic constant whose value is previously defined in this module. The actual location of the Case Block must be resolved at link time, by specification in a link directive.

This instruction is effective only if the INTR bit of the MODE register is clear. Otherwise, the interrupts will be processed by the normal vector mechanism. JPI is useful in systems where interrupts are not used but in which prioritization of polled inputs is important.

If the selected instruction does not include a jump operation, the next sequential instruction will be executed. Usually, then the instructions in a Priority Case Block will include jumps to avoid falling through to following cases.

Condition Codes affected: none

Example 1:

```
/* The following example shows how interrupt conditions might be
processed by a JPI within a polling loop. Interrupts are not
enabled anywhere in this implementation. This approach may be
helpful in a system in which response time is relatively
unimportant but stack space is extremely tight. The stack is not
used at all in this code. */

E1CASES equ h'140';
JPI E1CASES,

/* The case block, in general, will be in another segment: */
segment CASE_EXAMPLE;
 JMP E2 ;           /* arrive here if nothing pending */
 JMP ERROR ;        /* error if control comes here */
 JMP ERROR ;
 JMP ERROR ;
 JMP ERROR ;
 JMP ERROR ;
 JMP ERROR ;
 JMP ERROR ;
 JMP PINT0 ;        /* process int0 */
 JMP PINT1 ;        /* process int1 */
 JMP PINT2 ;        /* process int2 */
 JMP PINT3 ;        /* process int3 */
 JMP PINT4 ;        /* process int4 */
 JMP PINT5 ;        /* process int5 */
 JMP PINT6 ;        /* process int6 */
 JMP PINT7 ;        /* process int7 */
PINT0:              /* framework for each routine */

 CLI b'00000001';   /* clear the interrupt */
 JMP E2;            /* since original entry via JPI */
```

The appropriate link directive is: `locate CASE_EXAMPLE, h'140';`

PACSEL Assembler Reference (Cont.)

LDBP	
Instruction Type:	PROGCNTL
Operation:	Load Breakpoint Register
Syntax:	[label:] LDBP constant/label [,CPU] [,OUTCNTL];

Description:

The specified constant/label becomes the new breakpoint register value. The maximum allowed value is 1023 decimal.

Condition Codes affected: None

Example 1:

```
   LDBP 300          /* set Breakpoint to label value */
    SUB R2 R1,
    OUT h'FFFF';
 .
 .
 .
E1b:
```

LDBPD	
Instruction Type:	PROGCNTL
Operation:	Load Breakpoint Register from CPU result
Syntax:	[label:] LDBPD, CPU [,OUTCNTL];

Description:

The required CPU operation supplies the numerical value to be loaded into the BreakPoint register.

The maximum allowed value is 1023 decimal.

A CPU operation should be present. If the CPU operation is omitted, the breakpoint will be set to zero.

Condition Codes affected: see CPU instruction used

Example 1:

```
   LDBPD,            /* set Breakpoint to R1+R2 */
    ADD R1 R2,       /* source of value*/
    OUT h'FFFF';
 .
 .
 .
E2b:
```

PACSEL Assembler Reference (Cont.)

LDLC	
Instruction Type:	PROGCNTL
Operation:	Load Loop Counter
Syntax:	`[label:] LDLC constant/label [,CPU] [,OUTCNTL];`

Description:

The constant or the address given by the label is placed into the LC (Loop Counter) register. The maximum allowed value is 1023 decimal. This instruction supports loops and the CALL LCPTR and JMP LCPTR instructions.

Condition Codes affected: none

Example 1:

```
/* This example shows how a constant loop count is used */
  LDLC 35;          /* want loop to execute 36 times */
E1b:                /* loop body */
.
.
.
  LOOPNZ E1b;       /* conclude loop */
```

PACSEL Assembler Reference (Cont.)

LDLCD	
Instruction Type:	PROGCNTL
Operation:	Load Loop Counter with CPU result
Syntax:	`[label:] LDLCD, CPU [,OUTCNTL];`

Description:

The value computed by the obligatory CPU operation is placed into the LC (Loop Counter) register. The maximum allowed value is 1023. This instruction supports loops and the CALL LCPTR and JMP LCPTR instructions.

Condition Codes affected: see CPU operation used.

Example 1:

```
/* This example shows an alternative way of using the same
constant. */

  MOV R2 N;        /* want loop to execute N+1 times */
  LDLCD,           /* find (R1-R2) and load in same instruction */
   SUB R1 R2;
  E2b:             /* loop body */
.
.
.
 LOOPNZ E2b;       /* conclude loop*/
```

Example 2:

```
/* This example shows how to use a value computed at run time as
the loop count. The null case check is almost always advisable.
Note that if R1 contains 1 at entry, the loop will properly
execute exactly 1 time. */

                   /* R1 contains the loop count at entry */
  CMP R1 0;        /* check null case */
  JMPC Z E3c;      /* jump past if N=0 */
  LDLCD,           /* find (N-1) and load in same instruction */
   SUB R1 N;
E3b:               /* loop body */
.
.
.
   LOOPNZ E3b;     /* conclude loop */
E3c:
```

PACSEL Assembler Reference (Cont.)

LOOPNZ	
Instruction Type:	PROGCNTL
Operation:	Loop to label if Loop Counter nonzero
Syntax:	`[label:] LOOPNZ label [,CPU] [,OUTCNTL];`

Description:

The LC (Loop Counter) register is examined, and if it is nonzero, the program branches to 'label'. Otherwise, execution continues at the next sequential instruction. After the test, LC is decreased by one.

The LOOPNZ instruction may be used to conclude the body of an unnested or nested loop. In the case of a nested loop, the LC register value of the next outer loop should normally be restored immediately by executing a POPLC instruction after LOOPNZ.

Note: The LOOPNZ instruction tests the loop count first, then decrements it. As a result, loops will always be performed (Loop Count + 1) times.

Condition Codes affected: none

Example 1:

```
    LDLC 44;              /* want loop to execute 45 times */
    MOV R15 0;            /* initial value for R15 */
E1b:    ADD R15 1;        /* loop body: increment R14 */
                          /* loop body ... */
  .
  .
  .
    LOOPNZ E1b;           /* conclude loop */
                          /* R15=45 */
```

PACSEL Assembler Reference (Cont.)

MAINT	
Instruction Type:	OUTCNTL
Operation:	Maintain previously defined output value
Syntax:	`[label:] [PROGCNTL,] [CPU,] MAINT;`

Description:

MAINT is the default OUTCNTL operation, that is, if MAINT is specified, the assembler uses a default OUT value equal to that most previously specified. (If no OUT operations occur in a module, the assembler uses 0). As a result the default is maintained in order of assembly, not order of execution.

The use of MAINT is optional; the result is the same if no OUTCNTL operation is specified.

Condition Codes affected: none

See Also: OUT

Example 1:

```
CONT ,                  /* Full instruction form, do-nothing */
 NOP ,
 MAINT;
```

Example 2:

```
CONT, NOP, MAINT;       /* compact full form do-nothing */
```

Example 3:

```
MAINT;                  /* simplified do-nothing */
```

Example 4:

```
OUT 1;                  /* output 1 for one cycle, and */
MAINT;
OUT 2;                  /* output 2 for two cycles, and */
MAINT;
MAINT;
OUT 3;                  /* output 3 for three cycles */
MAINT;
MAINT;
MAINT;
```

Example 5:

```
/* The following two examples produce different results on the
basis of an instruction that is never executed, because of the
way assembly order controls the OUTCNTL default value. */

  JMP E5b;              /* produce 50% duty cycle on F0 line */
  OUT   1;              /* never executed, but sets default */
E5b:    MAINT;          /* F outputs set to 1 here */
  JMP E5b , OUT 0;      /* F outputs set to 0 here */
```

Example 6:

```
E6:     JMP E6b;        /* produce constant low of F0 line  */
   OUT   0;             /* never executed, but sets default */
E6b:    MAINT;          /* F outputs set to 0 here */
   JMP E6b , OUT 0;     /* F outputs set to 0 here */
```

PACSEL Assembler Reference (Cont.)

MOV	
Instruction Type:	CPU
Operation:	Move source to destination
Syntax:	`[label:] [PROGCNTL,] MOV dest src [,OUTCNTL];`

Description:

MOV copies the source to the destination.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

Condition Codes affected: see table

Example 1:

```
MOV             R1 33;          /* immediate */
```

Example 2:

```
MOV             R1 R2;          /* register to register */
```

Example 3:

```
MOV             R1 BC;
```

Example 4:

```
MOV BC R1;
```

Example 5:

```
MOV             R1 AIR
MOV             BC R1;
```

PACSEL Assembler Reference (Cont.)

MUL	
Instruction Type:	Macro
Operation:	dest-reg = src-reg1 * src-reg2
Syntax:	`[label:] MUL dest-reg src-reg1 src-reg2;`

Description:

This Macro multiplies the values of source register 1 and source register 2 and places the low-order result in source register 1 and in high-order the destination register.

The generated code will require 22 cycles and occupy 7 EPROM location. During execution of this code, OUTCNTRL is implied "MAINT".

Condition Codes affected: CY, Z, S, O

Example 1:

```
MUL R2 R1 R1;       /* R1 squared to R2 and Q */
```

PACSEL Assembler Reference (Cont.)

NEG	
Instruction Type:	CPU
Operation:	dest = 2's complement dest/src

Syntax:	`[label:] [PROGCNTL,] NEG dest/SRC [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] NEG dest src [,OUTCNTL];`

Description:

In the first form, the destination is replaced by its two's complement. In the second form, the source is two's complemented and stored in the destination.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

Example 1:

```
NEG   R1;      /* R1 <- 2's comp (R1) */
```

Example 2:

```
NEG R1 R3;     /* R1 <- 2's comp (R3) */
```

PACSEL Assembler Reference (Cont.)

NOP	
Instruction Type:	CPU
Operation:	No operation
Syntax:	`[label:] [PROGCNTL,] NOP [,OUTCNTL];`

Description:

NOP (No Operation) is the default CPU operation, that is, if no CPU operation is specified in an instruction, PAC1000 Peripheral Controller performs the default CPU Operation:

AND 0 0;

which sets the Z flag and sets the carry flag. Using NOP is optional.

Condition Codes affected: Z, CY

Example 1:

```
CONT ,                  /* Full instruction form, do-nothing */
  NOP ,
  MAINT;
```

Example 2:

```
CONT, NOP, MAINT;       /* compact full form do-nothing */
```

Example 3:

```
NOP;                    /* simplified do-nothing */
```

Example 4:

```
JMP E4,                 /* CPU only do-nothing */
  NOP ,
  OUT h'FFFF';
```

PACSEL Assembler Reference (Cont.)

OR	
Instruction Type:	CPU
Operation:	dest = src1 OR src2

Syntax:	`[label:] [PROGCNTL,] OR dest/src src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] OR dest src1 src2 [,OUTCNTL];`

Description:

In the first form, this instruction ORs the source and the destination and places the result in the destination.

In the second form, two registers are ORed and the result stored in the destination. One of the sources may be the same as the destination.

The sources and the destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

Example 1:

```
OR R2 R1;       /* CPU operation alone */
```

Example 2:

```
JMP E2B ,
OR R3 R31;      /I CPU operation combined with PROGCONTL op */

E2B:
```

Example 3:

```
JMP E2B ,
 OR R3 R31 ,    /* CPU operation combined with PROGCONTL op */
 OUT 3;         /* and OUTCNTL operation */.

E3B:
```

Example 4:

```
OR BC R1;       /* to Block Counter register */
```

PACSEL Assembler Reference (Cont.)

OUT	
Instruction Type:	OUTCNTL
Operation:	Output control (F) value
Syntax:	`Label:[PROGCNTL,] [CPU,] OUT <constant | expression>;`

Description:

OUT directs the device to place the specified constant value on the Control (F) outputs.

The value specified is placed on the 16 bit F (user) output lines during the instruction cycle which includes this operation. The constant is usually specified directly as a hex value in the range

0 <= constant <= h'FFFF'

or the value may be evaluated as an expression. The value is determined at assembly time; there is no way of computing an OUT value at run time.

The value may be formed from 2 up to 16 constants, or values that evaluate to constants. In this case, all the values are bitwise OR'd to produce the output value. The OR operator in this case is implied.

The value can also be an expression. The expression is evaluated and the evaluated result is the output value. All the arithmetic and logic operators which can be used with the 'set' directive can also be used with the OUT expression.

When an OUTCNTL operation is omitted from an instruction, the assembler will provide a default OUT value equal to that most recently specified. (If no OUT operations occur in a module, the assembler will use 0.) Thus OUT defaults are maintained in order of assembly, not order of execution.

Condition Codes affected: none

See Also: MAINT

Example 1:

```
/* The following examples will both produce a 75% duty cycle on
the F0 output at a frequency determined by the system clock */
  CONT,        /* full version */
   NOP,
OUT 1;         /* output '1' for one cycle */
CONT;          /* output '1' for one cycle */
CONT;          /* output '1' for one cycle */
JMP E1 ,
NOP,
OUT 0;         /* output '0' during jump cycle */
```

Example 2:

```
XXX equ 45;    /* define a constant */
OUT XXX;       /* minimal version */
;
;
JMP EXAMPLE2 , OUT 0 ;
```

PACSEL Assembler Reference (Cont.)

OUT (Cont.)

Example 3:

```
ZZZ equ 45;              /* define a constant */
YYY equ 3;               /* define second constant */
OUT ZZZ YYY;             /* minimal version */
;
;
JMP E2 , OUT 0;
```

Example 4:

```
ZZZ equ 45;              /* define a constant */
XXX equ 10;              /* define second constant */
OUT ZZZ : XXX;           /* OR of XXX and ZZZ */
OUT ZZZ & 6;             /* and of XXX and ZZZ */
OUT 10;
OUT ZZZ;
OUT XXX ; ZZZ & 6;       /* OR of XXX and ZZZ and with 6 */
```

PLDLC	
Instruction Type:	PROGCNTL
Operation:	Push and Load Loop Counter
Syntax:	`[label:] PDLC constant/label [,CPU] [,OUTCNTL];`

3

Description:

The current LC (Loop Counter) value is pushed on the stack, then the constant or address value is placed into the LC register. The Loop Counter value saved by PLDLC must be explicitly restored before resuming the enclosing loop.

This instruction may be used to initiate a nested loop. The end of the loop will be defined by a LOOPNZ instruction; the label specified as the LOOPNZ operand is the beginning of the loop. In other words, the PLDLC operation may precede the beginning of the nested loop body. Use LDLC to initiate an unnested loop.

Note: The LOOPNZ instruction tests the loop count first, then decrements it. As a result, loops will always be performed (Loop Count + 1) times.

Condition Codes affected: STKF

Example 1:

```
/* The following example shows how to nest loops. These loops
have constant Loop Count values. */

   LDLC n;               /* load outer loop count */
E1b:                     /* 1st instruction of outer loop */
.
.
.
  PLDLC m;               /* save LC and load inner loop count */
E1c:    ADD R1 10;       /* 1st instruction of inner loop */
  LOOPNZ E1c;            /* end of inner loop */
  POPLC;                 /* restore outer loop count */
  LOOPNZ E1b;            /* end of outer loop */
                         /* R1 = R1 + ( (N+1) * (10*(M=1) ) */
```

PACSEL Assembler Reference (Cont.)

PLDLCD	
Instruction Type:	PROGCNTL
Operation:	Push and Load Loop Counter from CPU result
Syntax:	`[label:]PLDLCD , CPU [,OUTCNTL];`

Description:

The current LC (Loop Counter) value is pushed on the stack, then the value computed in the CPU instruction is used. The Loop Counter value saved by PLDLC must be explicitly restored before resuming the enclosing loop.

This instruction may be used to initiate a nested loop. The end of the loop will be defined by a LOOPNZ instruction; the label specified as the LOOPNZ operand is the beginning of the loop. In other words, the PLDLCD operation may precede the beginning of the nested loop body. Use LDLC to initiate an unnested loop.

Note: The LOOPNZ instruction tests the loop count first, then decrements it. As a result, loops will always be performed (Loop Count + 1) times.

A CPU operation must be present if the second form is used. If the CPU operation is omitted, the loop count will be set to zero and the loop will execute once.

Condition Codes affected: STKF; also see CPU operation used

Example 1:

```
/* The following example shows how a nested loop can have a
computed loop count. The value in R2 is assumed to be nonzero.
Adding R1 to R2 is a simple way of providing a CPU-computed value
for this form of the instruction. */
  LDLC n;              /* load outer loop count */
E1b:                   /* 1st instruction of outer loop */
.
.
.
  PLDLCD,              /* save LC, and use R1+R2 as loopcount */
   ADD R2 R1;          /* ... all in same instruction */
E1c:    ADD R1 10;     /* 1st instruction of inner loop */
  LOOPNZ E1c;          /* end of inner loop */
  POPLC;               /* restore outer loop count */
  LOOPNZ E1b;          /* end of outer loop */
                       /* R1 = R1+( (N+1) * (10*(R2+R1+1) )
```

PACSEL Assembler Reference (Cont.)

POP	
Instruction Type:	PROGCNTL
Operation:	POP stack and discard
Syntax:	`[label:] POP [,CPU] [,OUTCNTL];`

Description:

The value on the top of the stack is popped and discarded.

This operation should be used with caution, since the stack generally contains subroutine return addresses or loop counts which may be expected to be on the stack by a return or loop conclusion instruction.

Condition Codes affected: STKF

Example 1:

```
/* This example shows how POP is used to discard the Loop Counter
value on the stack when a nested loop is aborted. */
  LDLC n;              /* load outer loop count */
E1b:                   /* 1st instruction of outer loop */
.
.
.
  PLDLC m;             /* save LC and load inner loop count */
E1c:                   /* 1st instruction of inner loop */
.
.
.
  JMPC STKF PANIC;     /* jump if things are bad */
  LOOPNZ E1c;          /* end of inner loop */
  POPLC;               /* restore outer loop count */
  LOOPNZ E1b;          /* end of outer loop */

PANIC:
  POP;                 /* restore stack by discarding count */
```

Example 2:

```
/* This example shows how POP can be used to discard return
addresses when an error condition is located inside nested
subroutines */
  CALL E2b;            /* outermost level */
pt_a:
.
.
.
  RET;
E2b:    CALL E2C;      /* next nested level */
pt_b:
.
.
.
  RET;
E2C:    CALL E2D;      /* next nested level */
pt_c:
.
.
.
  RET
E2D:                   /* deepest level */
.
.
.
  JMPNC STKF E2E;      /* skip if not stackfull, else climb out */
  POP   ;              /* discard return to pt_c */
  POP   ;              /* discard return to pt_b */
  RET   ;              /* return to pt_a */
E2E:
```

PACSEL Assembler Reference (Cont.)

POPLC	
Instruction Type:	PROGCNTL
Operation:	Pop Stack Loop Counter
Syntax:	`[label:] POPLC [,CPU] [,OUT CNTL];`

Description:

The contents of the LC (Loop Counter) register are popped from the stack. This instruction is used to conclude nested loops.

Condition Codes affected: STKF

Example 1:

```
/* This example shows how POPLC is used to restore the Loop
counter value following the conclusion of a nested loop. */

   LDLC n;            /* load outer loop count */
E1b:                  /* 1st instruction of outer loop */
.
.
.
   PDLC m;            /* save LC and load inner loop count */
E1c:                  /* 1st instruction of inner loop */
.
.
.
   LOOPNZ E1c:        /* end of inner loop */
   POPLC;             /* restore outer loop count */
   LOOPNZ E1b;        /* end of outer loop */
```

Example 2:

```
/* The following example shows how POPLC may be used with PUSHLC
to save and restore the Loop Counter value where a CALL LCPTR is
used within the loop. */

   LDLC n;            /* load loop count */
   MOV R21 0;         /* initialize register */
E2b:                  /* 1st instruction of loop */
.
.
.
   PUSHLC;            /* save LC */
   LDLCD ,            /* load call vector */
    ADD R20 R21;      /* already assumed to be in R20 */
   CALL LCPTR;        /* perform call */
   POPLC;             /* restore loop count */
   LOOPNZ E2b;        /* end of loop */
```

PACSEL Assembler Reference (Cont.)

PUSHLC	
Instruction Type:	PROGCNTL
Operation:	Push Loop Counter
Syntax:	`[label:] PUSHLC [,CPU] [,OUTCNTL];`

Description:

The contents of the LC (Loop Counter) register are pushed on the stack. This instruction is used at the beginning of a nested loop.

Condition Codes affected: STKF

Example 1:

```
/* This example show show PUSHLC is used to save the Loop Counter
value before the start of a nested loop. The PUSHLC+LDLC sequence
can more economically be replaced by PLDLC in many cases. */

   LDLC n;              /* load outer loop count */
E1b:                    /* 1st instruction of outer loop */
.
.
.
   PUSHLC;              /* save LC */
   LDLC m;              /* load inner loop count */
E1c:                    /* 1st instruction of inner loop */
.
.
.
   LOOPNZ E1c;          /* end of inner loop */
   POPLC;               /* restore outer loop count */
   LOOPNZ E1b;          /* end of outer loop */
```

Example 2:

```
/* The following example shows how PUSHLC may be used with POPLC
to save and restore the Loop Counter value where a CALL LCPTR is
used within the loop. */

   LDLC n;              /* load loop count */
E2b:                    /* 1st instruction of loop */
.
.
.
   PUSHLC;              /* save LC */
   LDLCD ,              /* load call vector */
    MOV R20 R20;        /* already assumed to be in R20 */
   CALL LCPTR;          /* perform call */
   POPLC;               /* restore loop count */
   LOOPNZ E2b;          /* end of loop */
```

PACSEL Assembler Reference (Cont.)

RC	
Instruction Type:	PROGCNTL
Operation:	Return Conditionally

Syntax:	`[label:] RC cc [,CPU] [OUTCNTL];`
	or
	`[label:] RC cc label2 [,CPU] [,OUTCNTL];`

Description:

The next instruction address is taken from the stack if the condition is TRUE. In the second form, control branches to "label2" if the condition is true. The stack is popped in any case. If the condition is false, this instruction has no effect.

The Condition Codes are:

INTR	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Condition Codes affected: STKF

Example 1:

```
    CALL E1b,
    NOP,
    OUT h'46';          /* output ASCII F during CALL cycle */
  .
  .
  .
  E1b:    RC Z ;        /* null subroutine for Zero condition */
```

Example 2:

```
  /* The following example shows the framework of a int 7 service
  routine that ignores the ACO (Address Counter Ones) condition. */
    org h'F';
    JMP INT7_SERVICE;    /* install service vector */
    org h'100';          /* arbitrary address */
  INT7_SERVICE:
  .
  .
  .
    RC ACO;              /* if int 7 caused by ACO, leave */
  .
  .
  .
    RET;
```

PACSEL Assembler Reference (Cont.)

RDFIFO	
Instruction Type:	CPU
Operation:	Read FIFO data to CPU destination
Syntax:	`[label:] [PROGCNTL,] RDFIFO [,OUTCNTL];`

Description:

The 16-bit data on the FIFO is moved to the CPU register whose address is specified by the address stored with the data.

If the FIFO is empty, the previous top value will be read. If the item is a command, then the FIFO exception condition will occur.

Condition Codes affected: CY, Z, S, O, FIIR, FIOR, FIXP, FICD

Example 1:

```
RDFIFO;      /* data presumed to have R3 target  */
DEC R3;      /* decrement data value */
```

RESTART	
Instruction Type:	PROGCNTL
Operation:	Restart by jump to location 0
Syntax:	`[label:] RESTART [,CPU] [,OUTCNTL];`

Description:

RESTART is an unconditional jump to the first program step, i.e., EPROM location zero. It is logically equivalent to a jump to location zero. Only stack pointer initialization is performed by this instruction.

Condition Codes affected: CY, Z, S, ACO, BCZ, FIIR, FIOR, FIXP, DOR

Example 1:

```
   JMPC STKF E1b:     /* no CPU, OUTCNTL specified */
.
.
.
E1b:  RESTART         /* use with caution */
```

Example 2:

```
   JMPC STKF E2b;
E2b:    RESTART, NOP, OUT h'FFFF' ;   /* output reset marker */
```

PACSEL Assembler Reference (Cont.)

RET	
Instruction Type:	PROGCNTL
Operation:	Return from subroutine

Syntax:	`[label:] RET [,CPU] [,OUTCNTL];`
	or
	`[label:] RET label2 [,CPU] [,OUTCNTL];`

Description:

RET is an unconditional subroutine return. In the first form, the next instruction address is taken from the stack.

In the second form, control branches to the specified label and the stack is popped.

Condition Codes affected: STKF

Example 1:

```
    CALL elb,
    NOP,
    OUT h'46';          /* output ASCII F during CALL cycle */
Elb:   RET;             /* null subroutine */
```

Example 2:

```
/* This example illustrates the use of RET to conclude an
interrupt service routine */

    org h'8'            /* external interrupt 0 vector */
    JMP INT0_SERVICE;
    ORG H'100'          /* arbitrary */
INT0_SERVICE:

    RET;                /* return-from-interrupt */
```

Example 3:

```
RET Elb;                /* go to address Elb and pop the stack */
```

PACSEL Assembler Reference (Cont.)

RNC	
Instruction Type:	PROGCNTL
Operation:	Return from subroutine if condition NOT TRUE

Syntax:	`[label:] RNC cc [,CPU] [,OUTCNTL];`
	or
	`[label:] RNC cc label2 [,CPU] [,OUTCNTL];`

Description:

RNC is a conditional subroutine return. In the first form, the next instruction address is taken from the stack if the condition is FALSE.

In the second form, if the condition is FALSE, control branches to 'label2' and the stack is popped.

The Condition Codes are:

INTR	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Condition Codes affected: STKF

Example 1:

```
    CALL Elb,
      NOP,
      OUT h'46';          /* output ASCII F during CALL cycle */

Elb:      RNC     Z;      /* null subroutine */
```

Example 2:

```
/* The following example shows the framework of an int 7 service
routine that processes only the STKF (Stack Full) condition. */

    org h'F';
    JMP INNT7_SERVICE     /* install service vector */

    org h'100;            /* arbitrary address */
INT7_SERVICE:

    RNC STKF;             /* if int 7 not caused by STKF, leave */
    JMP REINITIALIZE      /* else do appropriate fix */
```

PACSEL Assembler Reference (Cont.)

RSTCON	
Instruction Type:	PROGCNTL
Operation:	Reset bits in the control register
Syntax:	`[label:] RSTCON mask [,CPU] [,OUTCNTL];`

Description:

Use the mask to reset bits in the Control Register. Bits set to 1 in the mask reset the corresponding register bits. Zero bits have no effect.

The Control Register:

ASEL	AIREN	DIREN	HDSEL1	HDSEL0	ADOE	HADOE	HDOE	BCEN	ACEN

Ordinarily, the mask will be most conveniently given in hexadecimal notation.

Condition Codes affected: none

See Also: SETCON

Example 1:

```
RSTCON h'3FF';          /* Clear ALL Control Register Bits */
```

Example 2:

```
RSTCON b'0000000011';  /* clear only BCEN and ACEN */
```

RSTIO	
Instruction Type:	PROGCNTL
Operation:	Reset bits in the I/O Configuration Register
Syntax:	`[label:] RSTIO mask [,CPU] [,OUTCNTL];`

Description:

Use the mask to reset bits in the I/O Register. Bits set to 1 in the mask reset the corresponding register bits. Zero bits have no effect.

The I/O Configuration Register:

IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0

Ordinarily, the mask will be most conveniently given in hexadecimal notation. The mask value must be between 0 and FFh.

Condition Codes affected: none

See Also: SETIO

Example 1:

```
RSTIO  h'FF';           /* Clear I/O Configuration Register */
```

PACSEL Assembler Reference (Cont.)

RSTMODE	
Instruction Type:	PROGCNTL
Operation:	Reset bits in the Mode Register
Syntax:	`[label:] RSTMODE mask [,CPU] [,OUTCNTL];`

Description:

Use the mask to reset bits in the Mode Register. Bits set to 1 in the mask reset the corresponding register bits. Zero bits have no effect.

The Mode Register:

FIRST	FIIR	ADOEI	HADOE	HDOE	SIO	ACEN	BCEN	N/A	INTR

Ordinarily, the mask will be most conveniently given in a hexadecimal notation.

Condition Codes affected: none

See Also: SETMODE

Example 1:

```
RSTMODE h'FF';              /* Clear Mode Register */
```

Example 2:

```
SETMODE b'1000000000';      /* reset FIFO, then... */
RSTMODE b'1000000000';      /* immediately clear reset */
```

PACSEL Assembler Reference (Cont.)

SBC	
Instruction Type:	CPU
Operation:	dest = src1 - src2 - CY

Syntax:	`[label:] [PROGCNTL,] SBC dest/src src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] SBC dest src1 src2 [,OUTCNTL];`

Description:

In the first form, this instruction subtracts two values. The state of the carry bit, from previous subtractions, is included in the computation. If CY is not set, the difference is decreased by 1 during the subtract operation. In the second form, two registers are subtracted and the result placed in the destination register.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:
1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

Example 1:

```
SBC R1 R1;          /* zero R1 and add CY */
```

Example 2:

```
MOV R31 h'0';       /* load immediate value */
SUB R31 h'1';       /* subtract immediate */
SBC R31 h'0';       /* R31 now is 0 */
```

Example 3:

```
SBC BC R23;         /* BC = BC - R23, accounting for CY */
```

Example 4:

```
SBC BC R23 R24;     /* BC = R23 - R24, accounting for CY */
```

Example 5:

```
CONT ,
 SBC BC R23 R24 , /* full instruction format */
 OUT h'A5A5'  ;
```

PACSEL Assembler Reference (Cont.)

SETCON	
Instruction Type:	PROGCNTL
Operation:	Set bits in the Control Register
Syntax:	`[label:] SETCON mask [,CPU] [,OUTCNTL];`

Description:

Use the mask to set bits in the Control Register. Bits set to 1 in the mask set the corresponding register bits. Zero bits have no effect.

The Control Register:

ASEL	AIREN	DIREN	HDSEL1	HDSEL0	ADOE	HADOE	HDOE	BCEN	ACEN

Ordinarily the mask will be most conveniently given in hexadecimal notation.

Condition Codes affected: none

See Also: RSTCON

Example 1:

```
SETCON h'3FF';          /* Set ALL Control Register bits */
```

Example 2:

```
SETCON b'0000000011';   /* set only BCEN and ACEN */
```


SETIO	
Instruction Type:	PROGCNTL
Operation:	Set bits in the I/O Configuration Register
Syntax:	`[label:] SETIO mask [,CPU] [,OUTCNTL];`

Description:

Use the mask to set bits in the I/O Register. Bits set to 1 in the mask set the corresponding register bits. Zero bits have no effect.

The I/O Configuration Register:

IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0

Ordinarily, the mask will be most conveniently given in the hexadecimal notation. The mask value must be between 0 and FFh.

Condition Codes affected: none

See Also: RSTIO

Example 1:

```
SETIO h'FF';        /* Set all I/O Port lines to output */
```

PACSEL Assembler Reference (Cont.)

SETMODE	
Instruction Type:	PROGCNTL
Operation:	Set bits in the Mode Register
Syntax:	`[label:] SETMODE mask [,CPU] [,OUTCNTL];`

Description:

Use the mask to set bits in the Mode Register. Bits set to 1 in the mask set the corresponding register bits. Zero bits have no effect.

The Mode Register:

FIRST	FIIR	ADOEI	HADOE	HDOE	SIO	ACEN	BCEN	N/A	INTR

Ordinarily, the mask will be most conveniently given in hexadecimal notation.

Condition Codes affected: none

See Also: RSTMODE

Example 1:

```
SETMODE h'1';              /* Set Interrupt Mode */
```

Example 2:

```
SETMODE b'1000000000';     /* reset FIFO, then... */
RSTMODE b'1000000000';     /* immediately clear reset */
```

SHLR	
Instruction Type:	CPU
Operation:	Shift Left Register
Syntax:	`[label:] [PROGCNTL,] SHLR reg src [,OUTCNTL];`

Description:

This instruction shifts the selected register (R0 thru R31) left one bit. The data shifted in depends on the source operand:

Z	– the Zero bit flag
CY	– the Carry bit flag
S	– the Sign bit flag
0	– a binary '0'
1	– a binary '1'
RMSB	– the Most Significant Bit of this register
QMSB	– the Most Significant Bit of the Q register
SDAT	– Serial Data port in/out

If RMSB is chosen as the source, the data shifted out is shifted into the LSB of the register; the result is a "rotate."

Condition Codes affected: CY, Z, S, O

Example 1:

```
SHLR R1 Z;          /* shift the Zero flag into the LSB of R1 */
```

Example 2:

```
SHLR R1 1;          /* shift a '1' into the LSB of R1 */
```

Example 3:

```
SHLR R1 RMSB;       /* rotate R1 left one bit */
```

Example 4:

```
SHLR R1 QMSB;       /* shift the MSB of Q into the LSB of R1 */
```

PACSEL Assembler Reference (Cont.)

SHLRQ	
Instruction Type:	CPU
Operation:	Shift left register and Q
Syntax:	`[label:] [PROGCNTL,] SHLRQ reg rsource qsource [,OUTCNTL];`

Description:

This instruction shifts the selected register (R0 thru R31) and Q left one bit. The data shifted in depends on the rsource and qsource operands.

Z	– the Zero bit flag
CY	– the Carry bit flag
S	– the Sign bit flag
0	– a binary '0'
1	– a binary '1'
RMSB	– the Most Significant Bit of this register
QMSB	– the Most Significant Bit of the Q register
SDAT	– Serial Data port in/out

Condition Codes affected: CY, Z, S, O

Example 1:

```
SHLRQ R1 Z 1;       /* shift the Zero flag into the LSB of R1 */
                    /* also shift Q left one bit */
```

Example 2:

```
SHLRQ RQ CY 1;      /* shift a '1' into the LSB of R1 */
                    /* also shift Q left one bit */
```

Example 3:

```
SHLRQ R1 RMSB 1;    /* rotate R1 and Q left one bit */
```

Example 4:

```
SHLRQ R1 QMSB 0 ;   /* rotate R1 and Q left one bit */
```

PACSEL Assembler Reference (Cont.)

SHRR	
Instruction Type:	CPU
Operation:	Shift Right Register
Syntax:	`[label:] [PROGCNTL,] SHRR reg source [,OUTCNTL];`

Description:

This instruction shifts the selected register (R0 thru R31) right one bit. The data shifted in depends on the source operand:

Z	– the Zero bit flag
CY	– the Carry bit flag
S	– the Sign bit flag
0	– a binary '0'
1	– a binary '1'
RMSB	– the Most Significant Bit of this register
QMSB	– the Most Significant Bit of the Q register
SDAT	– Serial Data port in/out

Condition Codes affected: CY, Z, S, O

Example 1:

```
SHRR R1 Z;      /* shift the Zero flag into the MSB of R1 */
```

Example 2:

```
SHRR R1 1;      /* shift a '1' into the MSB of R1 */
```

Example 3:

```
SHRR R1 RLSB;   /* rotate R1 right one bit */
```

Example 4:

```
SHRR R1 QLSB;   /* shift the LSB of Q into the MSB  of R1 */
```

PACSEL Assembler Reference (Cont.)

SHRRQ	
Instruction Type:	CPU
Operation:	Shift Right Register and Q
Syntax:	`[label:] [PROGCNTL,] SHRRQ reg rsource qsource [,OUTCNTL];`

Description:

This instruction shifts the selected register (R0 thru R31) and Q right one bit. The data shifted in depends on the rsource and qsource operands

Z	– the Zero bit flag
CY	– the Carry bit flag
S	– the Sign bit flag
0	– a binary '0'
1	– a binary '1'
RMSB	– the Most Significant Bit of this register
QMSB	– the Most Significant Bit of the Q register
SDAT	– Serial Data port in/out

Condition Codes affected; CY, Z, S, O

Example 1:

```
SHRRQ R1 Z 1 ;       /* shift the Zero flag into the MSB of R1 */
```

Example 2:

```
SHRRQ R1 1 0;        /* shift a '1' into the MSB of R1 */
```

Example 3:

```
SHRRQ R1 RMSB CY;    /* rotate R1 and Q right one bit */
```

Example 4:

```
SHRRQ R1 QMSB Z;     /* rotate R1 and Q right one bit */
```

PACSEL Assembler Reference (Cont.)

SUB	
Instruction Type:	CPU
Operation:	dest = src1 - src2

Syntax:	`[label:] [PROGCNTL,] SUB dest/src src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] SUB dest src1 src2 [,OUTCNTL];`

Description:

In the first form, this instruction subtracts the source from the destination and places the result in the destination. In the second form, source 2 is subtracted from source 1 and the result placed in the destination register. This is a 2's complement operation.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used
   and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

Example 1:

```
SUB R1 R1;              /* zero R1 */
```

Example 2:

```
MOV   R31 h'0';         /* load immediate value */
SUB   R31 h'1';         /* subtract immediate */
```

Example 3:

```
SUB   BC R23;           /* subtract R23 from block counter */
```

Example 4:

```
SUB   BC R23 R24;       /* (R23 - R24 ) to block counter */
```

Example 5:

```
CONT ,
SUB BC R23 R24 ,        /* full instruction format */
OUT h'A5A5';
```

PACSEL Assembler Reference (Cont.)

TWB	
Instruction Type:	PROGCNTL
Operation:	Three-way branching

Syntax:	`[label:] TWB cc branch-label [,CPU], [,OUTCNTL];`
	`cc -> Condition code`
	`branch-label -> a label to branch to`

Description:

TWB is a three-way-branch instruction.

Here cc, the condition-code is evaluated. If it is TRUE then the Program counter value will be the address of next instruction. In other words, the program will execute next instruction and continues.

If cc evaluates to FALSE, then one of the following two cases is performed, based on the Loop counter value.

1. If Loop counter value is zero, then the Program counter value will be the branch-labels address, i.e., the program branches to the label specified.
2. If the Loop counter value is not zero, then the Program counter will be loaded with whatever value is on the top of the stack. Thus the program will branch to the address given by the top of the stack.

Note, however, that in this case, the top of the stack is not popped-out.

The Condition Codes are:

INTR	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Note:

1. It is the users' responsibility to have valid address on top of the stack prior to executing this instruction.
2. Also, user may have to load the Loop counter prior to this instruction.

Example 1:

```
LDLC label_1;          /* load loop counter */
PUSHLC;                /* push it on stack */

TWB Z LABEL_2;         /* If Z is TRUE, PC = PC + 1
                          If Z is FALSE, two cases :
                            i. if loopcounter = 0, PC = label_2
                           ii. if loopcounter ! = 0,
                               PC = Top of stack( label_1)
                       */
  OUT2;
label_1 :
  OUT5;
label_2 :
  OUT10;
```

PACSEL Assembler Reference (Cont.)

TWBNC	
Instruction Type:	PROGCNTL
Operation:	Three-way branching

Syntax:	`[label:] TWBNC cc branch-label [,CPU] [,OUTCNTL];`
	`cc -> Condition Code`
	`branch-label -> a label to branch to`

Description:

TWBNC is a three-way-branch instruction.

Here cc, the condition-code is evaluated. If it is TRUE then the Program counter value will be the address of next instruction. In other words, the program will execute next instruction and continues.

If cc evaluates to FALSE, then one of the following two cases is performed, based on the Loop counter value.

1. If Loop counter value is zero, then the Program counter value will be the branch-labels address, i.e., the program branches to the label specified.
2. If the Loop counter value is not zero, then the Program counter will be loaded with whatever value is on the top of the stack. Thus the program will branch to the address given by the top of the stack.

Note, however, that in this case, the top of stack is not popped-out.

The Condition Codes are:

INTR	BCZ	FIOR	FICD
Z	C	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Note:

1. It is users' responsibility to have valid address on top of the stack prior to executing this instruction.
2. Also, user may have to load the Loop counter prior to this instruction.

Example 1:

```
LDLC label_1;          /* load loop counter */
PUSHLC;                /* push it on stack */

MOV R0 10;
MOV R1 10;
SUB R0 R1;

TWBNC Z label_2;       /* If Z is TRUE, PC = PC + 1
                          If Z is FALSE, two cases :
                             i. if loopcounter = 0,
                            PC = label_2 ;
                            ii. if loopcounter ! = 0,
                            PC = Top of stack ( label_1)
                       */
  OUT2;
label_1 :
  OUT5;
label_2 :
  OUT10;
```

PACSEL Assembler Reference (Cont.)

XNOR	
Instruction Type:	CPU
Operation:	dest = src1 XNORs src2

Syntax:	`[label:] [PROGCNTL,] XNOR dest/src src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] XNOR dest src1 src2 [,OUTCNTL];`

Description:

In the first form, this instruction XNORs the source and destination and places the result in destination.

In the second form, two sources are XNORed and the result stored in the destination register.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

Example 1:

```
XNOR R1 R2;                 /* R1 = ( R1 XNOR R2) */
```

Example 2:

```
XNOR R31 h'OFFF';           /* XNOR immediate mask */
```

Example 3:

```
/* The following two examples accomplish the same result and
illustrate the use of a three operand XNOR */

MOV      R23 h'123';        /* load register with constant */
MOV      BC R23;            /* load Block Counter from register */
XNOR BC h'FFFE';            /* mask and load Block Counter */
```

Example 4:

```
MOV      R23 h'123';        /* load register with constant */
MOV      R24 h'FFFE';       /* load register with mask */
XNOR BC R23 R24;            /* mask and load Block Counter */
```

PACSEL Assembler Reference (Cont.)

XOR	
Instruction Type:	CPU
Operation:	dest = src1 XOR src2

Syntax:	`[label:] [PROGCNTL,] XOR dest/src src [,OUTCNTL];`
	or
	`[label:] [PROGCNTL,] XOR dest src1 src2 [,OUTCNTL];`

Description:

In the first form, this instruction XORs the source and destination and places the result in destination.

In the second form, two sources are XORed and the result stored in the destination register.

The sources and destination can be chosen from the following table:

Src/Dest	Arguments	Flags Affected
dest	DOR	AF, DOR
	IOR	AF
src	IIR, AIR	AF
	SWPV, <const>, DIR	AF
dest/src	R0...R31, Q, AOR	AF
	ACH, ACL	ACO, AF
	BC	BCZ, AF

Notes:

1. In the first form, either dest/src or src must be R0...R31 or Q
2. In the second form,
   a. at least one of the sources must be R0...R31 or Q
   b. no more than two distinct registers among R0...R31 may be used and
   c. src1 and src2 cannot reference the same member of R0...R31.
3. AF=Arithmetic Flags: Z, CY, S, O.

Example 1:

```
XOR R1 R2;              /* R1 = (R1 XOR R2) */
```

Example 2:

```
XOR R31 h'0FFF';        /* XOR immediate mask */
```

Example 3:

```
/* The following two examples accomplish the same result and
illustrate the use of a three operand XOR */

MOV   R23 h'123';       /* load register with constant */
MOV   BC R23            /* load Block Counter from register */
```

Example 4:

```
MOV   R23 h'123';       /* load register with constant */
MOV   R24 H'FFFE';      /* load register with mask */
XOR   BC R23 R24;       /* mask and load Block Counter */
```

PACSEL Macro Reference

:=	
Instruction Type	CPU
Operation:	Assign value

Syntax:	`[label:] [PROGCNTL,] dest := src [,OUTCNTL];`
	or
	`[label:] dest1 := dest2 := src [,PROGCNTL] [,OUTCNTL];`

Description:

I Form:

The := operator assigns the value of the source to the destination. The destination is one of the following:

R0...R31 ACH ACL BC AOR IOR DOR Q

II Form:

In this form, the destination can be any one of the following:

AOR, ACH, ACL, BC, IOR, DOR

and dest2 can only be,

R0...R31 Q

The source may be one of the following for both forms.

R0...R31 ACH ACL BC AOR IOR DOR Q <const> SWPV

Or the source may be an expression of one of the following forms:

- arg1 <arithmetic op> arg2 [<arithmetic op> arg3]
- arg1 <logical op> arg2
- <unary op> arg
- shift-arg1 <shift op> shift-src1 [shift-arg2 <shift op> shift-src2]

Where:

<arithmetic op>	is	+(add)	-(sub)	*(mul)	/(div)	
<logical op>	is	& (and)	I(or)	^(xor)	!^(xnor)	
<unary op>	is	-(neg)	++(inc)	--(dec)	~(inv)	
<shift op>	is	<< (shift left)		>> (shift right)		
shift-arg1	is	R0...R31	Q			
shift-arg2	is	R0...R31				
shift-src	is	CY	Z	S	1	0
		QLSB	QMSB	RMSB	RLSB	SDAT

Notes:

1. In expressions, one of the arguments must be R0...R31 or Q
2. Arg3, if present, must be "CP" (carry from previous operation)
3. MULTIPLY (*) and divide (/) are macro operations. In these cases, PROGCNTL and OUTCNTL operations should not be specified.
4. The shift operations fall into one of the following formats:

Rn	:=	RN	>>	shift-src			
Rn	:=	Rn	<<	shift-src			
QRn	:=	Q	>>	shift-src	Rn	>>	shift-src
QRn	:=	Q	<<	shift-src	Rn	<<	shift-src

PACSEL Macro Reference (Cont.)

:= (Cont.)

The PROGCNTL and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

Example 1:

```
R5 := BC;
```

Example 2:

```
R3 := BC + 1;
```

Example 3:

```
R27 := BC + Q;
```

Example 4:

```
Rss := ++ R3;
```

Example 5:

```
BC := R1 - R2 - CP;
```

Example 6:

```
AOR := R0 := R0 - R1;
```

ACSIZE	
Instruction Type:	PROGCNTL
Operation:	Set Address Counter size
Syntax:	`[label:] ACSIZE size [,CPU] [,OUTCNTL];`

Description:

Set the Address Counter size. The allowed values for "size" are 16 or 22.

The CPP and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

Example 1:

```
ACSIZE 22;
```

Example 2:

```
ACSIZE 16;
```

PACSEL Macro Reference (Cont.)

CALL	
Instruction Type:	PROGCNTL
Operation:	Call subroutine
Syntax:	`[label:] CALL label 2 [ON] [NOT] [condition-code] [,CPU] [,OUTCNTL];`

Description:

The current program counter is pushed on the stack, and control branches to label 2. If the ON phrase is specified, the condition code is evaluated, optionally inverted by NOT, and the call occurs only if the result is TRUE.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

The Condition Codes are:

INT	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Example 1:

```
CALL XXX;
```

Example 2:

```
CALL XXX ON CY;
```

Example 3:

```
CALL XXX ON NOT CY;
```

Example 4:

```
/* CY from before R1+R3 is used:*/
  CALL XXX ON NOT CY , R1 := R2 + R3;
```

CLEAR	
Instruction Type:	PROGCNTL
Operation:	Clear interrupt(s)
Syntax:	`[label:] CLEAR [int#] ... [int#] [,CPU] [,OUTCNTL];`

Description:

Clear the listed interrupts. The values

```
INT0  INT1  INT2  INT3  INT4  INT5  INT6  INT7
```

may be listed in any order.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

Example 1:

```
CLEAR INT1 INT3 INT4 INT0;     /* clear INT4 has no effect */
```

PACSEL Macro Reference (Cont.)

CONFIGURE	
Instruction Type:	PROGCNTL
Operation:	Set bits in the mode register
Syntax:	`[label:] CONFIGURE [p1] ... [p10] [,CPU] [,OUTCNTL];`

Description:

Set specified bits in the Mode Register. The arguments p1 ... p10 must all come from Set 1, or all from Set 2:

Set 1	Set 2
ACEN	FINRST
ADOE	IO0
BCEN	IO1
FIIR	IO23
FIRST	IO4
HADOE	IO5
HDOE	IO6
INTR	IP7
SIO	PCC

CONFIGURE should generally be used only once, during initialization.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

Example 1:

```
CONFIGURE INTR ADOE ;      /* interrupt mode, ADOE controlled
                           externally */
```

DISABLE	
Instruction Type:	PROGCNTL
Operation:	Disable specified interrupts
Syntax:	`[label:] DISABLE [int#] ... [int#] [,CPU] [,OUTCNTL];`

Description:

Disable the listed interrupts. The values

```
INT0  INT1  INT2  INT3  INT4  INT5  INT6  INT7
```

may be listed in any order. This instruction sets mask bits in the Interrupt Mask Register.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

See also: ENABLE

Example 1:

```
/* shut off software interrupt: INT4 always active */
  DISABLE INT4;
```

Example 2:

```
/* disable selected interrupts; note lack of order */
  DISABLE INT7 INT3 INT4 INT5 INT0;
```

PACSEL Macro Reference (Cont.)

ENABLE	
Instruction Type:	PROGCNTL
Operation:	Enable specified interrupts
Syntax:	`[label:] ENABLE [int#] ... [int#] [,CPU] [,OUTCNTL];`

Description:

Enable the listed interrupts. The values

```
INT0  INT1  INT2  INT3  INT4  INT5  INT6  INT7
```

may be listed in any order. This instruction clears mask bits in the Interrupt Mask Register.

If Interrupt 4 is enabled, PAC1000 Peripheral Controller will immediately process an Interrupt 4, since this interrupt is always active.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

See also: DISABLE

Example 1:

```
/* execute a software interrupt: INT4 always active */
  ENABLE INT4;
```

Example 2:

```
/* enable all interrupts; note lack of order */
  ENABLE INT7 INT6 INT3 INT4 INT5 INT2 INT2 INT0;
```

PACSEL Macro Reference (Cont.)

FOR...ENDFOR	
Instruction Type:	PROGCNTL structure
Operation:	Loop <count> times

Syntax:	`[label:]  FOR count [,CPU] [,OUTCNTL]`
	`[ executed <count+1> times ; ]`
	`.`
	`.`
	`.`
	`ENDFOR [,CPU] [,OUTCNTL];`

Description:

At the head of the loop, the count is stored (in one cycle) and the loop is executed. (The loop is always executed once, even if the count is zero.) At the conclusion of the loop body, the count is evaluated and then decremented. If the result is greater than zero, control is transferred to beginning of the loop body.

The maximum value for "count" is 1023 decimal.

Alternative usage: If "count" is omitted, the value computed from the CPU operation following "FOR" is used. If this alternative is chosen, the CPU operation is mandatory.

The CPU and OUTCNTL operations following the loop count, if present, are unconditionally executed in the same cycle. The CPU and OUTCNTL operations following ENDFOR are not included in the loop body.

Stack effects: an implicit PUSH is performed at the FOR and a match implicit POP is done at the ENDFOR. These extra operations support nested loops. If you branch out of the loop, you must manage the stack explicitly by performing a POP operation.

Timing: FOR executes in one machine cycle; ENDFOR generates two lines of code–a branch back to the beginning of the loop and a pop of the stack into the loop counter.

Example 1:

```
/* inefficient method of adding 47 to $1 */
  FOR 47;
        R1 := ++ R1;
  ENDFOR;
```

Example 2:

```
/* similar, with CPU and OUTCNTL */
  R2 := 3;
  FOR 47, R1 := R2 , OUT 2;
        R1 := ++ R1;
  ENDFOR, OUT 0;
```

Example 3:

```
/* similar, but count = CPU result = 3 */
  R2 := 3;
  FOR , R1 := R2 , OUT 2;
        R1 := ++ R1;
  ENDFOR, OUT 0;
```

PACSEL Macro Reference (Cont.)

GOTO	
Instruction Type:	PROGCNTL
Operation:	Unconditional jump
Syntax:	`[label:] GOTO label2 [ON] [NOT] [condition-code] [,CPU] [,OUTCNTL];`

Description:

Control branches to the label 2. If the ON phrase is specified, the condition code is evaluated, optionally inverted by NOT, and the operation occurs only if the result is TRUE.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

The Condition Codes are:

INT	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Example 1:

```
GOTO XXX;
```

Example 2:

```
GOTO XXX ON CY;
```

Example 3:

```
GOTO XXX ON NOT CY;
```

Example 4:

```
/* CY from before R1+R3 is used: */
GOTO XXX ON NOT CY , R1 := R2 + R3;
```

PACSEL Macro Reference (Cont.)

IF...ELSE...ENDIF

Instruction Type: PROGCNTL structure

Operation: Conditional Branch

Syntax:

```
[label:] IF [NOT] condition-code [,CPU] [,OUTCNTL];
                [ executed if condition true ; ]
         .
         .
         .
                [ ELSE] [,CPU] [ ,OUTCNTL];
                     [ executed if condition false ; ]
                ENDIF [,CPU] [,OUTCNTL];
```

or

```
[label:] IF arg1 <relational op> arg2 [,CPU] [,OUTCNTL];
                [ executed if condition true ; ]
         .
         .
         .
                [ ELSE] [,CPU] [ ,OUTCNTL];
                     [executed if condition false ; ]
                ENDIF [,CPU] [,OUTCNTL];
```

Description:

The condition-code is evaluated, and optionally inverted by the NOT keyword. If the result is TRUE, the IF portion is executed. Otherwise the ELSE portion is executed, if it is present. If it is not present, control passes to the ENDIF.

The CPU and OUTCNTL operations following the condition code, if present, are unconditionally executed in the same cycle.

The CPU and OUTCNTL operations following the ELSE are executed only if the ELSE phrase is selected. These are performed in an extra cycle inserted for the purpose. The CPU and OUTCNTL operations following the ENDIF, if present, are always executed.

This structure may be nested to a maximum depth of 15. Each IF must be terminated with a matching ENDIF.

The Condition Codes are:

INT	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

In the second form of this instruction, one of the relational operators

==

or

!=

is specified to test equality or inequality of two arguments. In this form, arg1 and arg2 may be any of the following:

R0...R31 ACH ACL BC AIR IIR DIR AOR SWPV a constant

If this form is used, 2 machine cycles are required for execution of the IF instruction.

PACSEL Macro Reference (Cont.)

IF...ELSE...ENDIF (Cont.)

Example 1:

```
IF ACO;                    /* if Address Counter all ones */
     ACO := R25;           /* relaod with contents of R25 */
ENDIF;
```

Example 2:

```
/* similar, with ... */
  IF CY , R25 := R23, OUT h'0';    /* CPU and OUTCNTL */
     BC := R25;
  ENDIF;
```

Example 3:

```
/* similar, with ELSE phrase */
  IF Z;
     ACO := R25;
  ELSE;
     AOR := ++ R25;
  ENDIF;
```

Example 4:

```
/* similar, with... */
  IF Z , R25 := R23, OUT h'0';     /* CPU and OUTCNTL */
  R20 :+ R25;
  ELSE;
     R26 := ++ R25;
  ENDIF;
```

Example 5:

```
/* logically same as example4 */
  IF NOT ACO , R25 := R25, OUT h'0';
    R26 := ++ R25;
  ELSE;
    R20 = R25;
  ENDIF;
```

PACSEL Macro Reference (Cont.)

INPUT	
Instruction Type:	PROGCNTL
Operation:	Set I/O port pin mode to input
Syntax:	`[label:] INPUT [PIN#] ... [pin#] [,CPU] [,OUTCNTL];`

Description:

Set the listed I/O port pins to inputs. The pins

IO0 IO1 IO2 IO3 IO4 IO5 IO6 IO7

may be listed in any order.

This instruction resets bits in the I/O Configuration Register.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

See also: OUTPUT

Example 1:

```
INPUT IO3 IO7 IO4;     /* set these pins to inputs */
```

OUT	
Instruction Type:	OUTCNTL
Operation:	Output control (F) value
Syntax:	`[label:] [PROGCNTL,] [CPU,] OUT constant;`

Description:

OUT directs the device to place the specified constant value on the Control outputs.

The value specified is placed on the 16 bit (user) output lines during the instruction cycle which includes this operation. The constant is usually specified directly as a hex value in the range

0 <= constant <= h'FFFF'

or the value may be evaluated as an expression. The value is determined at assembly time; there is no way of computing an OUT value at run time.

The value may be formed from 2 up to 16 constants, or values that evaluate to constants. In this case, all the values are bitwise OR'd to produce the output value. The OR operator in this case is implied.

When an OUTCNTL operation is omitted from an instruction, the assembler will provide a default OUT value equal to that most previously specified. (If no OUT operations occur in a module, the assembler will use 0.) Thus OUT defaults are maintained in order of assembly, not order of execution.

Example 1:

```
OUT 59;
```

Example 2:

```
XXX equ 45     ;
GOTO E2 , OUT XXX;
```

PACSEL Macro Reference (Cont.)

LOADBP	
Instruction Type:	PROGCNTL
Operation:	Load Breakpoint
Syntax:	`[label:] LOADBP constant/label <,CPU> <,OUTCNTL>;`

Description:

Load the Breakpoint Register with the specified constant or address value. The value may be a number or a symbol. The maximum value is 1023 decimal.

Alternative usage: If "constant" is omitted, the value computed from the CPU operation following "LOADBP" is used. If this alternative is chosen, the CPU operation is mandatory.

The CPU and OUTCNTL operations, if present, are executed in the same cycle.

Example 1:

```
LOADBP h'200';
```

Example 2:

```
LOADBP , R3 := R1 + ACH;     /* BP = R1 + ACH */
```

OUTPUT	
Instruction Type:	PROGCNTL
Operation:	Set I/O port pin mode to output
Syntax:	`[label:] OUTPUT [pin#] ... [pin#] [,CPU] [,OUTCNTL];`

Description:

Set the listed I/O port pins to outputs. The pins

IO0 IO1 IO2 IO3 IO4 IO5 IO6 IO7

may be listed in any order.

This instruction sets bits in the I/O Configuration Register.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

See also: INPUT

Example 1:

```
OUTPUT IO3 IO2 IO4;   /* set these pins to outputs */
```

PACSEL Macro Reference (Cont.)

PSWITCH...PRIORITY	
Instruction Type:	PROGCNTL structure
Operation:	Prioritized eightway branch

Syntax:	`[label:] PSWITCH [,CPU] [,OUTCNTL];`
	`PRIORITY n, [PROGCNTL] [,CPU] [,OUTCNTL];`
	`PRIORITY m, [PROGCNTL] [,CPU] [,OUTCNTL];`
	`... [up to 8 branches total]`
	`ENDPSWITCH [,CPU] [,OUTCNTL];`

Description:

The current interrupt status is evaluated. If no interrupt source is active, then the program branches to the ENDPSWITCH location.

If one interrupt is active, the program branches to the corresponding case. If more than one is active, the program branches to highest priority case.

If a case does not include a jump operation, the next sequential case will be executed. Ordinarily, then, each case will contain a jump.

This operation is effective only if the INTR bit of the MODE register is clear. Otherwise, the interrupts will be processed by the normal interrupt vector mechanism. This structure is useful in systems where interrupts are not used but in which prioritization of polled inputs is important.

The CPU and OUTCNTL operations following the PSWITCH, if present, are unconditionally executed in the same cycle. The CPU and OUTCNTL operations following the ENDPSWITCH, if present, are unconditionally executed if control reaches the ENDPSWITCH.

Example 1:

```
PSWITCH
    PRIORITY 0, GOTO PINT0; /* process into 0 */
    PRIORITY 1, GOTO PINT1; /* process into 1 */
    PRIORITY 2, GOTO PINT2; /* process into 2 */
    PRIORITY 3, GOTO PINT3; /* process into 3 */
    PRIORITY 4, GOTO PINT4; /* process into 4 */
    PRIORITY 5, GOTO PINT5; /* process into 5 */
    PRIORITY 6, GOTO PINT6; /* process into 6 */
    PRIORITY 7, GOTO PINT7; /* process into 7 */
ENDPSWITCH;
PINT0:              /* framework for each routine */
.
.
.
   JMP NEXT;        /* original entry via PSWITCH */
```

PACSEL Macro Reference (Cont.)

RESET	
Instruction Type:	PROGCNTL
Operation:	Reset bits in the Control Register
Syntax:	`[label:] RESET [PL] ... [P10] [,CPU] [,OUTCNTL];`

Description:

Reset specified bits in the Control Register. The parameters

ACEN	BCEN	HDOE	HADOE	ADOE
HDSEL1	HDSEL0	DIREN	AIREN	ASEL

may be listed in any order.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

See also: SET

Example 1:

```
RESET HDSEL0 DIREN ASEL;
```

RETURN	
Instruction Type:	PROGCNTL
Operation:	Return from subroutine
Syntax:	`[label:] RETURN [ON] [NOT] [condition-code] [,CPU] [,OUTCNTL];`

Description:

Control is returned to the code following the most recent CALL. If the ON phrase is specified, the condition code is evaluated, optionally inverted by NOT, and the return occurs only if the result is TRUE.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

The Condition Codes are:

INT	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7-CC0		

Example 1:

```
    CALL XXX;
XXX:  IF BCZ;
          RETURN;
    ELSE;
          R1 := ++ R1;
          RETURN;
ENDIF:
CALL XXX;
```

Example 2:

```
    CALL YYY;
YYY: RETURN ON BCZ     /* logically the same as Example 1 */
    R1 := ++ R1;
    RETURN;
```

PACSEL Macro Reference (Cont.)

SET	
Instruction Type:	PROGCNTL
Operation:	Set bits in the Control Register
Syntax:	`[label:] SET [p1] ... [p10] [,CPU] [,OUTCNTL];`

Description:

Set specified bits in the Control Register. The parameters

```
ACEN      BCEN      HDOE      HADOE     ADOE
HDSEL1    HDSEL0    DIREN     AIREN     ASEL
```

may be listed in any order.

The CPU and OUTCNTL operations, if present, are unconditionally executed in the same cycle.

See also: RESET

Example 1:

```
SET HDSEL0 DIREN ASEL;
```

PACSEL Macro Reference (Cont.)

SWITCH...CASE	
Instruction Type:	PROGCNTL structure
Operation:	Multiway branch

Syntax:	`[label:] SWITCH casegroup [,CPU] [,OUTCNTL];`
	`case n, [,PROGCNTL] [,CPU] [,OUTCNTL];`
	`case m, [PROGCNTL] [,CPU] [,OUTCNTL];`
	`... [up to 16 cases total]`
	`ENDSWITCH [,CPU] [,OUTCNTL];`

Description:

The value of the specified Case Group (CG0, CG1, CG2, or CG3) is used to branch control to one of up to 16 specified cases. The value of case enumerators (n, m, ...) are numbers or constants between 0 and 15, in any order.

If a case does not include a jump operation, the next sequential case will be executed. Ordinarily, then, each case will contain a jump.

The Case Groups are:

'CG0' = [CC3, CC2, CC1, CC0]
'CG1' = [CC7, CC6, CC5, CC4]
'CG2' = [INTR, BCZ, FIOR, FICD]
'CG3' = [Z, O, S, CY]

The CPU and OUTCNTL operations following the case group, if present, are unconditionally executed in the same cycle. The CPU and OUTCNTL operations following the ENDSWITCH, if present, are unconditionally executed if control reaches the ENDSWITCH.

Example 1:

```
    SWITCH CG2 , OUT h'FFFF';           /* output marker */
          CASE 00, GOTO NEXT, OUT 0 ;   /* output CG value... */
          CASE 01, GOTO NEXT, OUT 1;
          CASE 02, GOTO NEXT, OUT 2;
          CASE 03, GOTO NEXT, OUT 3;
          CASE 04, GOTO NEXT, OUT 4;
          CASE 05, GOTO NEXT, OUT 5;
          CASE 06, GOTO NEXT, OUT 6;
          CASE 07, GOTO NEXT, OUT 7;
          CASE 08, GOTO NEXT, OUT 8;
          CASE 09, GOTO NEXT, OUT 9;
          CASE 10, GOTO NEXT, OUT 10;
          CASE 11, GOTO NEXT, OUT 11;
          CASE 12, GOTO NEXT, OUT 12;
          CASE 13, GOTO NEXT, OUT 13;
          CASE 14, GOTO NEXT, OUT 14;
          CASE 15, GOTO NEXT, OUT 15;
NEXT:     ENDSWITCH, OUT 16;            /*output end marker*/
```

PACSEL Macro Reference (Cont.)

WHILE...ENDWHILE	
Instruction Type:	PROGCNTL structure
Operation:	Conditional Loop

Syntax:	`[label:] WHILE [NOT] condition-code [,CPU] [,OUTCNTL];`
	`[ executed while condition true ; ]`
	`.`
	`.`
	`.`
	`ENDWHILE [,CPU] [,OUTCNTL];`

Description:

The condition-code is evaluated, and optionally inverted by the NOT keyword. The statements inside the structure are executed only if the result is TRUE. Otherwise control passes immediately to the code following ENDWHILE. The loop body will not be executed at all if the condition is initially FALSE.

The CPU and OUTCNTL operations following the condition code, if present, are unconditionally executed in the same cycle. The CPU and OUTCNTL operations following ENDWHILE are logically included in the loop body. These will not be executed when the condition is or becomes false.

The Condition Codes are:

INT	BCZ	FIOR	FICD
Z	O	S	CY
ACO	FIXP	FIIR	STKF
DOR	CC7 CC0		

This structure may be nested to a maximum depth of 15. Each WHILE must be terminated with a matching ENDWHILE.

Example 1:

```
OUT h'FFFF';       /* output all ones */
WHILE NOT BCZ ,
   OUT 0;          /* output 0 until Block Counter = zero, */
ENDWHILE;
OUT h'FFFF';       /* then output all ones */
```

Example 2:

```
OUT h'FFFF';                    /* output all ones */
WHILE NOT BCZ , OUT h'A5A5';    /* insert 1 cycle of A5A5 */
OUT 0;                          /* output 0 until Block Counter =
                                zero, */

ENDWHILE;
OUT h'FFFF';                    /* then output all ones */
```

PAC1000 Application Notes

Section Index

PAC1000 Application Notes

Programmable Peripheral Application Note 005

PAC1000 as a High-Speed Four-Channel DMA Controller

By Arye Ziklik and Kiran Buch

Abstract

The objective of this Application Note is to demonstrate the use of the PAC1000 Programmable Peripheral Controller in a typical high performance application. The text describes an implementation of a generic four-channel DMA controller that supports transfer rates of up to *16 Mbyte/sec* (8 Mword/sec) in 16-bit data-bus environments.

This Application Note covers the terminology of DMA operations as well as an implementation description. The readers will be able to use this article as a get-started tutorial that shows how to configure the PAC1000 for any specific task.

Introduction

A DMA (Direct Memory Access) controller coordinates fast data transfers between peripheral devices and the system memory. All possible transfer combinations might occur: device to device, device to memory or memory to memory. By taking care of these high-speed transfers, the host computer (typically a Microprocessor) is off-loaded from these time-consuming tasks and can execute other operations concurrently, on its local bus.

We refer to peripherals such as FIFOs, video, communication, graphics or serial channel controllers, latches, ports, etc., as devices in this text. The distinction between memory and device is that a memory needs an explicit address in order to specify a byte or a word, whereas a device requires only strobes (such as: $\overline{RD}$, $\overline{WR}$, $\overline{CS}$) combined sometimes with additional hand-shaking signals for data accessing.

The PAC1000 is a perfect match for most DMA applications. Its unique structure, shown in Figure 1 and Figure 2, allows the user to execute three independent instructions in one cycle. The ability of the PAC1000 to perform three different tasks concurrently (Control, Output and CPU) is fully exploited here, thereby speeding-up DMA transfers.

For example, during DMA operations, the control section checks for the block-count termination, the output control section generates $\overline{RD}$ and $\overline{WR}$ strobes, and the CPU calculates and produces the next address. All these activities occur simultaneously during the same clock cycle(!).

Unlike most other available DMA controllers, the PAC1000 is a programmable peripheral controller. It may be easily modified by reprogramming to support various DMA schemes.

Figure 3 illustrates a typical system configuration using the PAC1000 as a DMA controller. The host controls the system bus as well as its local bus (not shown here). It can also access the memories, the devices as well as the PAC1000 via the system bus. It does so by driving the Address, Control and Data buses.

Initially the PAC1000 is in the slave mode, waiting for host messages. Once the host begins a channel initialization phase by writing into the PAC1000's FIFO, a DMA operation will start. In that phase, the host instructs the PAC1000 of the required DMA transfer. The PAC1000 then decodes the transfer type and optimizes it internally to perform at the fastest rate the surrounding hardware allows. At this point the PAC1000 requests the system bus from the bus arbiter. When the bus is granted to the PAC1000, it becomes the Bus Master, driving the address, data and control buses.

If the DMA operation is fully completed, or a higher priority transfer is pending, or the host or active devices abort the transfer, a DMA transfer can be successfully terminated or suspended, respectively. In all of these cases, system control is returned to the host and the PAC1000 re-enters to Slave Mode.

Figure 1. PAC1000 Programmable Peripheral Controller Block Diagram

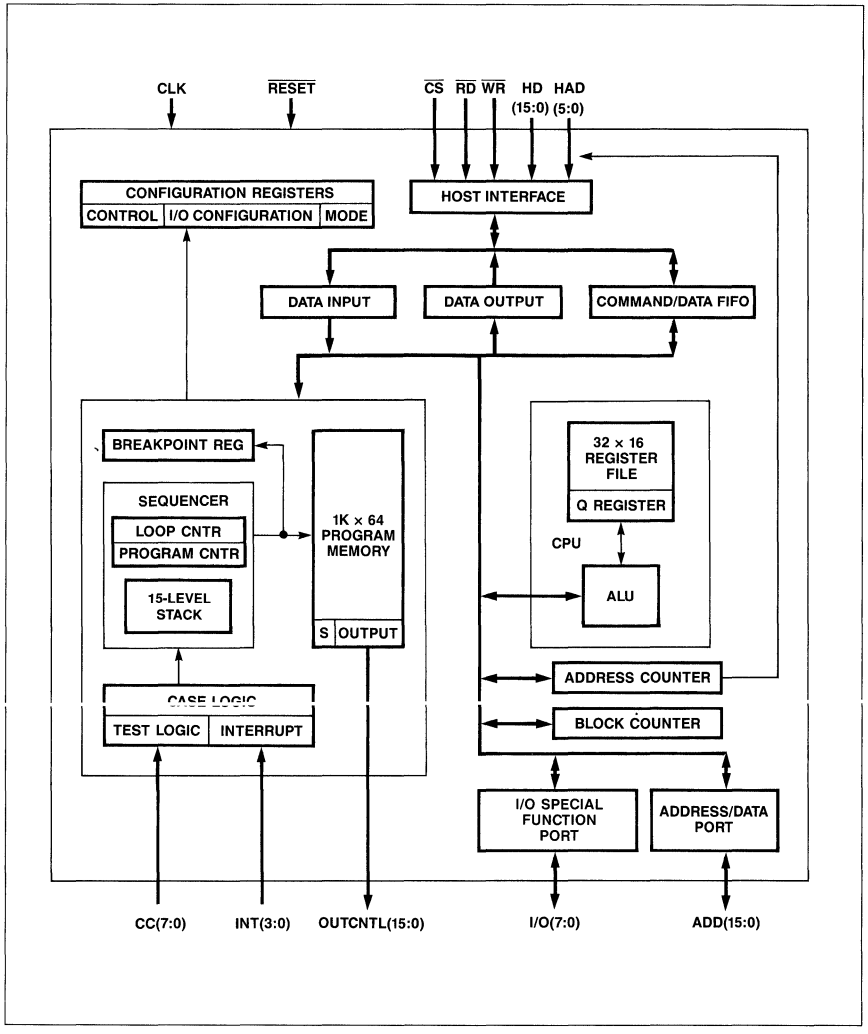
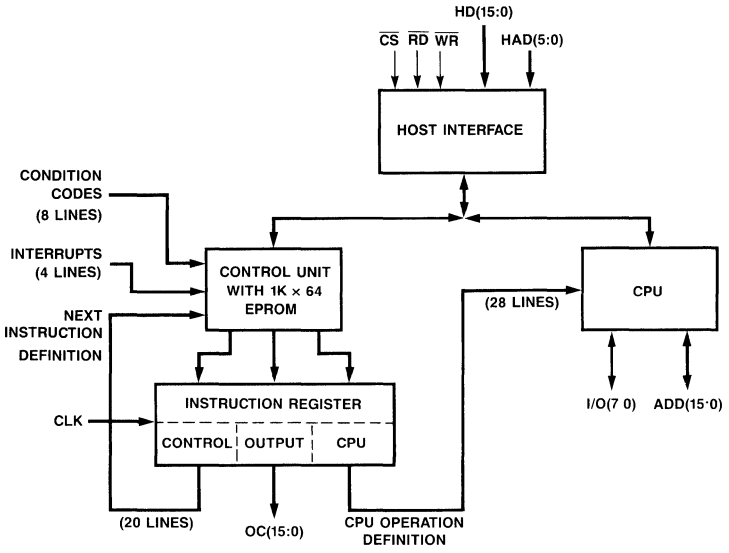


Figure 2. Single Cycle Control Architecture



Important Features:

- One cycle per instruction.
- 16 MHz instruction execution rate.
- Every instruction executes 3 parallel operations (Control, Output, CPU).

4

Transfer Modes

There are two transfer modes: Fly-by and Dual cycle.

Fly-by is the fastest transfer mode (refer to Figure 4). Transfers can be carried out at a rate of up to 8 Mword/sec provided that the PAC1000 uses a 16-MHz clock. In this application note, Fly-by can only be used between memory and device if they share the same data-bus path (either 8 or 16 bits).

The fly-by operation is initiated by a DMARQ from the device. The PAC1000 explicitly addresses the memory, while sending the $\overline{RD}$ strobe to the source side and the $\overline{WR}$ strobe to the destination side. It also acknowledges the device with the $\overline{DMACK}$ signal that serves as the device's $\overline{CS}$ signal. Data is then directly transferred from the source to the destination in one bus cycle.

Double-cycle is a transfer mode comprised of two bus cycles. It takes place whenever one of the following DMA combinations is specified (refer to Figure 5):

- ❑ Memory to/from device that is not connected to the same part of the data-bus.
- ❑ Memory to Memory transfers (require the generation of two different explicit addresses).
- ❑ Device to Device transfers (with simple additional hardware it might be easily upgraded to support the Fly-by mode, too).

Once the transfer has started, the PAC1000 reads an operand from the source on the first bus-cycle, processes it, and then writes that operand on the second bus cycle into the destination.

The READY signal enables the PAC1000 to synchronize its operations with slow memories or devices (whenever they are explicitly addressed). READY is an active-high signal, derived from the address decoder. It is driven low as long as the addressed memory or device is not ready to finish the current bus-cycle.

Figure 3. System Block Diagram

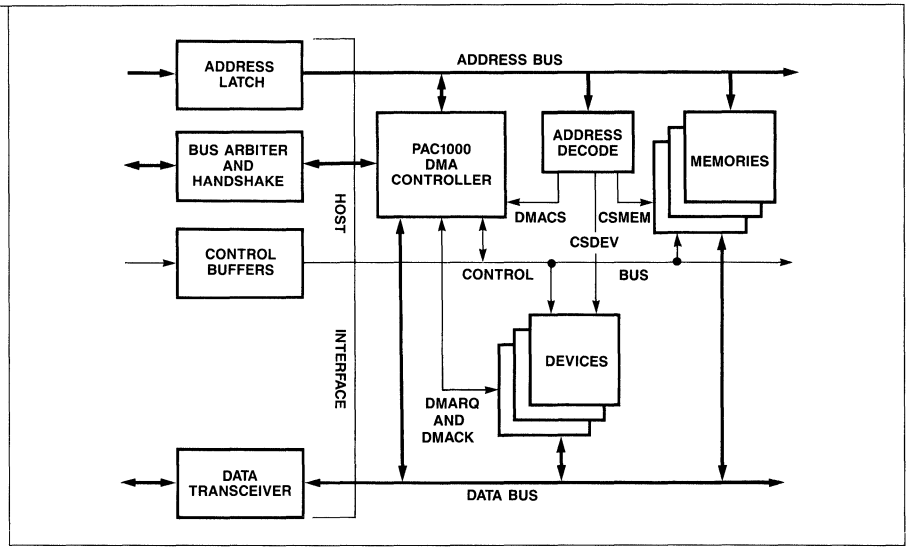


Figure 4. Fly-by DMA Transfer

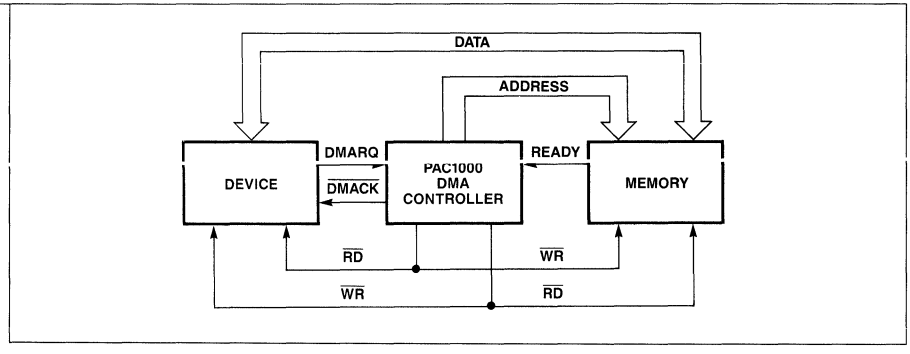


Figure 5. Double Cycle DMA Transfer — Memory to Device

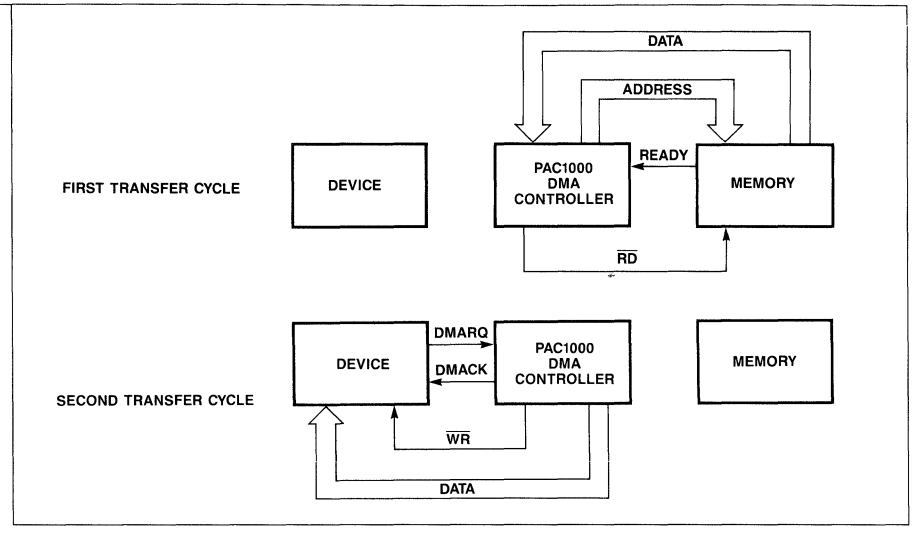


Figure 5. (Cont.) Double Cycle DMA Transfer — Memory to Memory

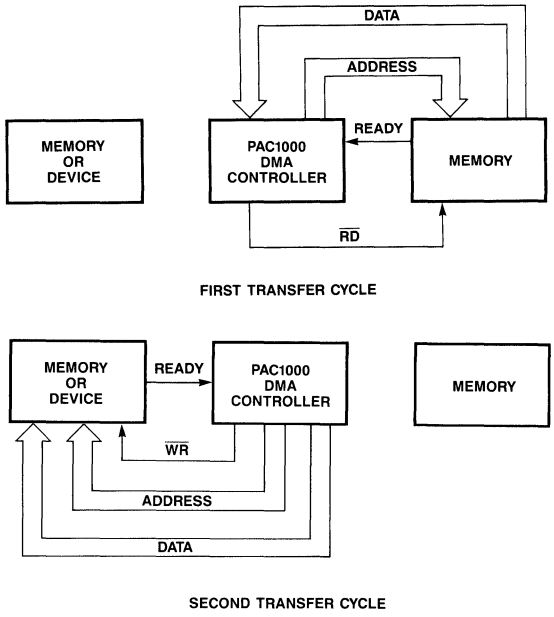
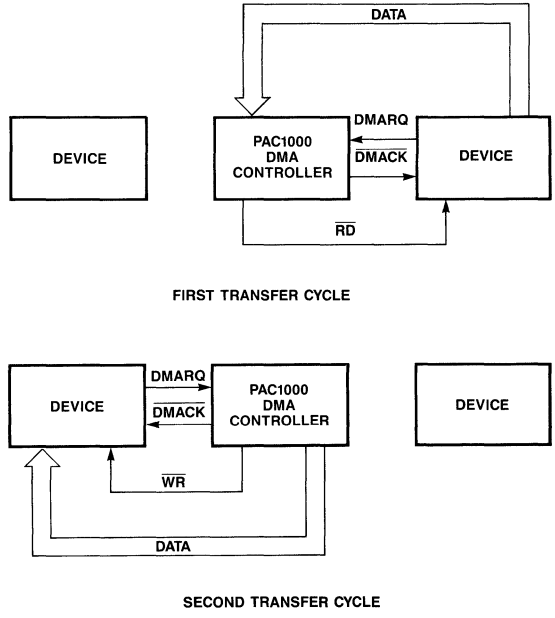


Figure 5. (Cont.) Double Cycle DMA Transfer — Device to Device



4

Request Modes

Requests may be externally generated by a device or internally created by the auto-request mechanism of the PAC1000, whenever a memory to memory transfer is performed. Auto-requests are always pending so that the PAC1000 can work at its maximum speed, provided that the memories are always ready. Otherwise, the PAC1000 adapts itself to the READY signal.

External requests may be of either the block-type or of the single-operand transfer mode. Block-type transfers are provided for high-speed devices that are capable of meeting the speed rate of the PAC1000. DMARQ is asserted at the beginning of the block transfer and remains so as long as the transfer is in process. Single-operand transfers are used by slow devices. They toggle on and off the DMARQ. Each individual transfer is indicated by an active high DMARQ level. When the transfer is completed, DMARQ is held low until the device is ready for the next transfer cycle, and so on.

Some important observations:

- Memory to device (or device to memory) transfers will begin only after an external DMARQ is asserted by the device.
- Synchronization with the memory is always achieved via the Ready signal.

Table 1 briefly summarizes the transfer and request options:

Table 1. Summary of Transfer and Request Modes

Transfer Type	DMA Mode	Transfer Mode
Memory to Memory	Two Bus-cycles	Block
Memory to Device or Device to Memory	Fly-by or Two Bus-Cycles	Block or Single Operand
Device to Device	Two Bus-Cycles	Block or Single Operand

Functional Description

General:

Figure 6 contains the circuit diagram. Refer also to Appendix 1 for the Pin Description Table. The PAC1000 is configured in this application as a four-channel DMA controller. This means that it can handle up to four DMA transfers concurrently, on a prioritized basis. Each of the channels can be any one of the above-mentioned DMA transfer types. The maximum transfer rate is accomplished during Fly-by transfers with rates approaching 8 Mword/sec for word transfers or 8 Mbyte/sec for byte transfers. Double-cycle transfer modes achieve a rate of up to 4 Mword/sec (in word transfers) or 4 Mbyte/sec (in byte transfers). The only exception to this is the Memory to Memory transfer mode which is a little bit slower due to the internal creation of two different 24-bit addresses.

The PAC1000 drives 24 address lines and handles a 16-bit data bus, so it is well tuned for most common high-performance buses or Microprocessors. The maximum operand block-size is 64K (in accordance with VMEbus specs, for example).

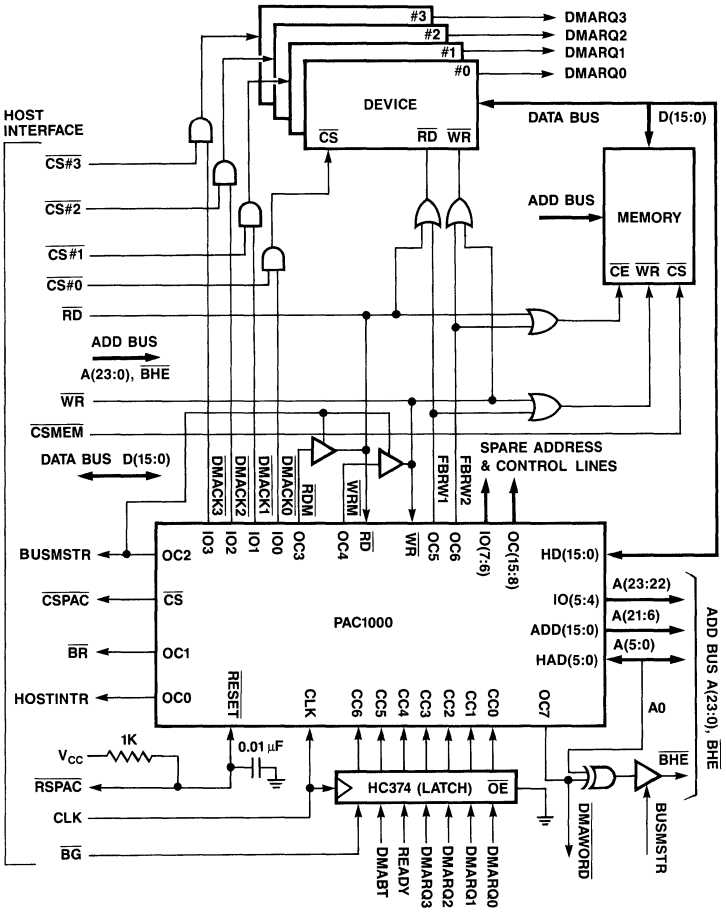
Host-PAC1000 Communication:

DMA specifications are programmed into the PAC1000 by the host, according to the message format of Appendix 2. The host writes eight words into the PAC1000's FIFO. The command message fully specifies one of the four possible channels that can be active at the same time. Word 1 defines the transfer characteristics of the DMA operation: transfer type, data bus width, device numbers (redundant in Memory to Memory operations), channel-priority and transfer mode. Bit 12 in that word serves as a software abort-command bit. When set, it instructs the DMA controller to cease the transfers of the channel specified in that command buffer.

The low-order byte of word 7 is a DMA-transfer identification number. It assigns a serial number to a DMA process. Whenever the PAC1000 sends a status message to the host, that number is also included in order to unambiguously identify the process that has either normally terminated or abnormally aborted (by an external device or due to a PAC1000 exception).

Functional Description (Cont.)

Figure 6. PAC1000 Configured as a Generic High-Speed DMA Controller



Functional Description (Cont.)

Several fields in the command buffer are optional. For instance, in transfers where devices are involved, one can still specify the explicit addresses of the source and the destination even though it has already been defined by the command word's device-number field (Appendix 2 — command word format). This feature allows the programmer to define the device interface with either explicit or implicit address.

Whenever the PAC1000 has to inform the host of an important event, it prepares a status word in its DOR (Data Output Register), enters the slave mode and interrupts the host by raising the HOSTINTR line. The possible messages are:

- Reject the Command buffer with the specified identification number because of internal discrepancies or illegal combinations.
- Propagate a Hardware DMA abort, generated by the source or the destination of the current transfer.
- Signal a PAC1000 exception. The host is capable of reading the PAC1000's SR register in order to find out the cause.
- An end-of-count message. This transfer has been normally terminated.

Initial State and Slave Mode:
After a reset (either a power-on reset or a reset through the $\overline{\text{RSPAC}}$ line driven from the host side), the PAC1000 enters its initial state, which is the Slave Mode. Table 2 describes the signal states during the Slave Mode. The PAC1000 monitors its internal FIIR flag (FIFO Input Ready) and when it is not set, the FIFO is full with a new command buffer written by the host. The PAC1000 decodes the message and acts accordingly. If it is a memory to memory transfer, then it immediately requests the bus. When one or two devices participate in a transfer operation, the PAC1000 monitors the corresponding DMARQ lines to determine when to issue a bus request to the arbiter. The PAC1000 requests the bus by lowering $\overline{\text{BR}}$. Then it waits for $\overline{\text{BG}}$ to go low in order to switch to the Master Mode.

Master Mode:
Upon gaining mastership, the PAC1000 drives the HOSTINTR signal low and BUSMSTR high. BUSMSTR remains high (active) as long as the PAC1000 remains master of the system bus, thereby enabling $\overline{\text{RDM}}$ and $\overline{\text{WRM}}$ to $\overline{\text{RD}}$ and $\overline{\text{WR}}$, respectively. $\overline{\text{BR}}$ is set high (= not active). According to the required DMA operation, the PAC1000 drives the appropriate address and data lines, and the $\overline{\text{RDM}}$, $\overline{\text{WRM}}$ and $\overline{\text{DMACK}}$ signals.

DMA transfers may be successfully ended (when the terminal-count expires) or aborted. Abortion can emanate either from an external DMABT signal that is driven by one of the DMA participants, or from an internal exception recognized by the PAC1000. Whenever one of the above events occur, the PAC1000 changes its mode to the Slave mode, writes a status word into the DOR register (discussed previously) and raises the HOSTINTR line to cause the host to read that information through its own Interrupt routine.

Releasing and resuming bus control:
The host is allocated a higher priority than the PAC1000 by the bus arbiter. This is done in order to enable the host to suspend DMA transfers whenever it needs the bus. Each time the host accesses an address that resides within the system bus domain (including the $\overline{\text{CSPAC}}$ address), the bus will be granted. If the PAC1000 is the current master (as reflected by BUSMSTR), the bus arbiter will negate $\overline{\text{BG}}$ (high level). The PAC1000 monitors this line while it is a bus Master and consequently will relinquish the bus and return to the slave mode. The host might use the bus for programming the PAC1000 with a new DMA channel. Upon completion of the host activities over the system bus ($\overline{\text{BG}}$ becomes high), the PAC1000 checks whether DMA transfers are still pending. If this is the case, it will request the bus. When the bus is granted, it will determine whether to continue the suspended transfer or to start a higher priority pending-DMA request. If it starts a higher priority transfer, then the suspended operation will be resumed after the completion of the higher priority transfer.

$\overline{\text{DMAWORD}}$ is set low during word transfers and high during byte transfers. It is used to derive the $\overline{\text{BHE}}$ strobe, as displayed in Figure 6. The most efficient transfer method is the word transfer mode. In order to use it, the specified addresses must be even, otherwise the PAC1000 will perform only in the byte transfer mode regardless of the command word content.

Functional Description (Cont.)

Table 2. Signal States During the Slave Mode

PAC1000 Signal Names	Function	Signal States
ADD(15:0)	A(21:6)	Float
HAD(5:0)	A(5:0)	Input
IO(5:4)	A(23:22)	Float
OC6, OC5	FBRW2, FBRW1	0, 0 — Normal Operation
OC4, OC3	$\overline{RDM}$, $\overline{WRM}$	Don't Care
10(3:0)	$\overline{DMACK}$ (3:0)	1,1,1,1 — Normal Operation
OC2	BUSMSTR	0 — Non-active
OC1	$\overline{BR}$	1 — Non-active
OC0	HOSTINTR	0 — Non-active
OC7	$\overline{DMAWORD}$	Don't Care
HD(15:0)	D(15:0)	Input

Hardware Considerations

Figure 6 is the detailed schematic diagram. The host side is beyond the scope of this paper since it is application dependent. In addition to the PAC1000, there are a few standard glue-logic chips used to interface with the memory and the four devices.

Throughout the following description it is assumed that the glue-logic components belong to the HC family. However, since the PAC1000 is a fully TTL compatible device implemented in CMOS technology, the reader can use other glue-logic families like: LSTTL, HCT, etc.

The HC374 latch is gated into the condition code inputs by the PAC1000's clock, thus ensuring that the CC7–CC0 lines will meet the set-up time requirements.

The three-state buffers controlled by BUSMSTR, are part of a HC126 chip. They are used to float the PAC1000's $\overline{BHE}$, $\overline{RDM}$, $\overline{WRM}$ control lines during slave operations, because at that time these signals are driven by the host.

The four AND-Gates amount to one HC08 chip. They enable either the host side (during Slave operations) or the PAC1000 (in the Master Mode) to drive the appropriate device $\overline{CS}$ signals.

The four OR-Gates comprise together one HC32 chip. They are used during Fly-by operations to avoid $\overline{CE}$, $\overline{RD}$, $\overline{WR}$ from reaching the selected devices and memories concurrently (for functional explanation, refer to the Pin Description Table, Appendix 1).

Prior to the setting of $\overline{BG}$ in the active position (low), the arbiter floats the data bus D(15:0), address bus A(23:0) and $\overline{BHE}$, $\overline{RD}$ and $\overline{WR}$ from the host side. As long as BUSMSTR remains high, these lines are driven by the PAC1000.

The six chip select lines from the host side ($\overline{CS}$#3 — $\overline{CS}$#0, $\overline{CSMEM}$ and $\overline{CSPAC}$) are derived from the system address decode block, as illustrated in Figure 6. During the time that the PAC1000 is the bus master, the address decode block (shown in Figure 3) is driven by the PAC1000's address lines. Therefore, the PAC1000 can access memories and devices in the same manner the host does.

The $\overline{DMACK3}$–$\overline{DMACK0}$ signals provide the PAC1000 with an alternative chip select generation method to the devices. It is considerably faster than the host's method, since there is no need to generate explicit device addresses inside the PAC1000.

Hardware Considerations (Cont.)

In this application note, it is assumed that the READY signal is produced by the address decoder. However, if a device or memory can generate the READY signal independent of the decoder, the system designer can connect it with a Three-state buffer so that it will drive the READY input whenever it is chip-selected.

The host programmer is free to choose whether to synchronize the PAC1000 with slow devices via single operand transfers or through the READY mechanism. READY is always considered when the PAC1000 generates an explicit address. The selection between single operand transfer and READY is done in the command word (see Appendix 2).

As seen in Figure 6 there are several spare pins, such as output controls, I/Os, interrupts and condition codes. These pins can be used to perform other operations in parallel (unrelated to the DMA controller function), without any performance degradation of the DMA task.

PAC1000 Internal Resources Usage

Using PAC1000 as a 4-channel DMA controller utilizes most of the resources available on the chip, shown in Figure 1.

The Host microprocessor uses the FIFO to program the DMA request in to PAC1000. Internal condition codes are used to monitor FIFO status, CPU operation flags and external condition code inputs are used to monitor situations like bus-grant, DMA requests by the devices, etc. The CPU registers are used to store source and destination addresses, device numbers and other relevant information about the DMA transfers in progress.

To achieve the fastest transfer rate possible with PAC1000, address generation and block size counting are achieved by different methods depending on the type of transfer. For example, for the Device-Memory fly-by transfers, a nested loop is set up using the loop counter and the stack for maintaining block count and ACH and ACL are used as independent registers for address generation. On the other hand, for the memory to memory transfers, Block counter is used for counting and address generation is done by using ACH and ACL as 22-bit counter.

The IOR is used to output chip selects to the devices. The OUTCTL lines are used to generate Read and Write signals and also used for generating hand-shake signals to the host.

The data bus and associated CPU registers are used to read data in and out of PAC1000 for non-fly-by transfers.

Software Considerations

All the algorithms described so far are internally realized by Software. Flowcharts and partial code implementation (of all the important transfer procedures referred to in the flowcharts) can be found in Appendix 3 and Appendix 4, respectively. Both flowcharts and code listings contain sufficient explanations that let the reader understand the subjects they describe. The attached code listings cover all the important DMA transfer procedures (see Appendix 4).

Conclusion

PAC1000 is perfectly suitable for any DMA transfers which require an intelligent processor that can adapt its data handling according to the changing requirements of its interface. The PAC1000 does so by properly exploiting its unique structure of a very high speed sequencer combined with a programmable ALU and user configurable ports. The PAC1000's programmability enables it to handle complex tasks concurrently in a very efficient manner, unlike all other existing DMA controllers that are restricted to perform in a predefined environment.

Appendix 1:

The PAC1000 is configured in this application note as a generic DMA controller. It has a separate 24-bit address (that can be easily expanded) and a 16-bit data bus. It also has a set of control signals to enable operation as a bus master or a bus slave. The following table defines the individual PAC1000 pins. These brief descriptions are provided for reference only. Each signal is further detailed within the sections that describe the associated DMA function. For pin identifications refer to Figure 6.

Pin Descriptions

Symbol	Type	Name and Function
A(23:22)	O	**Address Lines A(23:22):** Output the two most significant address lines during Master operations. Tied to IO(5:4) on the PAC1000. Float in Slave Mode.
A(21:6)	O	**Address Lines A(21:6):** Output the mentioned address lines only in Master Mode. Connected to ADD(15:0) on the PAC1000. Float in Slave operations.
A(5:0)	I/O	**Address Lines A(5:0):** Bidirectional address lines. Input during Slave operations, output in Master mode. Tied to HAD(5:0) on the PAC1000.
FBRW2 FBRW1	O O	**Fly-by Read/Write (2:1):** Enable fly-by DMA operations. In fly-by mode, operands are transferred directly from the source to the destination bypassing the DMA controller. FBRW2 and FBRW1 are tied to OC6 and OC5, respectively.

FBRW2	FBRW1	
0	0	— Normal operation.
0	1	— Enable fly-by from memory to device.
1	0	— Enable fly-by from device to memory.
1	1	— Illegal.

$\overline{WR}$	I	**Write:** Active as an input, only in Slave Mode. When low, HD(15:0) is written into the PAC1000.
$\overline{RD}$	I	**Read:** Active as an input, only in Slave Mode. When low, HD(15:0) is driven by the PAC1000.
$\overline{WRM}$	O	**Write-Out:** Active as an output, only in Master Mode. Enabled by BUSMSTR signal. Tied to OC4 on the PAC1000.
$\overline{RDM}$	O	**Read-Out:** Active as an output, only in Master Mode. Enabled by BUSMSTR signal. Tied to OC3 on the PAC1000.
$\overline{DMACK(3:0)}$	O	**DMA Acknowledge (3:0):** 4 active low signals High in Slave Mode. Correspond to the 4 devices shown in Figure 6 respectively. Chip select the active devices during DMA operations. In the PAC1000 they are tied to IO(3:0) lines.
BUSMSTR	O	**Bus-Master:** An active high signal. Asserted whenever the PAC1000 is the current Bus Master. Informs arbiters or hosts not to access the bus before the PAC1000 relinquishes it. Enables OC4 and OC3 into $\overline{WR}$ and $\overline{RD}$, respectively. Connected to OC2 on the PAC1000.
$\overline{CSPAC}$	I	**PAC1000 Chip Select:** This pin is driven low whenever the PAC1000 is addressed in a slave bus read or write cycle.
$\overline{BR}$	O	**Bus Request:** The PAC1000 drives this pin low whenever it requests the bus due to pending DMA requests.

Appendix 1 (Cont.)

Pin Descriptions (Cont.)

Symbol	Type	Name and Function
HOSTINTR	O	**Host Interrupt:** The PAC1000 interrupts the host in order to inform him of one of the following events: PAC1000 exception, Terminal-Count or DMA aborted by a device. The OC0 line is assigned to this signal.
CLK	I	**Clock:** 20 MHz clock input to the PAC1000. It also latches the condition codes to ensure the proper Set-up time.
CC7	I	**DMA Abort:** An active-high input driven by the memories and/or devices currently participating in the DMA process. Whenever it is sensed high, the PAC1000 will generate a HOSTINTR signal towards the host after writing into the DOR register the appropriate status word.
CC6	I	**Bus Grant:** An active-low signal monitored by the PAC1000 to determine when it is in the Master mode or when to relinquish the buses and enter the Slave Mode.
CC4	I	**Ready:** An active-high signal (RDY) that enables the PAC1000 to synchronize its DMA cycles with slow memories or devices in the Master Mode.
CC(3:0)	I	**DMA Requests (3:0):** External DMA requests monitored by the PAC1000. Active-high signals, driven by the four devices.
$\overline{\text{DMAWORD}}$	O	**DMA Word or Byte Transfers:** Determines whether the next DMA cycle will be of word (low) or byte (high) length. Used to derive the $\overline{\text{BHE}}$ (Bus High Enable) signal that enables data lines D15:D8 in the Master Mode. $\overline{\text{BLE}}$ is directly driven by the A0 address line.
$\overline{\text{RSPAC}}$	I	**Reset PAC1000:** This asynchronous input initializes the state of. PAC1000. RESET must be held low for at least two clock cycles.
$\overline{\text{D(15:0)}}$	I/O	**Data-Bus (15:0):** This is the 16-bit data bus. During Master cycles, it is controlled and sometimes also driven by the PAC1000. In Slave mode the host drives it. Tied to HD(15:0) on the PAC1000.

Appendix 2: Host-DMA Message Formats

1) Host to PAC1000 Commands (via the FIFO)

HD(15:0) CONTENT	HAD(5:0) CONTENT					
	HAD5	HAD4	HAD3	HAD2	HAD1	HAD0
Word 1: Command word (see paragraph 3).	0	0	0	0	0	0
Word 2: 16 low-order source address lines.	0	0	0	0	0	1
Word 3: 8 high-order source address lines.	0	0	0	0	1	0
Word 4: 16 low-order destination address lines.	0	0	0	0	1	1
Word 5: 8 high-order destination address lines.	0	0	0	1	0	0
Word 6: 16 bit block-count.	0	0	0	1	0	1
Word 7: 8 bit DMA-transfer identification byte.	0	0	0	1	1	0
Word 8: Spare.	0	0	0	1	1	1

2) PAC1000 to Host Status Word (via DOR register)

b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00

b15,b14,b13,b12,b11,b10,b9,b8: DMA-transfer identification byte.
b7,b6,b5,b4: spare.
b3: Reject or accept the DMA transfer identified by b15 ÷ b8. 1 — reject. 0 — accept.
b2: 1 — PAC1000 aborted. 0 — Normal operation
b1: 1 — DMA terminal-count completed 0 — Normal operation
b0: 1 — PAC1000 exception occurred 0 — Normal operation

Appendix 2 (Cont.)

3) Command Word Format

b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00

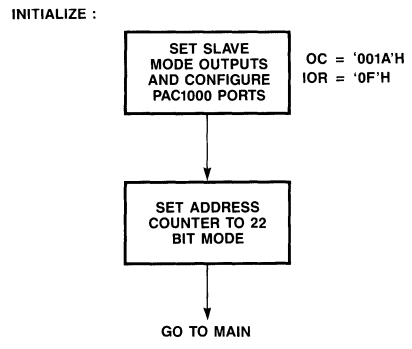
b15,b14:	spare.
b13:	block transfer or single transfer mode. 1 — DMA block operation. 0 — DMA single operand transfer mode.
b12:	DMA abort bit. Quits DMA-transfer specified in word 7. 1 — abort. 0 — nop.
b10,b9:	Priority level of this DMA-transfer. 00 — level 0 (lowest priority level). 01 — level 1 02 — level 2 03 — level 3 (highest priority level).
b9,b8:	Source Device number for DMA transfer or Abort. 00 — Device #0 01 — Device #1 02 — Device #2 03 — Device #3
b7,b6:	Dest. Device number for DMA transfer or Abort. 00 — Device #0 01 — Device #1 02 — Device #2 03 — Device #3
b5,b4:	Destination data bus definition. 00 — Data bus is D7–D0 (bit bits). 01 — Data bus is D15–D8 (8 bits). 02 — Data bus is D15–D0 (16 bits). 03 — Illegal.
b3,b2:	Source data bus definition. 00 — Data bus is D7–D0 (8 bits). 01 — Data bus is D15–D8 (8 bits). 02 — Data bus is D15–D0 (16 bits). 03 — Illegal.
b1,b0:	DMA transfer mode. 00 — Memory to memory. 01 — Memory to device. 02 — Device to device. 03 — Device to memory.

Appendix 3

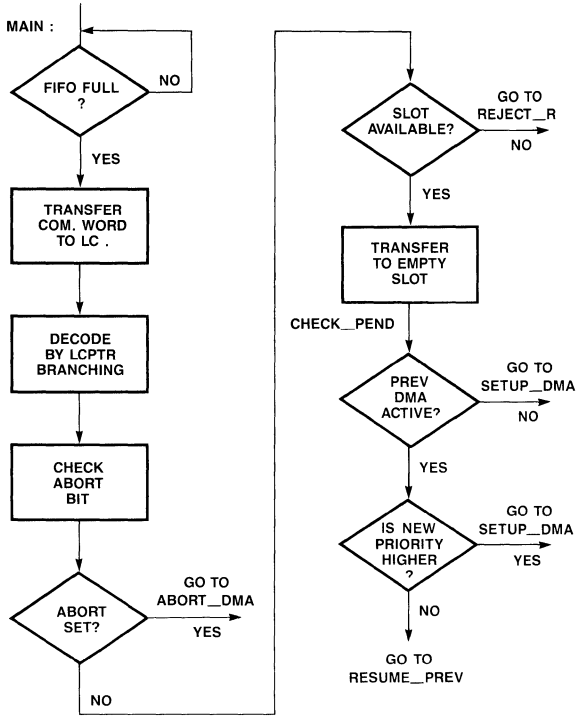
General Note:

Code implementation of labels marked with an asterisk (*) can be found in Appendix 4.

Initialization



Main Loop

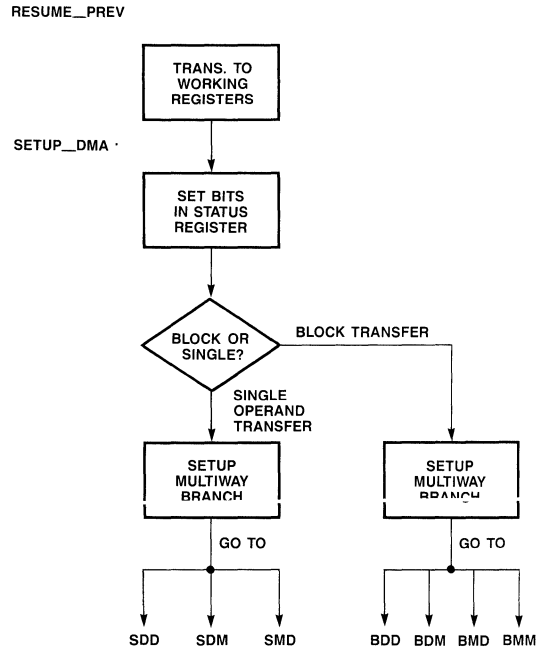


Legend:

1. **Slot:** The PAC1000 can handle up to 4 DMA channels concurrently. Slot means empty register space inside the PAC1000 that is allocated for a pending channel.
2. **LCPTR branching:** A goto instruction of the command section, enabling multi-way branching of the program according to a value loaded into the LC register by the ALU (executed in two cycles).

Appendix 3 (Cont.)

Setting Up the Transfer



Legend:

1. SDD — single operand transfer, device to device.
2. SDM — single operand transfer, device to memory.
3. SMD — single operand transfer, memory to device.
4. BDD — block transfer, device to device.
5. BDM — block transfer, device to memory.
6. BMD — block transfer, memory to device.
7. BMM — block transfer, memory to memory.

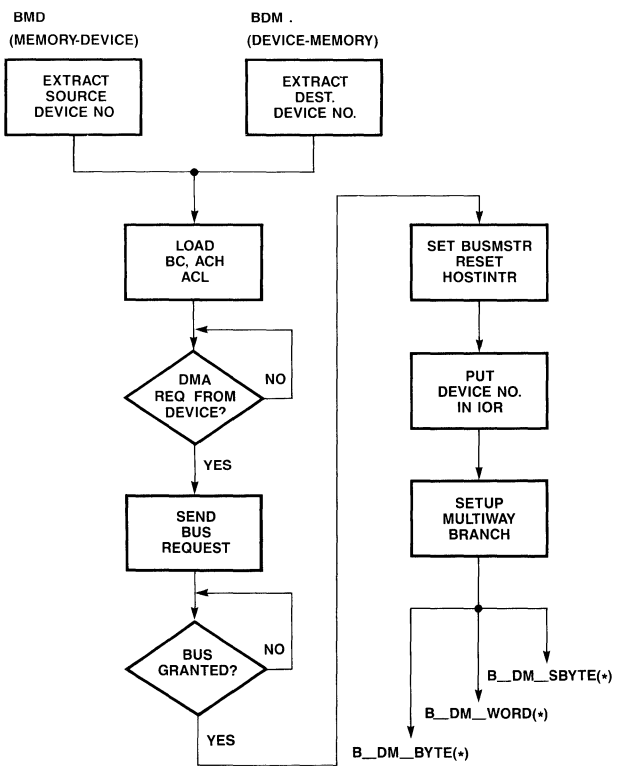
General Remarks:

In a single operand transfer, at least one of the involved devices requests a DMA transfer for each operand. This method is used with slow devices.

Block transfers are used to move data blocks between fast memories and/or devices. A DMA request is set for every block transfer.

Appendix 3 (Cont.)

Device to Memory Block Transfer



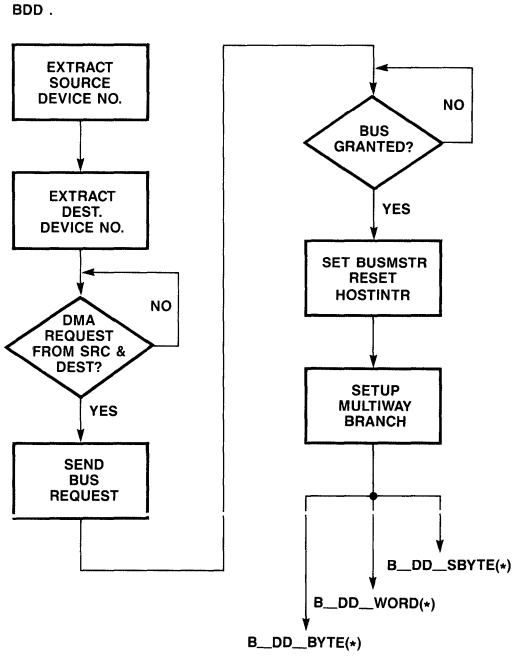
Legend:

1. B_dm_byte: block device to/from memory transfer of bytes.
2. B_dm_word: block device to/from memory transfer of words.
3. B_dm_sbyte: block device to/from memory transfer of swapped bytes. Occurs whenever the transfer is between even and odd addresses.

4

Appendix 3 (Cont.)

Device to Device Block Transfer

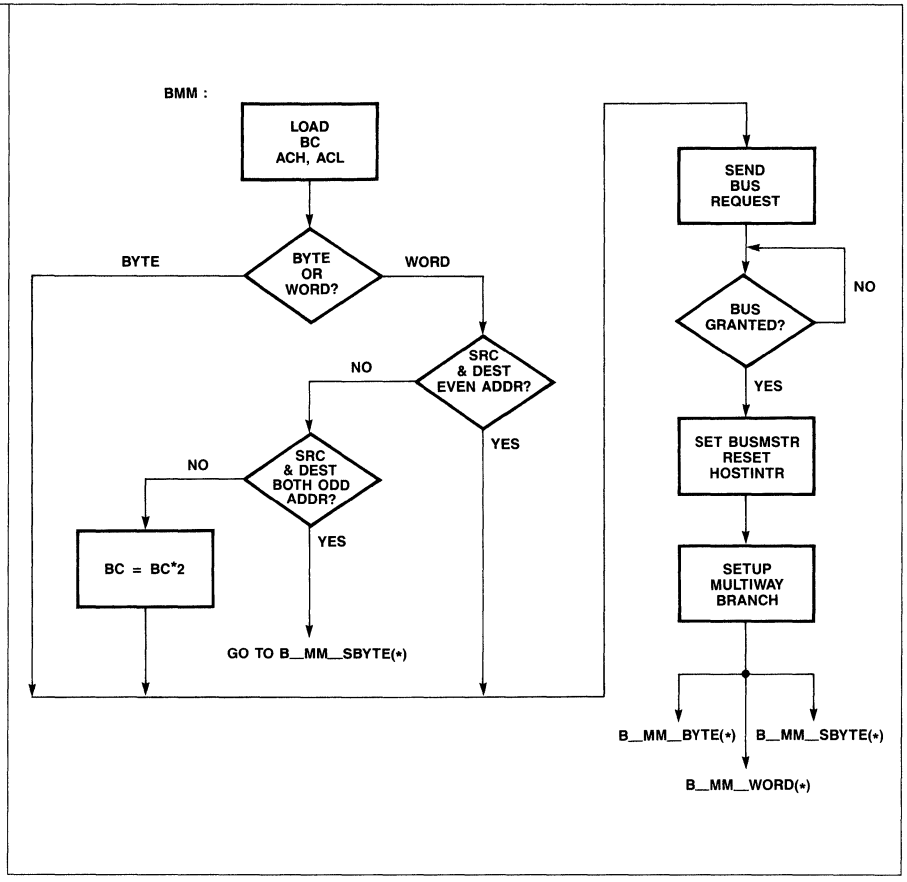


Legend:

1. B_dd_byte: block device to device transfer of bytes.
2. B_dd_word: block device to device transfer of words.
3. B_dd_sbyte: block device to device transfer of swapped bytes. Happens whenever the transfer is between even and odd addresses.

Appendix 3 (Cont.)

Memory to Memory Block Transfer



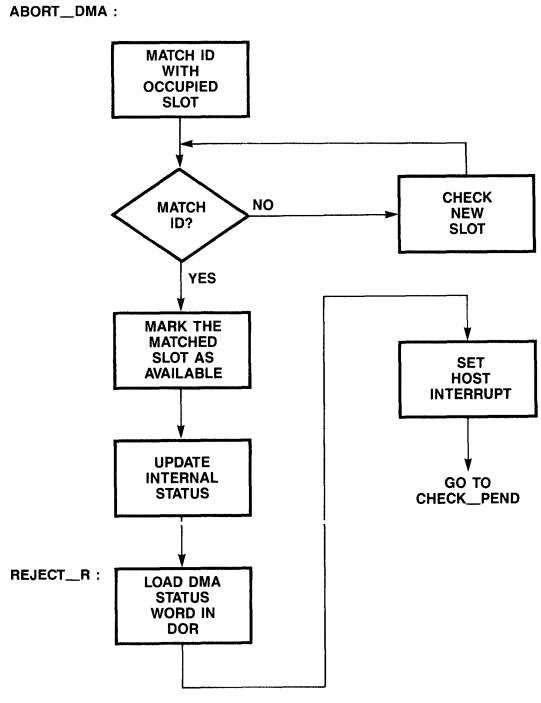
Legend:

1. B_mm_byte: block memory to memory transfer of bytes.
2. B_mm_word: block memory to memory transfer of words.
3. B_mm_sbyte: block memory to memory transfer of swapped bytes. Occurs whenever the transfer is between even and odd addresses.

4

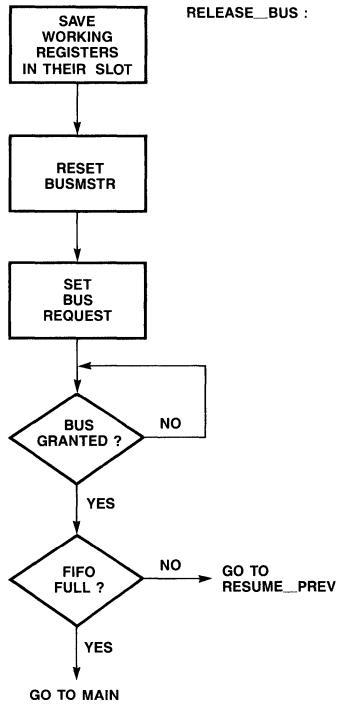
Appendix 3 (Cont.)

Abort DMA Transfer

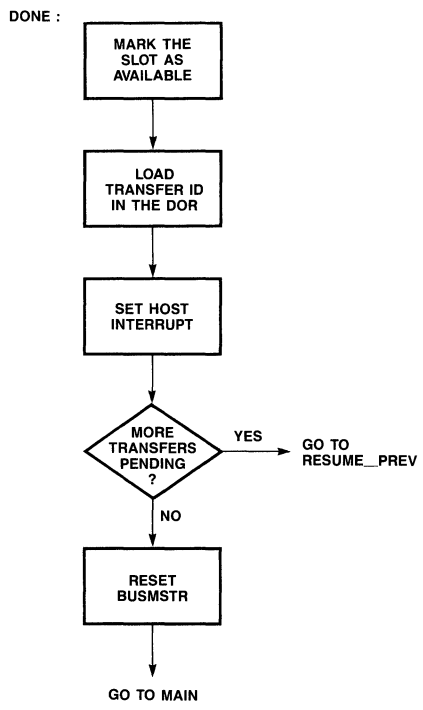


Appendix 3 (Cont.)

Bus Release



End of Transfer



4

Appendix 4

```
/*************************************************************************/
/* device to memory byte transfer in the fly-by mode. The start address */
/* of the memory is loaded in  R3 and R4 and the device number is loaded*/
/* in Q . Assume that the initial protocol has been gone through and     */
/* PAC has  control of the bus.    For simplicity it is assumed that the*/
/* block size is a multiple of 64 and R5*64 = block size.                */
/*************************************************************************/
segment b_dm_byte ;
     /* define equates */
     bgn equ CC7                ;  /* bus grant (active low)      */
     ready equ CC4              ;  /* ready input                 */
     b_dm_byte_norm equ h'00de' ;  /* dma active w/o read/write   */
     b_dm_byte_read equ h'00d6' ;  /* read (active low )          */
     b_dm_byte_write equ h'00c6';  /* write (active low )         */
init_b_dm_byte :
     ACH := R3                  ;  /* upper 16 bits address       */
     SET ASEL ADOE HADOE  ,
     ACL := R4                  ;  /* select counter to output ,
                                   enable ADD and HAD output, and
                                   load lower address in ACL     */
     IOR := ~ Q ,
     OUT b_dm_byte_norm         ;  /* select device #        */
     Q := 1                     ;  /* address increment for byte    */
     LDLCD , MOV R5 R5          ;  /* R5 * 64 -> block count        */
/*************************************************************************/
/* start of outer transfer loop                                          */
/*************************************************************************/
x1:  PLDLC H'3F'                ;  /* push cnt to stack and load 64
                                   in cnt                 */
/*************************************************************************/
/* start inner transfer loop                                             */
/*************************************************************************/
y1 : JMPNC ready y1 ,
     OUT b_dm_byte_read         ;   /* wait till ready signal high    */
     LOOPNZ y1     ,
     ACL := ACL + Q ,
     OUT b_dm_byte_write        ;   /* strobe the write signal and
                                   set up the next address      */
/*************************************************************************/
/* end inner loop                                                        */
/*************************************************************************/
     POPLC    ,
     ACH := ++ ACH ,
     OUT b_dm_byte_read         ;  /* pop stack to cnt , increment
                                   upper address bits       */
     JMPC bgn release_bus       ;  /* check if bus grant has been
                                   taken away               */
     LOOPNZ x1                  ;  /* loop back if counter not zero*/
/*************************************************************************/
/* end outer loop                                                        */
/*************************************************************************/
done :
     ...................
     ...................
release_bus :
     ...................
/*************************************************************************/
```

Appendix 4 (Cont.)

```
/*************************************************************************/
/* device to memory word transfer in the fly-by mode. The start address */
/* of the memory is loaded in  R3 and R4 and the device number is loaded*/
/* in Q . For simplicity it is assumed that the block size is a multiple*/
/* of 64 and R5*64 = block size.                                         */
/*************************************************************************/
segment b_dm_word ;
     /* define equates */
     bgn equ CC7            ; /* bus grant (active low)      */
     ready equ CC4          ; /* ready input                 */
     b_dm_word_norm equ h'00de' ;  /* dma active w/o read/write  */
     b_dm_word_read equ h'00d6' ;  /* read (active low )         */
     b_dm_word_write equ h'00c6';  /* write (active low )        */
init_b_dm_word :
     ACH := R3                  ;  /* upper 16 bits address      */
     SET ASEL ADOE HADOE  ,
     ACL := R4                  ;  /* select counter to output ,
                                   enable ADD and HAD output, and
                                   load lower address in ACL    */
     IOR := ~ Q ,
     OUT b_dm_word_norm         ;  /* select device #       */
     Q := 2                     ;  /* address increment for word   */
     LDLCD , MOV R5 R5          ;  /* R5 * 64 -> block size (words)*/
/*************************************************************************/
/* start of outer transfer loop                                          */
/*************************************************************************/
x1:  PLDLC H'1F'                ;  /* push cnt to stack and load 32
                                   in cnt                 */
/*************************************************************************/
/* start inner transfer loop                                             */
/*************************************************************************/
y1 : JMPNC ready y1 ,
     OUT b_dm_word_read         ;   /* wait till ready signal high    */
     LOOPNZ y1     ,
     ACL := ACL + Q ,
     OUT b_dm_word_write        ;   /* strobe the write signal and
                                   set up the next address      */
/*************************************************************************/
/* end inner loop                                                        */
/*************************************************************************/
     POPLC    ,
     ACH := ++ ACH ,
     OUT b_dm_word_read         ;  /* pop stack to cnt , increment
                                   upper address bits       */
     JMPC bgn release_bus       ;  /* check if bus grant has been
                                   taken away               */
     LOOPNZ x1                  ;  /* loop back if counter not zero*/
/*************************************************************************/
/* end outer loop                                                        */
/*************************************************************************/
done :
     ...................
     ...................
release_bus :
     ...................
/*************************************************************************/
```

Appendix 4 (Cont.)

```
/*************************************************************************/
/* device to memory byte transfer in the fly-by mode. The start address */
/* of the memory is loaded in  R3 and R4 and the device number is loaded*/
/* in Q . For simplicity it is assumed that the block size is a multiple*/
/* of 64. This code illustrates individual transfer mode(non-block mode)*/
/*************************************************************************/
segment s_dm_byte ;
     /* define equates */
     bgn equ CC7              ; /* bus grant (active low)      */
     ready equ CC4            ; /* ready input                 */
     s_dm_byte_norm equ h'00de' ;  /* dma active w/o read/write    */
     s_dm_byte_read equ h'00d6' ;  /* read (active low )          */
     s_dm_byte_write equ h'00c6';  /* write (active low )         */
init_s_dm_byte :
     ACH := R3                ;  /* upper 16 bits address        */
     SET ASEL ADOE HADOE  ,
     ACL := R4                ;  /* select counter to output ,
                                    enable ADD and HAD output, and
                                    load lower address in ACL    */
     BC := R5                 ;  /* load block size in to BC    */
     IOR := ~ Q ,
     OUT s_dm_byte_norm       ;  /* select device #             */
     CMP Q H'0001'          ;      /* find out if device #0        */
     JMPC Z dev0            ;
     CMP Q H'0002'          ;      /* if device # 1               */
     JMPC Z dev1            ;
     CMP Q H'0004'          ;      /* if device # 2               */
     JMPC Z dev2            ;
                                   /* else it is device # 3
/*************************************************************************/
/* start  transfer loop for dev#3                                       */
/*************************************************************************/
dev3 :
     JMPC bgn release_bus       ;  /* monitor bus grant            */
     JMPNC CC3 dev3       ,
     OUT s_dm_byte_read       ;   /* branch to check for dma request
                                    from  device3                */
     SET ACEN BCEN ,
     OUT s_dm_byte_write    ;      /* start counter               */
     RESET ACEN BCEN ,
     OUT s_dm_byte_norm     ;       /* stop counter                */
     JMPNC BCZ dev3               ;  /* loop back if not done       */
     JMP done          ;

/*************************************************************************/
/* start  transfer loop for dev#2                                       */
/*************************************************************************/
dev2 :
     JMPC bgn release_bus       ; /* monitor bus grant            */
     JMPNC CC2 dev2       ,
     OUT s_dm_byte_read       ;  /* branch to check for dma request
                                    from  device2                */
     SET ACEN BCEN ,
     OUT s_dm_byte_write      ;  /* start counter               */
```

Appendix 4 (Cont.)

```
     RESET ACEN BCEN ,
     OUT s_dm_byte_norm          ;  /* stop counter                */
     JMPNC BCZ dev2              ;  /* loop back if not done */
     JMP done                    ;
/*************************************************************************/
/* start  transfer loop for dev#1                                        */
/*************************************************************************/
dev1 :
     JMPC bgn release_bus        ;    /* monitor bus grant             */
     JMPNC CC1 dev1      ,
     OUT s_dm_byte_read          ;     /* branch to check for dma request
                                        from  device1                  */
     SET ACEN BCEN ,
     OUT s_dm_byte_write         ;    /* start counter       */
     RESET ACEN BCEN ,
     OUT s_dm_byte_norm          ;    /* stop counter              */
     JMPNC BCZ dev1              ;    /* loop back if not done     */
     JMP done          ;
/*************************************************************************/
/* start  transfer loop for dev#0                                        */
/*************************************************************************/
dev0 :
     JMPC bgn release_bus        ;    /* monitor bus grant             */
     JMPNC CC3 dev0      ,
     OUT s_dm_byte_read          ;     /* branch to check for dma request
                                        from  device3                  */
     SET ACEN BCEN ,
     OUT s_dm_byte_write    ;    /* start counter      */
     RESET ACEN BCEN ,
     OUT s_dm_byte_norm     ;    /* stop counter            */
     JMPNC BCZ dev0         ;    /* loop back if not done   */

/*************************************************************************/
done :
     ......................
     ......................
release_bus :
     ......................
/*************************************************************************/
```

Appendix 4 (Cont.)

```
/*************************************************************************/
/* code to illustrate device to memory transfer in non fly by mode .     */
/* This is used when data bus is connected d7-d0 to d15-d8  or the       */
/* other way around. Use counter to output addresses.Q contains device   */
/* number and R3 R4 contain destination address.R5 contains block size. */
/*************************************************************************/
segment b_dm_sbyte ;
     /* define equates */
     b_dm_sbyte_norm equ h'009e' ;
     b_dm_sbyte_read equ h'0096' ;
     b_dm_sbyte_write equ h'008e';
     rdy equ CC4              ;
     bgn equ CC7              ;
init_b_dm_sbyte :
     BC := R5 ,
     OUT b_dm_sbyte_norm     ;    /* load block size in bcnt     */
     SET DIREN ASEL HADOE ADOE   ;/* select  counter to output ,
                                    enable had output           */
     ACH := R3          ;
     ACL := R4          ;
/*************************************************************************/
/* start of transfer loop                                                */
/*************************************************************************/
b_dm_sbyte :
     JMPC bgn release_bus        ;
     SET DIREN                   ; /* enable DIR                  */
srdy :
     JMPNC rdy  srdy ,
     OUT b_dm_sbyte_read         ; /* wait till source ready     */
     SET HDOE HDSEL0 ,
     AOR := DIR              ;     /* when src is ready read the data
                                    in , enable HD output , select
                                    DOR to output            */
     DOR := SWPV    ,
     OUT b_dm_sbyte_write         ;   /* put swapped data in DOR      */
     SET ACEN BCEN ,
     OUT b_dm_sbyte_norm     ;     /* start counter , output swapped
                                    data                     */
     RESET ACEN BCEN HDOE        ;
     JMPNC BCZ b_dm_sbyte        ;
/*************************************************************************/
/* end of transfer loop                                                  */
/*************************************************************************/
done :
     ..................
release_bus :
     ..................
/*************************************************************************/
```

Appendix 4 (Cont.)

```
/*************************************************************************/
/* code to illustrate memory to memory transfer.Use counter to output   */
/* both addresses.R1,R2 contain source address and R3 R4 contain dest    */
/* address . R5 contains block size.                                     */
/*************************************************************************/
segment b_mm_byte ;
     /* define equates */
     b_mm_byte_norm equ h'009e' ;
     b_mm_byte_read equ h'0096' ;
     b_mm_byte_write equ h'008e';
     rdy equ CC4              ;
     bgn equ CC7              ;
init_b_mm_byte :
     BC := R5 ,
     OUT b_mm_byte_norm      ;     /* load block size in bcnt */
     SET ASEL HADOE ADOE          ;    /* select  counter to output ,
                                     enable had output            */
/*************************************************************************/
/* start of transfer loop                                                */
/*************************************************************************/
b_mm_byte :
     JMPC bgn release_bus ,
     ACH := R1          ;           /* monitor bus grant , source
                                     address in R1        */
     SET DIREN , ACL := R2       ;  /* enable dir, r2 <- low 6 bits */
srdy :
     JMPNC rdy  srdy ,
     OUT b_mm_byte_read      ;      /* wait till source ready     */
     SET ACEN HDOE HDSEL0 ,
     DOR := DIR              ;      /* when src is ready read the data
                                     in , enable HD output , select
                                     DOR to output            */
     RESET ACEN DIREN ,R1 := ACH,
     OUT b_mm_byte_norm      ;      /* stop counter , store it back in
                                     to registers             */
     ADD R2 ACL Q ARDREG ACH R3 ;   /* mov ACL back to r1 and at the
                                     same time load r3 to ach     */
     ACL := R4                  ;   /* ach,acl have dest address    */
drdy :
     JMPNC rdy drdy             ;   /* wait for destination ready   */
     SET ACEN BCEN ,
     OUT b_mm_byte_write         ;  /* when dest is ready , write the
                                     data, increment counter , also
                                     enable block counter     */
     RESET ACEN BCEN HDOE ,
     R3 := ACH ,
     OUT b_mm_byte_norm      ;      /* stop counters , set HD to input
                                     save dest address (upper 16) */
     JMPNC BCZ b_mm_byte ,
     R4 := ACL          ;           /* loop back if block counter not
                                     zero , also save lower 6 bits
                                     of dest address          */
/*************************************************************************/
/* end of transfer loop                                                  */
/*************************************************************************/
done :
     ....................
release_bus :
     ....................
/*************************************************************************/
```

Appendix 4 (Cont.)

```
/*************************************************************************/
/* code to illustrate memory to memory transfer (word mode).Use counter */
/* to output both addresses.R1,R2 contain source address and R3 R4       */
/* contain destination address . R5 contains block size in words.        */
/*************************************************************************/
segment b_mm_word ;
     /* define equates */
     b_mm_word_norm equ h'009e' ;
     b_mm_word_read equ h'0096' ;
     b_mm_word_write equ h'008e';
     rdy equ CC4              ;
     bgn equ CC7              ;
init_b_mm_word :
     BC := R5 ,
     OUT b_mm_word_norm          ;    /* load block size in bcnt  */
     SET ASEL HADOE ADOE         ;    /* select  counter to output ,
                                    enable had output            */
/*************************************************************************/
/* start of transfer loop                                                */
/*************************************************************************/
b_mm_word :
     JMPC bgn release_bus ,
     ACH := R1                   ;   /* monitor bus grant , source
                                     address in R1        */
     SET DIREN , ACL := R2       ;   /* enable dir,ACL <- low 6 bits */
srdy :    JMPNC rdy  srdy ,
     OUT b_mm_word_read          ;   /* wait till source ready     */
     SET ACEN HDOE HDSEL0 ,
     DOR := DIR                  ;   /* when src is ready read the data
                                     in , enable HD output , select
                                     DOR to output              */
     OUT b_mm_word_norm        ;
     RESET ACEN DIREN ,
     ADD R1 ACH Q ARDREG ACH R3 ;   /* stop counter , store ACH in to
                                     R1 and also load ACH with R3 */
     ADD R2 ACL Q ARDREG ACL R4 ;   /* store ACL in R2 and at the same
                                     time put R4 in to ACL    */
drdy :    JMPNC rdy drdy         ;  /* wait for destination ready   */
     SET ACEN BCEN ,
     OUT b_mm_word_write         ;  /* when dest is ready , write the
                                     data, increment counter , also
                                     enable block counter     */
     RESET BCEN HDOE ,
     OUT b_mm_word_norm          ;  /* stop block counter, set HD to
                                     input                      */

     RESET ACEN  ,  R3 := ACH  ;    /* stop add counter ,
                                     save dest address (upper 16) */
     JMPNC BCZ b_mm_word ,
     R4 := ACL                 ;    /* loop back if block counter not
                                     zero , also save lower 6 bits
                                     of dest address          */
/*************************************************************************/
/* end of transfer loop                                                  */
/*************************************************************************/
done :
     ..................
release_bus :
     ..................
/*************************************************************************/
```

Appendix 4 (Cont.)

```
/*************************************************************************/
/* code to illustrate memory to memory transfer  from D7-D0 to D15-D8    */
/* or vice-versa. Use counter to output both addresses .R1 , R2 contain */
/* source address and R3 R4 contain destination address.R5 contains      */
/* block size. Data is read in to AOR and byte-swpped before outputting */
/* through DOR.                                                          */
/*************************************************************************/
segment b_mm_sbyte ;
     /* define equates */
     b_mm_sbyte_norm equ h'009e' ;
     b_mm_sbyte_read equ h'0096' ;
     b_mm_sbyte_write equ h'008e';
     rdy equ CC4              ;
     bgn equ CC7              ;
init_b_mm_sbyte :
     BC := R5,OUT b_mm_sbyte_norm;     /* load block size in bcnt */
     SET ASEL HADOE ADOE         ;     /* select  counter to output ,
                                          enable had output          */
/*************************************************************************/
/* start of transfer loop                                                */
/*************************************************************************/
b_mm_sbyte :
     JMPC bgn release_bus ,
     ACH := R1          ;           /* monitor bus grant , source
                                    address in R1        */
     SET DIREN , ACL := R2       ;  /* enable dir, r2 <- low 6 bits */
srdy :    JMPNC rdy  srdy ,
     OUT b_mm_sbyte_read          ; /* wait till source ready     */
     SET ACEN HDOE HDSEL0 ,
     AOR := DIR              ;      /* when src is ready read the data
                                    in , enable HD output , select
                                     DOR to output              */
     RESET ACEN DIREN,R1 := ACH ,
     OUT b_mm_sbyte_norm         ;  /* stop counter , store it back in
                                    to registers               */
     ADD R2 ACL Q ARDREG ACH R3 ;   /* mov ACL back to r1 and at the
                                    same time load r3 to ach     */
     ACL := R4                   ;  /* ach,acl have dest address    */
drdy :    JMPNC rdy drdy,DOR := SWPV ; /* wait for destination ready
                                    and write swapped value       */
     SET ACEN BCEN ,
     OUT b_mm_sbyte_write         ; /* when dest is ready , write the
                                    data, increment counter , also
                                    enable block counter     */
     RESET ACEN BCEN HDOE ,
     R3 := ACH ,
     OUT b_mm_sbyte_norm          ; /* stop counters , set HD to input
                                    save dest address (upper 16) */
     JMPNC BCZ b_mm_sbyte ,
     R4 := ACL          ;           /* loop back if block counter not
                                    zero , also save lower 6 bits
                                    of dest address          */
/*************************************************************************/
/* end of transfer loop                                                  */
/*************************************************************************/
done :
     ...................
release_bus :
     ...................
/*************************************************************************/
```

Appendix 4 (Cont.)

```
/*************************************************************************/
/* code to illustrate device to device transfers in the byte as well as */
/* word mode.  source device is in r1 and dest device is in r3. block   */
/* size is in r5.                                                        */
/*************************************************************************/
segment b_dd_bw ;
     /* define equates */
     b_dd_bw_norm equ h'009e'  ;
     b_dd_bw_read equ h'0096'  ;
     b_dd_bw_write equ h'008e' ;
     rdy equ CC4            ;
     bgn equ CC7            ;
init_b_dd_bw :
     SET DIREN , IOR := ~ R1 ,
     OUT b_dd_bw_norm          ;    /* enable DIR and output source
                                    device chip select        */
/*************************************************************************/
/* start of transfer loop                                                */
/*************************************************************************/
b_dd_byte :
b_dd_word :
b_dd_bw :
     JMPC bgn release_bus ,
     IOR := ~ R3 ,
     OUT b_dd_bw_read       ;  /* read source device and output
                               dest device chip select , also
                               monitor bus grant         */
     SET  HDOE HDSEL0 ,
     DOR := DIR ,
     OUT b_dd_bw_norm       ;  /* enable HD output , select DOR
                               to output                 */

     RESET HDOE ,
     DEC R5 ,
     OUT b_dd_bw_write      ;  /* HD to input , decrement count ,
                               output write strobe       */
     JMPNC Z b_dd_bw ,
     IOR := ~ R1 ,
     OUT b_dd_bw_norm       ;  /* loop back if R5 not zero , also
                               output src device cs         */
/*************************************************************************/
/* end of transfer loop                                                  */
/*************************************************************************/
done :
     ..................
     ..................
release_bus :
     ..................
/*************************************************************************/
```

Appendix 4 (Cont.)

```
/*************************************************************************/
/* code to illustrate device to device transfer in non fly by mode .     */
/* This is used when data bus is connected d7-d0 to d15-d8  or the       */
/* other way around. Source device # is in R1 and dest device # in R3    */
/*************************************************************************/
segment b_dd_sbyte :
     /* define equates */
     b_dd_sbyte_norm equ h'009e' ;
     b_dd_sbyte_read equ h'0096' ;
     b_dd_sbyte_write equ h'008e';
     rdy equ CC4              ;
     bgn equ CC7              ;
init_b_dd_sbyte :
     SET DIREN , IOR := ~ R1 ,
     OUT b_dd_sbyte_norm         ; /* enable DIR and output source
                                   device chip select       */
/*************************************************************************/
/* start of transfer loop                                                */
/*************************************************************************/
b_dd_sbyte :
     JMPC bgn release_bus ,
     IOR := ~ R3 ,
     OUT b_dd_sbyte_read         ; /* read source device and output
                                    dest device chip select , also
                                    monitor bus grant       */
     AOR := DIR             ;     /* read in the data        */
     SET  HDOE HDSEL0 ,
     DOR := SWPV ,
     OUT b_dd_sbyte_write        ; /* enable HD output , select DOR
                                   to output , put swapped data in
                                   DOR                     */
     RESET HDOE ,
     DEC R5 ,
     OUT b_dd_sbyte_norm         ; /* HD to input , decrement count ,
                                   output write strobe      */
     JMPNC Z b_dd_sbyte ,
     IOR := ~ R1                 ; /* loop back if R5 not zero , also
                                   output src device cs         */
/*************************************************************************/
/* end of transfer loop                                                  */
/*************************************************************************/
done :
     ..................
     ..................
release_bus :
     ..................
/*************************************************************************/
```

Programmable Peripheral Application Brief 006

PAC1000 as a 16 Bi-Directional Serial Channel Controller

By Arye Ziklik

Introduction

This Application Brief describes a Communications Controller that utilizes the PAC1000 as the board level control element in a 16 bi-directional serial channel board. The aggregate board throughput is around *1 Mbyte/sec.*

Serialization and de-serialization of the data is handled by eight Serial Communication Controllers (SCC). Every SCC has two bi-directional serial channels with individual baud rate generator and digital phase loop mechanism. The SCC can handle all the customary synchronous and asynchronous protocols as well as the popular serial data encoding/decoding schemes. With a 16-MHz clock, the maximum bit rate in every individual channel can be up to 2 Mbps.

PAC1000 — Host Interface

The PAC1000 performs the low level function of moving the data to and from the serial devices and buffer RAM memory. The host interface is a generic 32-bit system. The host processor communicates with the PAC1000 through two interrupt lines, two status signals and a mail-box area that resides in the buffer memory. Prior to accessing the board, the host drives the "system board access" signal. The PAC1000 is interrupted (INT3) and relinquishes control of the board's data and address buses as long as that signal is active (as reflected by CC0). The host reads and/or writes into the buffer memory. After completion of this activity, it updates the mail-box region and then lowers the "system board access" signal. The PAC1000 continuously monitors that signal. After CC0 is negated, the PAC1000 can raise its "PAC1000-board master" signal and start controlling the data/address buses and control signals. Whenever it needs a fast response from the host, the PAC1000 updates the mail-box portion of the shared buffer memory, lowers the "PAC1000-board master" signal and activates the system interrupt.

Buffer Memory Structure

The high speed buffer memory is composed of 64K bytes of static RAM that can be accessed in three ways: by bytes (during SCC transfer operations), by words (when accessed by the PAC1000), or by double words (from the host side). Memory access configuration is determined by the PAC1000 output control signals (OC port).

The buffer memory is divided into three regions:

1) SCC control image register space that includes copies of the SCC registers.
2) Buffer message space where the 32 buffers of the corresponding serial channels are stored.
3) Mail-box area in which the PAC1000 exchanges command and status information with the host. This region also contains the pointers to the 32 channel buffers.

Whenever instructed to do so, the PAC1000 writes the image register content of a channel into the corresponding SCC, thereby initializing that channel for a particular transfer mode. Buffer message sizes are allocated by the host according to the speed of each individual channel. The pointers of the buffers are stored in the mail-box area.

Every transfer takes place between the buffer memory and the selected SCC. The PAC1000 is acting in this design as a 32-channel DMA controller, capable also of communicating with the host processor through their mail-box region. Once the board is properly configured, the only interface of the host system is the reading of data from the receive and mail-box buffers and the placing of new data into the transmit and mail-box buffers. The PAC1000 off-loads the host processor from maintaining the low level control of each channel.

PAC1000 — SCC Devices Interface

The high speed data transfers are achieved due to the very fast response of the PAC1000 to the channel service requests. The SCCs are programmed to request DMA transfers whenever they are either ready to transmit or containing new received characters.

The 16 received character DMA requests are priority encoded and latched. The encoder output is connected to the PAC1000's CC3 pin. The 16 transmit DMA requests are priority encoded and latched, too. Their encoder drives the CC2 input pin. The condition code multiplexer presents to the CC7–CC4 the highest priority encoded-channel-number of the pending receiver request, or the transmitter request, or the highest priority SCC number that is currently requesting an interrupt service via the CC1 pin. The receiver requests have higher priority over the transmitter requests. The lowest service priority is assigned to the SCC interrupts. This configuration ensures a very fast response time of the PAC1000 to DMA requests and SCC interrupts. Condition code latency is 125 ns and multi-way branching according to the CC7–CC4 lines requires additional 125 ns. Therefore, 250 ns after a high priority DMA request, the service routine will be initiated. The condition code lines CC3, CC2 and CC1 are continuously monitored by the PAC1000 during the time that it is the board master. Therefore it responds immediately when either a DMA request or an SCC interrupt is pending.

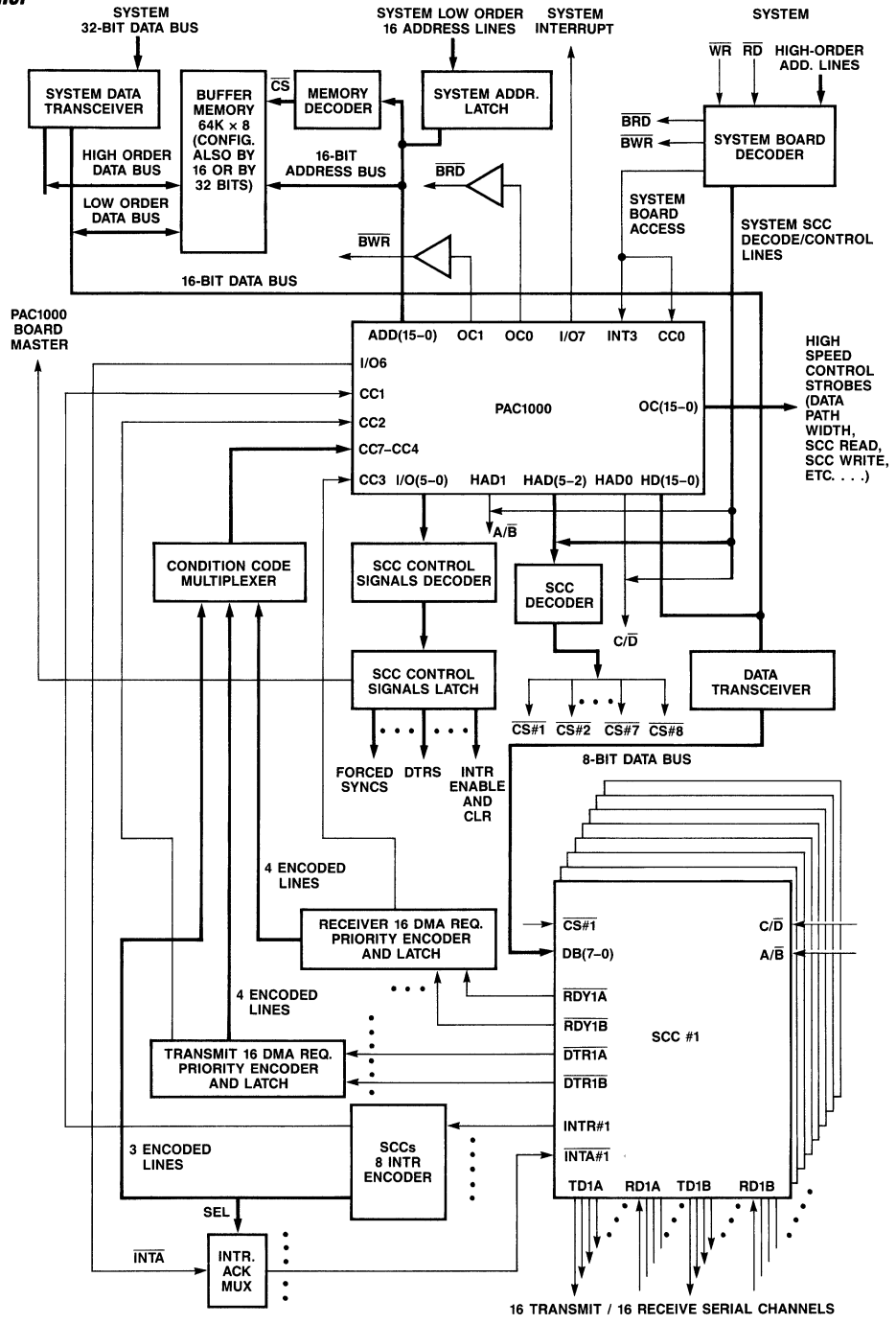
The regular SCC interrupt lines are also prioritized and latched by an 8 interrupt encoder. These interrupts are requested by erroneous SCC channels or whenever block transfers are completed. The interrupt priority encoder is also connected to the condition code multiplexer. The three encoded lines that denote the number of the serviced SCC route the INTA signal issued by the PAC1000 (via the I/O6 pin) to the corresponding SCC.

Miscellaneous

In addition to functioning as an SCC controller, the PAC1000 can also generate all the necessary signals for modem control and modem interface through the SCC control signal latch.

The PAC1000 output control (OC) port generates various control strobes such as data path width definition, read/write, multiplexer and decoder select, etc.

PAC1000 as a 16 Bi-Directional Serial Channel Controller



Programmable Peripheral Application Brief 007

Hardware Interfacing the PAC1000 as a Micro Channel Bus Controller

By Arye Ziklik

Abstract

This application brief describes how to use the PAC1000 Programmable Peripheral Controller as a Micro Channel (MCA) bus controller.

The MCA bus uses asynchronous and synchronous procedures to control and transfer data on the bus. The data is transferred from a master board to a slave board, or from the PS/2 mother-board (the system) to a slave. This application brief describes the use of the PAC1000 on a master board and on a slave board.

In both applications the PAC1000 is handling the synchronous functions, the asynchronous functions are implemented by external PALs.

MCA Signal Descriptions

The bus signals described in this chapter are the most important and essential signals to understand the application brief. The buffers needed per each signal are summarized in Table 2. The timing relations between the signals is drawn in Figure 1.

A0–A23

Address bits generated by the bus master to address memory and IO slaves attached to the bus. The address bits are unlatched and must be latched by the slaves using either the trailing edge of $\overline{ADL}$ or the leading edge of $\overline{CMD}$ signals.

D0–D15

Data bits, valid during the period $\overline{CMD}$ signal is low. The data is driven by bidirectional three-state drivers.

$\overline{ADL}$

Address Decode Latch, driven by the bus master. The signal is used by the slaves to latch valid address and status bits.

$\overline{CD}$_DS16

Card Data Size 16, driven by 16 bit slaves to provide an indication to the master about their data bus width. Eight-bit slaves do not drive this line.

$\overline{DS}$_16_RTN

Data Size 16 Return. A signal generated by the PS/2 system by AND-ing all the $\overline{CD}$_DS16 signals received from all the slave connected to the bus. The signal is provided by the PS/2 system to the bus masters.

M/$\overline{IO}$

Memory/$\overline{IO}$, driven by the bus master and indicates a memory or IO cycle. M/$\overline{IO}$ is latched by the slave at the leading edge of $\overline{CMD}$ signal.

$\overline{S0}$, $\overline{S1}$

Status bits, driven by the bus master and indicate the start of read or write cycle. The status bits are latched by the slaves using the leading edge of $\overline{CMD}$.

$\overline{CMD}$

Command signal is driven by the bus master and defines the period data is valid on the data bus. The leading edge of $\overline{CMD}$ is used to latch the unlatched signals: A0–23, M/$\overline{IO}$, $\overline{S0}$, and $\overline{S1}$. The trailing edge of $\overline{CMD}$ indicates the end of the bus cycle.

$\overline{CD}$_SFDBK

Card Select Feedback. When a bus master addresses a memory or an IO slave, the addressed slave drives $\overline{CD}$_SFDBK active as a positive acknowledgement of its presence at the specified address.

$\overline{CD}$_CHRDY

Channel Ready. This line is pulled inactive (not ready) by a slave to allow additional time to complete a bus cycle.

CHRDYRTN

Channel Ready Return. Generated on the PS/2 system board by AND-ing the $\overline{CD}$_CHRDY signals driven by all the slaves. The signal is provided by the system to the mastr driving the bus.

MCA Signal Descriptions (Cont.)

ARB0–ARB3
Arbitration Bus priority signals. These four signals represent the priority levels for masters seeking control on the bus. The four signals represent 16 priority levels, level 15 represents the lowest priority, level 0 represents the highest priority and belongs to the PS/2 system.

ARB/$\overline{GNT}$
Arbitrate/$\overline{Grant}$. When high, this signal indicates an arbitration cycle is in process. When low, indicates that a master has been granted. ARB/$\overline{GNT}$ is driven by the system.

$\overline{PREEMPT}$
Used by the arbitration bus masters to request the bus.

$\overline{BURST}$
Indicates that the master requests the bus for transferring a block of data.

$\overline{IRQ}$
Interrupt Request is used to signal the system that a device requires attention.

CHRESET
Channel Reset, active high reset signal generated by the system and sent to all the boards on the bus.

Figure 1. Micro Channel Basic Transfer Cycle

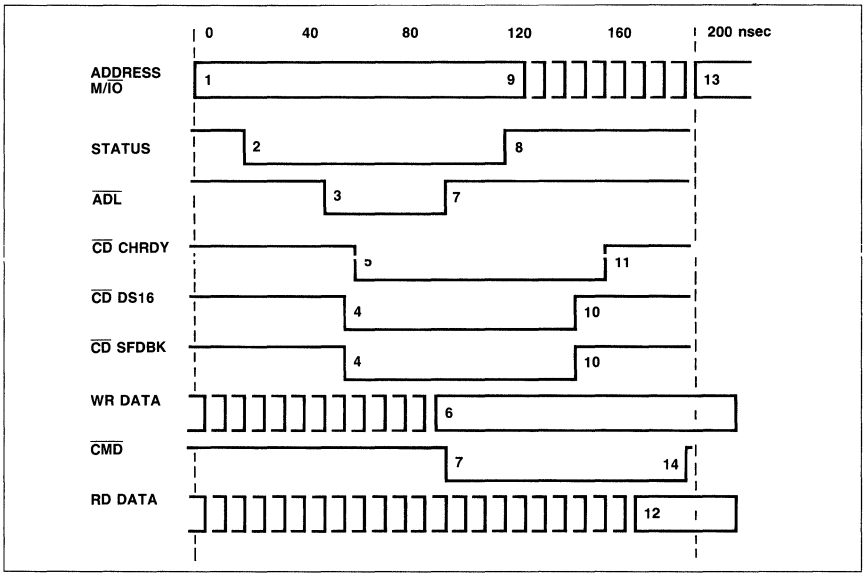


Table 1. The States Generated M/$\overline{IO}$, $\overline{S0}$ and $\overline{S1}$

M/$\overline{IO}$	$\overline{S0}$	$\overline{S1}$	
0	0	1	I/O write.
0	1	0	I/O read.
1	0	1	Memory write.
1	1	0	Memory read.

MCA Timing Parameters

The PAC1000 as a bus master transfers data on the MCA bus with a control sequence based on the following events:

- The addres bus and M/$\overline{IO}$ signal become valid.
- The status signals $\overline{S0}$ and $\overline{S1}$ are valid 10 nsec minimum after (1).
- $\overline{ADL}$ is valid 45 nsec minimum after (1).
- In response to the unlatched address decode, the selected slave responses with $\overline{CD}$_SFDBK (and $\overline{CD}$_DS16 if it is a 16 bit slave). The maximum allowable response time of the slave is 55 nsec maximum from (1).

MCA Timing Parameters (Cont.)

- In response to (1), the slave responds with $\overline{CD}$_CHRDY. The maximum allowable response time is 60 nsec maximum from (1).
- Write data appears on the bus for the write cycle. The data has to be valid before the leading edge of $\overline{CMD}$.
- $\overline{CMD}$ becomes active and $\overline{ADL}$ inactive typically 85 nsec minimum after (1). The unlatched signals on the bus are latched.
- The status signals become inactive after they were latched.
- The address bus becomes inactive after the address was latched.
- In response to the address change, the slave's unlatched responses ($\overline{CD}$_CFDBK AND $\overline{CD}$_DS16) are invalid.
- System stays in this state until $\overline{CD}$_CHRDY is ready.
- The slave places data on the bus in response to a read.
- The address and M/$\overline{IO}$ are valid for the next cycle.
- $\overline{CMD}$ goes inactive, ending the cycle.

Operation Modes

The PAC1000 working as a MCA controller can handle the following functions:

- Bus signal generator.
- Card setup.

The bus arbitration logic and signal decoding are pure asynchronous functions and implemented by two PALs.

Bus Slave Board

On a bus slave board the PAC1000 may be used to implement the POS registers.

The Programmable Option Select (POS) registers main objectives are:

- Eliminate switches from the board.
- Positively identify any card connected to the system.

The POS registers on a PS/2 board replace the switches by using software writeable registers. There are eight POS registers, each one is 8-bit wide. The POS registers are addressed by $\overline{CD}$_SETUP signal and by address bits A0–2. The POS registers are located at I/O addresses 100H to 107H. The eight POS registers are located in the PAC1000 and control the board's functions.

The POS registers' interface to the MCA is a decoder which decodes the sytem's access to the registers and generates the RD and WR signals to the PAC1000.

The address decoder and slave logic are most of the circuitry needed for the slave functions. The decoder has to decode the address on the bus and to respond with $\overline{CD}$_SFDBK, $\overline{CD}$_CHRDY and $\overline{CD}$_DS16 signals. The address decoder might be for memory, I/O or for both. The decoder's size depends on the number of address bits it is decoding. The decoder's $\overline{CS}$ outputs are latched by the leading edge of $\overline{CMD}$ and are stable until the end of the bus cycle. The decoder generates the feedbacks to the bus, $\overline{CD}$_SFDBK, $\overline{CD}$_DS16 and $\overline{CD}$_CHRDY. These signals are not latched and are very time critical. The decoder responds with these outputs at 55 nsec maximum after the address is stable.

Bus Master Board

A master board is a board with a CPU which requests the MCA bus. When granted by the PS/2 system, the master board is driving the bus signals.

On a master board the PAC1000 can handle the following functions:

- POS registers (similar to the bus slave board).
- Generation of the bus signals

The other functions of a bus controller are implemented by PALs because the functions are pure asynchronous.

The bus signals are generated by the PAC1000 after the CPU is granted to be a bus master. The process of getting the bus is done in the following sequence:

- The CPU is requesting the bus through one of the interface lines with the PAC1000.
- The PAC1000 is setting the bus request line which is buffered by drivers and sent to the MCA system.
- The system gets the request, and sets a bus arbitration cycle which is handled by the bus arbiter circuit (a PAL).

Operation Modes (Cont.)

- The bus arbiter sends the PAC1000 the signal MASTER which tells the board that the bus was granted and the board may drive the bus.
- The PAC1000 signals the CPU that it is the bus master.
- The PAC1000 is enabling the address and data drivers, and the CPU drives the address and data to the bus.
- The PAC1000 generates all the bus signals in the right sequence and the right timing requirements as defined by the MCA bus standard.
- After the CPU is done, it releases the bus request. The PAC1000 translates it to the right signal sequence on the MCA bus and releases the bus buffers.

On the bus master board the PAC1000 may implement a lot of control functions and save glue logic.

For example:
The PAC1000 can handle several DMA operations on the board, or be used as a high speed controller for various applications.

PAC1000 in a Micro Channel Slave Board

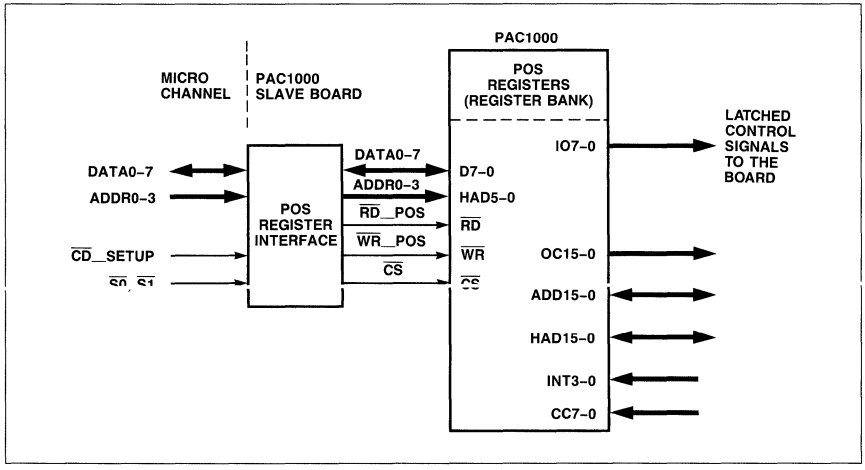
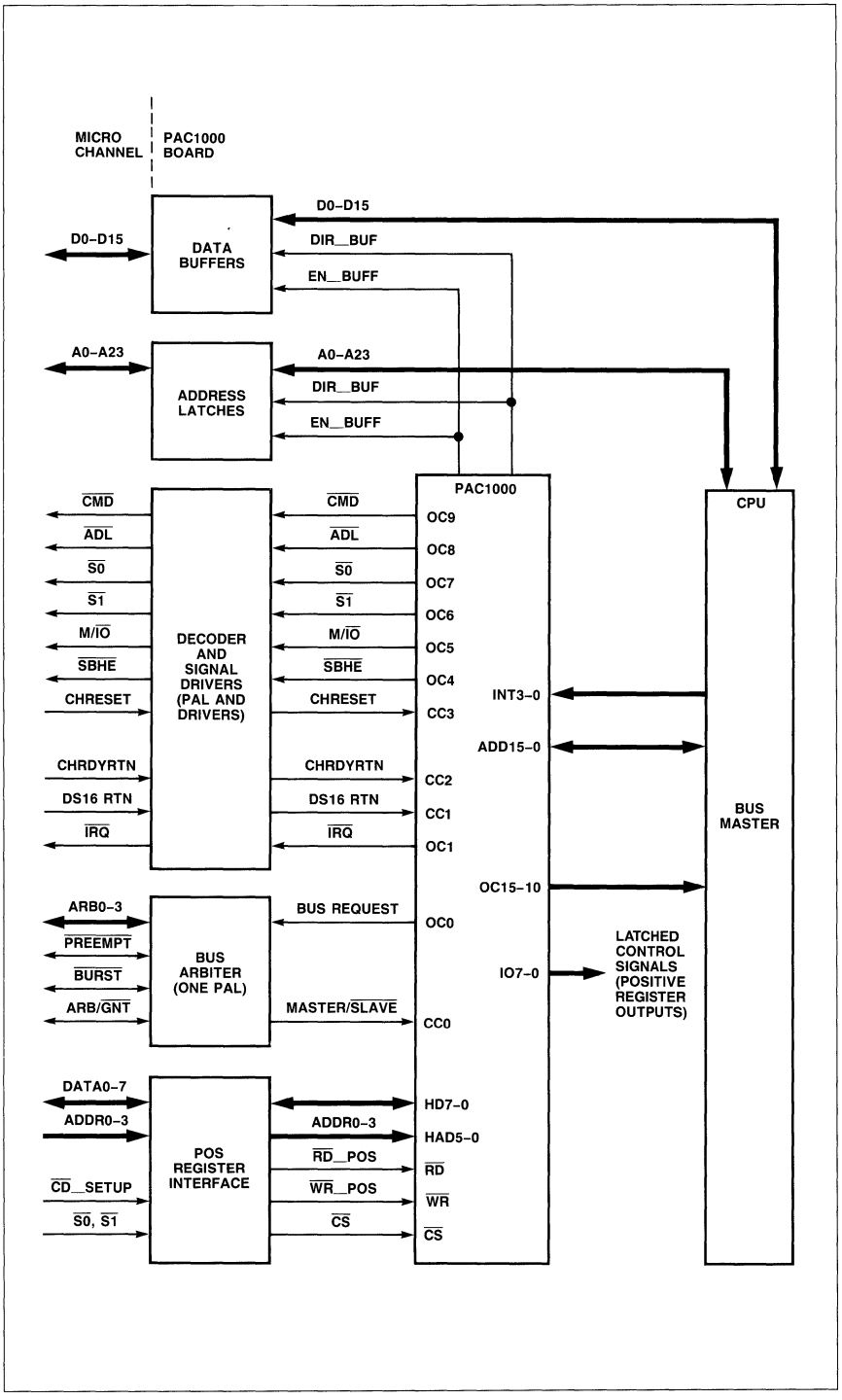


Table 2. Driver Requirement for PS/2 Signals

Signal Name	Driver Type
A(0–23)	TS 24 mA (TS = Three-State)
D(0–15)	TS 24 mA
$\overline{ADL}$	TS 24 mA
$\overline{CD}$_DS16	TP 6 mA (TP = Totem Pole)
$\overline{DS}$_16RTN	BD 24 mA (BD = Bus Driver)
M/$\overline{IO}$	TS 24 mA
$\overline{S0}$, $\overline{S1}$	TS 24 mA
$\overline{CMD}$	TS 24 mA
$\overline{CD}$_SFDBK	TP 6 mA
$\overline{CD}$_CHRDY	TP 6 mA
CHRDYRTN	BD 24 mA
ARB(0–3)	OC 24 mA (OC = Open Collector)
$\overline{PREEMPT}$	OC 24 mA
$\overline{BURST}$	OC 24 mA
ARB/$\overline{GNT}$	BD 24 mA

PAC1000 as a Micro Channel Master



Programmable Peripheral

Application Note 008

PAC1000 Programmable Peripheral Controller with a Built-In Self Test Capability

By David Fong

Abstract

The objective of this Application Note is to demonstrate the Built-In Self Test (BIST) capability of the PAC1000 High-Performance Programmable Peripheral Controller. This article describes the basic instructions needed to implement BIST.

Introduction

With increasing device densities on one chip, more devices are needed for BIST to check the functionality of the internal logic. Current serial scan techniques for board level verification would take too much time and resources. The current PAC1000 will only test the ALU and its status flags, the address and block counter, and the sequencer. Future versions in the WS-PAC Family will have even larger sizes of EPROM and may test the control EPROM.

Usage and Limitations

The program is accessible by calling the BIST program. The program occupies forty-five lines of EPROM code. The program can be reduced in size by specifying extra CPU registers to hold the constants h'FFFF', h'0000', h'AAAA', h'5555' and h'FFF4'.

Certain conditions must be met prior to programming the code to ensure that this program will work correctly. The stack should be empty because the program exercises the stack. In addition, location h'3FF' must be reserved because the BIST uses this location to verify the contents of the stack as a '1.' The outputs should be placed in a mode where the existing system is not affected. The 'MAINT' instruction will ensure that the OC is the same throughout the program. However, this example was not implemented in that manner. Instead, it uses set values to assist in debugging the program. Users can do a global substitution of "OUT h'xxxx' " with "MAINT" in their word processor to fully implement this BIST program.

This BIST is not a panacea for system designers. A 'PASS' condition is indicated by a return to the main calling program. The output control will be h'0000'. A 'FAIL' condition will result in some endless loop or jump to some portion of the program. In the event that it does fail after about 170 clock cycles, the system must disable the PAC1000 from the rest of the system in some manner. Future versions of the PAC1000 may include a watchdog timer to interrupt and timeout the BIST.

The variables that can be altered by the user are listed at the beginning of the BIST.mal file. The current values used will only exercise the counters in a simple manner. The user can modify these variables to increase the confidence level of the program at the expense of a longer test cycle.

Usage and Limitations (Cont.)

A summary of the instructions used and the functional blocks follow below:

```
/*********************************************************************/
/* registers destroyed : R0,R1,R2,R3 and R4                          */
/*                       AOR,ACH,ACL,BC,LC and stack                 */
/*                                                                   */
/* stack should be empty before calling this program                 */
/*                                                                   */
/* the block counter, address counter, ALU with register file and    */
/* flags,and the sequencer with stack and counter are tested         */
/*                                                                   */
/* flags checked: BCZ,ACO,CY,Z,O,S,and STKF                          */
/* ALU instructions used: ADC,AND,ADD,MOV,NOP,SHRR,SHRL,SUB          */
/* CONTROL instructions used: ACSIZE,CONT,JMPNC,JMPC,LDLCD,          */
/*              LOOPNZ,PLDLC,POP,RET,RNC,RSTCON and SETCON           */
/*                                                                   */
/* DATA from EPROM used: 0000, FFFF, FFF4, AAAA, 5555 ,0008 , 0010,  */
/*              03FF, 0019                                           */
/*********************************************************************/
```

Confidence Level

The program executes some of the possible internal critical paths of the PAC1000. From tester and simulation measurements, the test of condition codes and branching were consistently the longest. Similarly for the ALU, flag generation such as adding with a carryout is considered a critical path. The counters have a critical path in propagating the carry. Overall, the confidence level of this test is considered to be high.

Analysis of the Program

The currently executing program calls the BIST program by using the 'CALL' instruction. The instruction following 'CALL' which is the return address is pushed to the stack and is not destroyed by the BIST program. See Figure 1 for the BIST flowchart. The BIST tests the PAC1000 functional blocks in the following order:

1. Block Counter and flag BCZ.
2. Address Counter and flag ACO.
3. ALU with shifter and flags CY, Z, O and S.
4. Sequencer with stack and loop counter, and flag STKF.

Some subtleties of programming the PAC1000 are presented. In the ALU section, certain flags must be forced to zero before being tested upon, unlike the normal microprocessors where the individual flags are set and reset by instructions. The ALU result of each cycle updates each flag on the next rising edge of the clock. For example, to check the zero flag (Z), some ALU instruction forces the Z flag to zero. See the instructions below:

```
        MOV R2 R2 , OUT h'0138' ;
        /* force zero flag Z=0 */
zero:   JMPNC Z zero, AND AOR R1 ,
        OUT h'0139' ;
```

Next, loading the loop counter from the ALU needs special treatment. The data must be present at the ALU output before the instruction to load the loop counter executes. See the instructions below:

```
        MOV R4 short, OUT h'014B' ;
            /* force ALU output to the
            value of short = h'0010' */
        LDLCD , MOV R4 R4, OUT h'014C' ;
            /* load 0010 to LC */
```

Analysis of the Simulation Output

Looking at the block counter outputs BC(15:0) from cycle 7 through 18, the counter counts continuously until disabled. The block counter contents wraps around from h'0000' to h'ffff' and down. Note that the BCZ flag remains latched until new data is loaded to the block counter. Because of the latched flag BCZ, there is a minimum of two cycles before the next instruction is executed after the loop. Figure 2 shows the loop with the minimum number of latency cycles before executing the next line of program code.

Figure 1.
Built-In-Self-Test
Flowchart

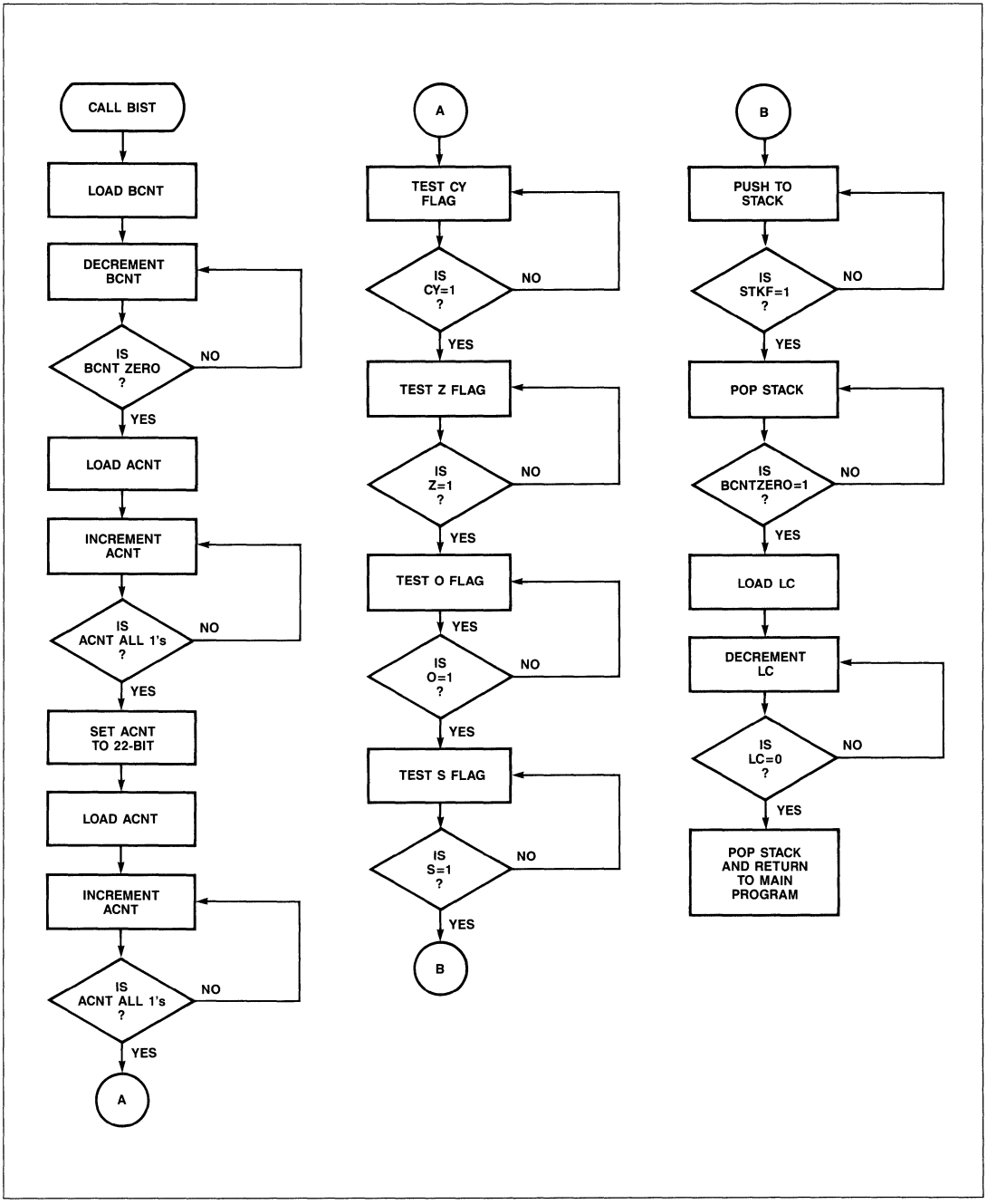
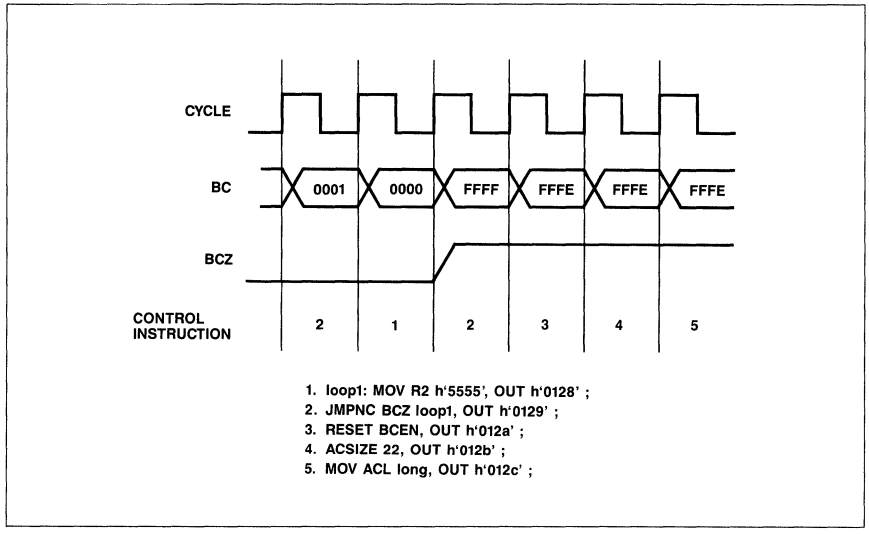


Figure 2. BCZ Flag: Example Cycle-by-Cycle Simulation




```
/*********************************************/
/* Main calling program          02/03/89    */
/* David Fong                    Rev. 1.0    */
/* main.mal                                  */
/*********************************************/

segment main ;

external bist ;

main1:

/* initialize */
/* not needed */

/* call bist program */

          CALL bist , OUT h'0123' ;     /* call the BIST program */
          /* return to main program */
FORE:     JMP FORE , OUT h'0000' ; /* loop forever */

end ;
```

```
/*********************************************/
/* Program to jump back to main bist program */
/* David A. Fong 02/03/89        Rev. 1.0    */
/* jmpf.mal                                  */
/*********************************************/

segment jmp ;
external jmpf ;

JMP jmpf , OUT h'FFFF' ; /* jmpf is an external address */
/* this tests branching with all 1's */
end ;
```

```
/*****************************************************************/
/* Built-In-Self-Test Program  02/03/89                          */
/* David A. Fong               Rev. 1.0                          */
/* bist.mal                                                      */
/*****************************************************************/
/* registers destroyed : R0,R1,R2,R3 and R4                      */
/*                       AOR,ACH,ACL,BC,LC and stack             */
/*                                                               */
/* stack should be empty before calling this program             */
/*                                                               */
/* the block counter, address counter, ALU with register file and*/
/* flags,and the sequencer with stack and counter are tested     */
/*                                                               */
/* flags checked: BCZ,ACO,CY,Z,O,S,and STKF                      */
/* ALU instructions used: ADC,AND,ADD,MOV,NOP,SHRR,SHRL,SUB      */
/* CONTROL instructions used: ACSIZE,CONT,JMPNC,JMPC,LDLCD,      */
/*             LOOPNZ,PLDLC,POP,RET,RNC,RSTCON and SETCON        */
/*                                                               */
/*DATA from EPROM used: 0000, FFFF, FFF4, AAAA, 5555 ,0008 , 0010*/
/*             03FF, 0019                                        */
/*****************************************************************/


segment c_bist ;
entry bist,jmpf ;   /* entry points into this program */

/* define equates for user to substitute */
shorter equ h'0008' ;
short     equ h'0010' ;
medium    equ h'03ff' ;
long      equ h'fff4' ;
popper    equ h'0019' ;

/*****************************/
/* test the counters and     */
/* initialize the registers */
/*****************************/

bist:     MOV R1 h'0000', OUT h'0124'; /*the outputs should be placed*/
                                       /* in a non-functional mode */
          MOV R0 h'FFFF' , OUT h'0125' ; /* in this program it is not*/
          MOV BC shorter , OUT h'0126' ;/*because it was needed to*/
          SETCON h'002' , OUT h'0127' ; /*debug enable block counter */
loop1:    MOV R2 h'5555' , OUT h'0128' ;
          JMPNC BCZ loop1 , OUT h'0129' ;
RSTCON h'002' , OUT h'012A' ; /* disable block counter */

/* R0 = FFFF ; R1 = 0000 ; R2 = 5555 */

/* test the 22-bit address counter */

          ACSIZE 22 , MOV ACH R0 , OUT h'012B' ;
          MOV ACL long , OUT h'012C' ;
          SETCON h'001' , OUT h'012D' ; /* enable address counter */
```

```
loop2:    MOV R3 h'AAAA' , OUT h'012E' ;
          JMPNC ACO loop2 , OUT h'012F' ;
          RSTCON h'001' , OUT h'0130' ; /* disable address counter */

/* R0 = FFFF ; R1 = 0000 ; R2 = 5555 ; R3 = AAAA */

/* test the 16-bit address counter */

          ACSIZE 16 , OUT h'0131' ;
          MOV ACH long , OUT h'0132' ;
          SETCON h'001' , OUT h'0133' ; /* enable address counter */
loop3:    MOV R4 h'0000' , OUT h'0134' ;
          JMPNC ACO loop3 , OUT h'0135' ;
          RSTCON h'001' , MOV R3 R3 , OUT h'0136' ;
          /* disable address counter */
/* and do a dummy ALU instruction so that Z=0 and CY=0 */
/* note: a NOP instruction will force Z=1 and CY=1 on the */
/* following cycle*/

/* R0 = FFFF ; R1 = 0000 ; R2 = 5555 ; R3 = AAAA ; R4 = 0000 */
/* R4 is the working register */

/****************/
/* test the ALU */
/****************/

carry:    JMPNC CY carry , ADC AOR R0 , OUT h'0137' ;/*test carryout */

          MOV R2 R2 , OUT h'0138' ;     /* force zero flag = 0 */
zero:     JMPNC Z zero , AND AOR R1 , OUT h'0139' ;/*test all the alu*/
                         /* outputs are zero */

over:     SUB AOR R3 R2 , OUT h'013A' ; /* test for overflow */
          JMPNC O over , OUT h'013B' ;  /* test for overflow */

f15:      JMPNC S f15 , ADD AOR R1 R0 , OUT h'013C' ;/*test sign bit*/

/* test the alu shifting */

shftl:    SHLR R2 Z , OUT h'013D' ;
          AND AOR R3 R2 , OUT h'013E' ; /*should not loop*/
                                        /*but fall-thru */
          JMPC Z shftl , OUT h'013F' ;

shftr:    SHRR R2 Z , OUT h'0140' ;
          AND AOR R3 R2 , OUT h'0141' ;
               /* should not loop,but fall-thru */
          JMPNC Z shftr , OUT h'0142' ;

/**********************/
/* test the sequencer */
/**********************/

          MOV BC short , OUT h'0143' ;
```

```
          SETCON h'002' , OUT h'0144' ; /* enable block counter */
stack:    PLDLC medium , OUT h'0145' ;
          JMPNC STKF stack , OUT h'0146';
          /*exit loop when stack is full */
                    /* the return address will not be */
                    /* overwritten , only the top of stack*/
          MOV BC popper , OUT h'0147' ;
jmpf:     RNC BCZ , OUT h'0148' ;
                    /*should come out of loop when empty+1*/
                    /* which is the return address */
          POP , NOP , OUT h'0149' ;
          /* pop one more time but don't pop */
          /* the last return address */
          RSTCON h'002' , OUT h'014A' ; /* disable block counter */

     /* test the loop counter */
          MOV R4 short , OUT h'014B' ;
          LDLCD , MOV R4 R4 , OUT h'014C' ;/* load 16 into the LC*/
lp:       ADC AOR R4 , OUT h'014D' ;    /* aor = aor + r4 */
          LOOPNZ lp , OUT h'014E';/*check that loop count is not zero*/

          RET , OUT h'014F' ;      /* return to calling program */

end ;


/*********************************/
/* bist linker file   02/03/89    */
/* David Fong         Rev. 1.0    */
/* exbist.ml                      */
/*********************************/

place main , c_bist , jmp ;   /* place the segments  */
load main , bist , jmpf ;     /* load the .mal files */

locate main , h'000' ;   /* locate main and init file */
locate c_bist , h'011' ; /* locate bist file after interrupt */
locate jmp , h'3ff' ;    /* locate jmp at 3ff to test '1' from stack */

end ;


.T
         RCCCCCCCCCIIIIIIIIIIIICWRHHHHHHHHHHHHHHHHHHHHHHHHHAAAAAAAAAAAAAAAA
         ECCCCCCCCCOOOOOOOOONNNNSRDDDDDDDDDDDDDDDDDAAAAAADDDDDDDDDDDDDDDD
         S7654321076543210TTTTBBB1111119876543210DDDDDDDDDDDDDDDDDDDDDDD
         E                3210   543210          543210111119876543210
         T                                           543210
         B
TIME
1        00000000000000000000011100000000000000000000000000000000000000
2        10000000000000000000011100000000000000000000000000000000000000
```

```
**********************************************************************

                          O U T P U T   T A B L E

P A C S I M     Ver. 1.09                    Mon Feb 13 15:12:09 198
**********************************************************************

        PPP OOOO LLL AAAA BBBB AAAA AA BASCOSZ RRRR RRRR RRRR RRRR
        CCC CCCC CCC OOOO CCCC CCCC CC CCTY    3333 2222 1111 0000
        173 1173 173 RRRR 1173 HHHH LL ZOK
        1:: 51:: 1:: 1173 51:: 1173 53   F     1173 1173 1173 1173
        :40 ::40 :40 51:: ::40 51:: ::         51:: 51:: 51:: 51::
        8   18   8   ::40 18   ::40 40         ::40 ::40 ::40 ::40
            2        18   2    18              18   18   18   18
                     2         2               2    2    2    2
  TIME
     1  000 0000 000 0000 0000 0000 00 0010000 0000 0000 0000 0000
     2  000 0000 000 0000 0000 0000 00 1010000 0000 0000 0000 0000
     3  011 0123 000 0000 0000 0000 00 1000001 0000 0000 0000 0000
     4  012 0124 000 0000 0000 0000 00 1001001 0000 0000 0000 0000
     5  013 0125 000 0000 0000 0000 00 1000001 0000 0000 0000 0000
     6  014 0126 000 0000 0000 0000 00 1000010 0000 0000 0000 ffff
     7  015 0127 000 0000 0008 0000 00 0000000 0000 0000 0000 ffff
     8  016 0128 000 0000 0007 0000 00 0001001 0000 0000 0000 ffff
     9  015 0129 000 0000 0006 0000 00 0000000 0000 5555 0000 ffff
    10  016 0128 000 0000 0005 0000 00 0001001 0000 5555 0000 ffff
    11  015 0129 000 0000 0004 0000 00 0000000 0000 5555 0000 ffff
    12  016 0128 000 0000 0003 0000 00 0001001 0000 5555 0000 ffff
    13  015 0129 000 0000 0002 0000 00 0000000 0000 5555 0000 ffff
    14  016 0128 000 0000 0001 0000 00 0001001 0000 5555 0000 ffff
    15  015 0129 000 0000 0000 0000 00 0000000 0000 5555 0000 ffff
    16  016 0128 000 0000 ffff 0000 00 1001001 0000 5555 0000 ffff
    17  017 0129 000 0000 fffe 0000 00 1000000 0000 5555 0000 ffff
    18  018 012a 000 0000 fffd 0000 00 1001001 0000 5555 0000 ffff
    19  019 012b 000 0000 fffd 0000 00 1001001 0000 5555 0000 ffff
    20  01a 012c 000 0000 fffd ffff 00 1000010 0000 5555 0000 ffff
    21  01b 012d 000 0000 fffd ffff 34 1000010 0000 5555 0000 ffff
    22  01c 012e 000 0000 fffd ffff 35 1001001 0000 5555 0000 ffff
    23  01b 012f 000 0000 fffd ffff 36 1000010 aaaa 5555 0000 ffff
    24  01c 012e 000 0000 fffd ffff 37 1001001 aaaa 5555 0000 ffff
    25  01b 012f 000 0000 fffd ffff 38 1000010 aaaa 5555 0000 ffff
    26  01c 012e 000 0000 fffd ffff 39 1001001 aaaa 5555 0000 ffff
    27  01b 012f 000 0000 fffd ffff 3a 1000010 aaaa 5555 0000 ffff
    28  01c 012e 000 0000 fffd ffff 3b 1001001 aaaa 5555 0000 ffff
    29  01b 012f 000 0000 fffd ffff 3c 1000010 aaaa 5555 0000 ffff
    30  01c 012e 000 0000 fffd ffff 3d 1001001 aaaa 5555 0000 ffff
    31  01b 012f 000 0000 fffd ffff 3e 1000010 aaaa 5555 0000 ffff
    32  01c 012e 000 0000 fffd ffff 3f 1001001 aaaa 5555 0000 ffff
    33  01d 012f 000 0000 fffd 0000 00 1100010 aaaa 5555 0000 ffff
    34  01e 0130 000 0000 fffd 0000 01 1101001 aaaa 5555 0000 ffff
    35  01f 0131 000 0000 fffd 0000 01 1101001 aaaa 5555 0000 ffff
    36  020 0132 000 0000 fffd 0000 01 1101001 aaaa 5555 0000 ffff
    37  021 0133 000 0000 fffd fff4 01 1000010 aaaa 5555 0000 ffff
```

Programmable Peripheral
Application Note 009
In-Circuit Debugging for the PAC1000 Programmable Peripheral Controller

By David Fong

Abstract

This Application Note is used to illustrate the in-circuit debugging capabilities of the PAC1000 programmable peripheral controller.

Introduction

With the increasing densities and complexities of integrated circuits, the usage of tools such as in-circuit debuggers and emulators is greatly desired by the heroic hardware designer. The PAC1000 supports the usage of in-circuit debuggers.

A review of BP (breakpoint) and SS (single step) is discussed. SS is the method of stepping through the program code one instruction at a time through manual means. In the case of the PAC1000, there is no manual means with a single-step switch. Instead, an interrupt which is set internally through the program is set. This interrupt can then call upon an ISR (interrupt service routine). This subroutine then dumps out the contents of all the possible registers that can be read out. These registers must then be written into the system memory by the user to use in his monitor program. SS is useful for checking that every cycle is executing correctly.

On the other hand, BP is the method of interrupting the program at a specific program location. This allows the program in the PAC1000 to run in real-time system conditions. This breakpoint is passed to the PAC1000 through the FIFO instead of having a fixed address through the program. BP is useful for intermittently checking the execution of the program.

There is no preference on which method is the best. Generally, it is determined by the situation. If the system designer doesn't trust their own system in the beginning of debug, then they will use SS. After the system becomes more debugged, breakpoint is needed occasionally.

Usage and Limitations

Either SS or BP interrupts can occur. Because both use the same initial ISR, the ISR will differentiate between the two by testing for a specific data pattern that accompanies the breakpoint/single-step data through the FIFO. One way was to test for a specific external condition code but that was determined to be inflexible since a specific condition code needed to be dedicated for this task. Instead, two words are written into the CPU registers. These two registers must be reserved for breakpoint/single-step operation. In this example, R0 and R1 are reserved. Register R1 is the mask that is 'AND'd with R0 which is written from the FIFO to produce the Z (zero) flag that is tested. See Figure 1 for the data format that is written into the FIFO and CPU register R0.

The BP state continues with its program by reading out the contents of some registers to the host interface bus. Note the usage of the FIFO to read out the contents of the register to the ADD bus. BP reads out only the input and output registers that can be read as source to the HD bus. Whereas, SS reads out the CPU registers as well as the input and output registers to ADD.

Not all the registers can be read out or if at all with difficulty. CPU registers as was illustrated by this program was read out using the FIFO. However, the user could have individually read out each register. Unfortunately, there would have been a lot of overhead program space taken. The stack cannot be read out because the contents of the stack would affect the program flow. The interrupt mask register and interrupt pending register cannot be read out or to the CPU. Future PAC1000 versions may support extra functions to allow the user to more easily access the internal registers.

In summary, the single-step program dumps out the following registers to the ADD bus: CPU registers R31–R0, DIR, AIR, ACH, ACL, IIR, and BC. Whereas, the breakpoint program dumps out the following registers to the HD bus: DIR, AIR, AOR, ACH, ACL, IIR, and BC.

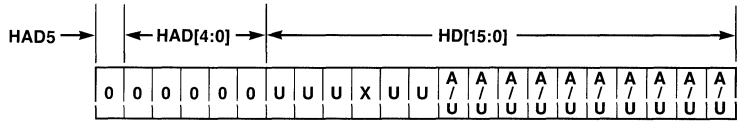
Analysis of Program

This single program incorporates essentially two programs. One for breakpoint and one for single-step. To differentiate between the two programs since they use the same interrupt INT6, the data in register R0 is tested upon and the corresponding action is taken. If Z is true, then breakpoint will occur, else single-step will occur. See BREAKPOINT/SINGLE-STEP algorithm Figure 2.

Note that the Interrupt Jump Table is located at h'008' through h'00f'. The PAC1000 interrupt vector from the internal interrupt jumps to these individual locations. In addition, note that neither conditional nor unconditional jumps were allowed to be executed when either the breakpoint or the single-step interrupts occurred. This also applies to other interrupts. The delay interval from the time of interrupt to executing the interrupt is two cycles. See Figure 3 for the timing relationship of interrupt to the beginning of execution of the interrupt service routine (ISR).

The single-step subroutine utilizes the FIFO to externally address the CPU dual-port registers. The usage of the FIFO in conjunction with loops reduces the size of the control store. However, the contents of the FIFO must be empty before using it.

Figure 1. Host to PAC1000 Commands (Via the FIFO)



LEGEND:

U: User-Defined
X: Test Bit
A/U: Breakpoint Address or User-Defined

HAD5 = FICD: The flag to indicate that the contents are data FICD=0 or a command FICD=1.

HAD[4:0] = The B address to the CPU register file which in this case is register R0.

HD[15:13] = User-defined.

HD[12] = Test bit to differentiate between breakpoint and single-step. HD12=0 for breakpoint and HD12=1 for single-step.

HD[11:10] = User-defined.

HD[9:0] = Breakpoint Address or for single-step user-defined.

Figure 2. Breakpoint/ Single-Step Flowchart

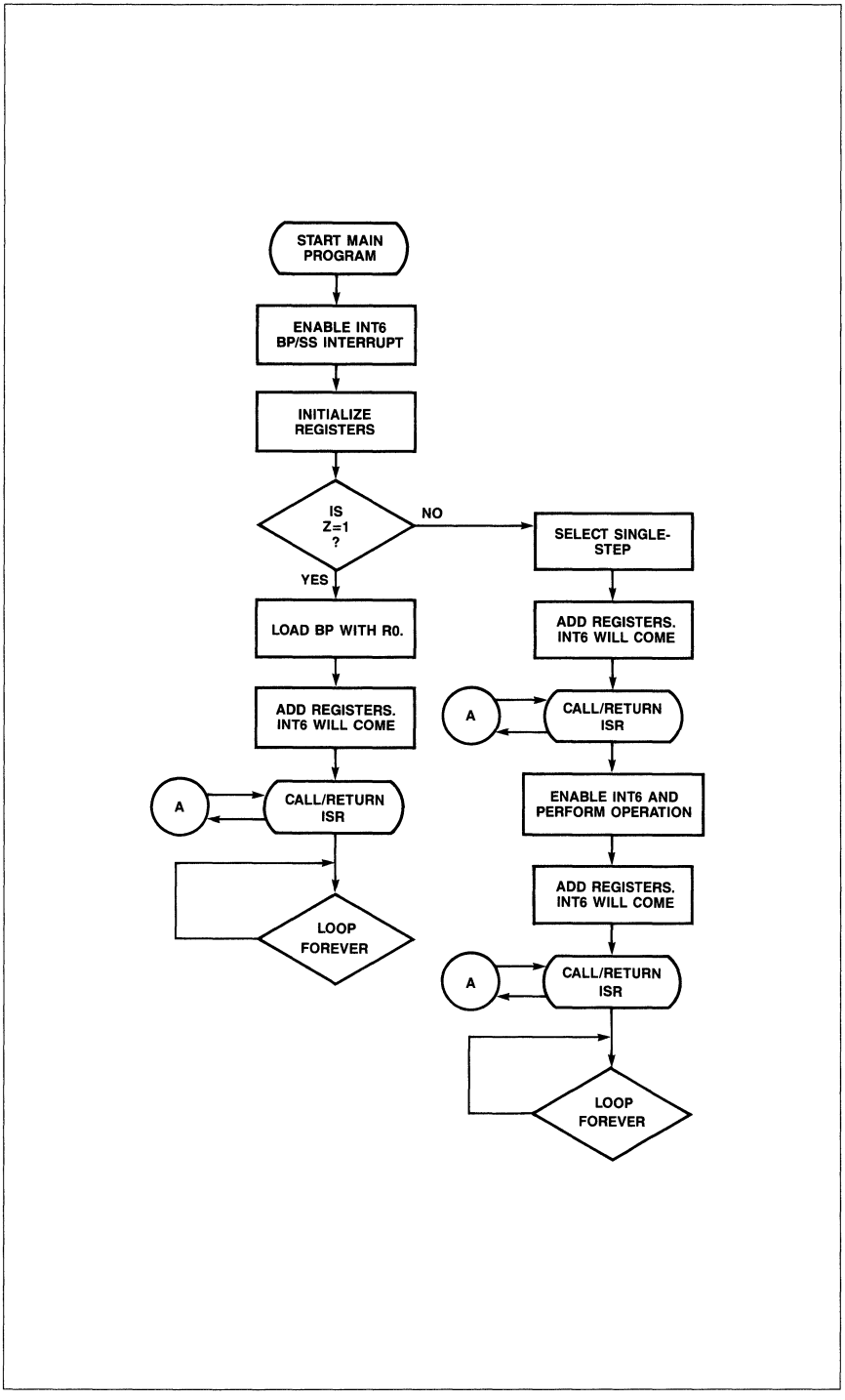


Figure 2. Breakpoint/ Single-Step Flowchart (Cont.)

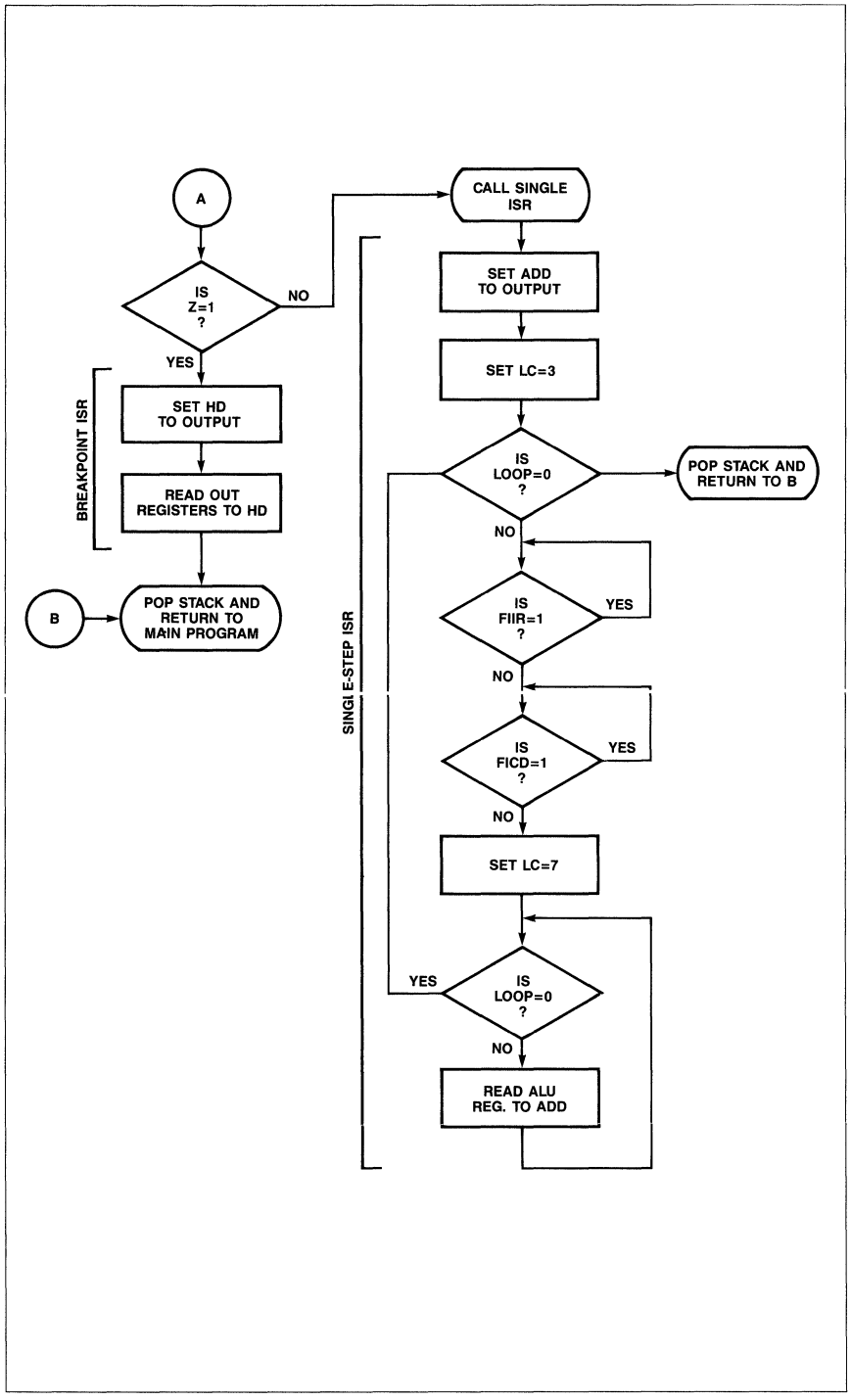
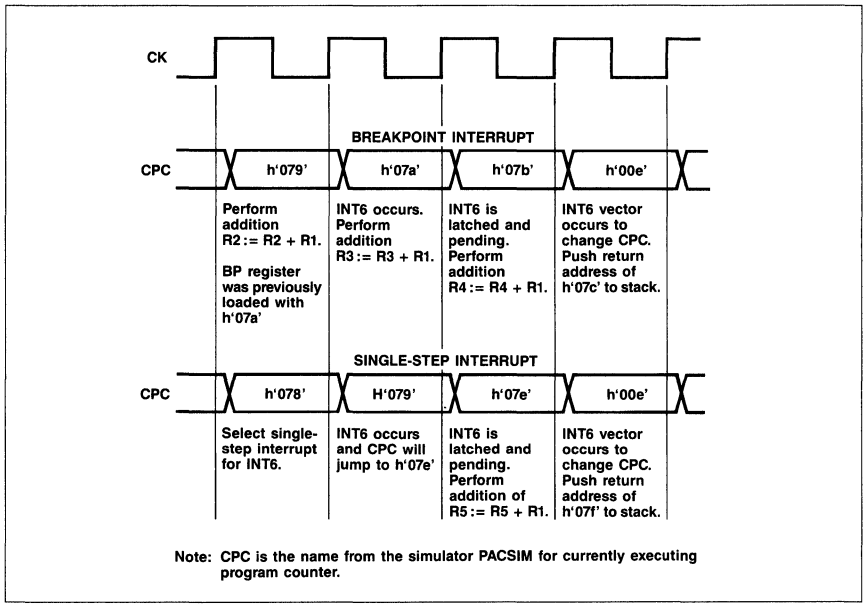


Figure 3. Sequence of Events for Interrupt Timing



Note: CPC is the name from the simulator PACSIM for currently executing program counter.

```
/*****************************************/
/* BP and SS linker file       04/03/89    */
/* David Fong              Rev. 1.0        */
/* bpss.ml                                 */
/*****************************************/

place main, int, intserv, init, single ;     /* place the segments */
load main, int, intserv, init, single ; /* load the .mal files */

locate init , h'000' ;          /* locate the init file */
locate intserv , h'008' ;       /* locate the interrupt vectors */
locate main , h'050' ;          /* locate the main file */
locate int , h'100' ;           /* locate the ISR */
locate single , h'200' ; /* locate the single files */

end ;

/*********************************/
/* INITIALIZATION   04/03/89     */
/* David Fong       Rev. 1.0     */
/* init.mal                      */
/*********************************/

segment init ;

external main1 ;

SETMODE h'001' , OUT h'0002' ;     /* switch to interrupt mode */
ENABLE INT6 , OUT h'0001' ;
JMP main1 , OUT h'0000' ; /* jump to main program */

end ;
```

```
/*********************************/
/* Main program          04/03/89 */
/* David Fong       Rev. 1.0      */
/* main.mal                       */
/*********************************/

segment main ;

entry main1 ;

main1 :

/***********************************************/
/* BEGIN MAIN PROGRAM                          */
/***********************************************/

/* initialize registers */

R1 := h'1000' , OUT h'0050' ;/* the twelveth bit R1.12 tests for BP/SS*/
/* IF Z=1 (which means R1.12 = 0 ) THEN run breakpoint program */
/* ELSE run single-step program */
R2 := h'0002' , OUT h'0051' ;
R3 := h'0003' , OUT h'0052' ;
R4 := h'0004' , OUT h'0053' ;
R5 := h'0005' , OUT h'0054' ;
R6 := h'0006' , OUT h'0055' ;
R7 := h'0007' , OUT h'0056' ;
R8 := h'0008' , OUT h'0057' ;
R9 := h'0009' , OUT h'0058' ;
R10 := h'000a' , OUT h'0059' ;
R11 := h'000b' , OUT h'005a' ;
R12 := h'000c' , OUT h'005b' ;
R13 := h'000d' , OUT h'005c' ;
R14 := h'000e' , OUT h'005d' ;
R15 := h'000f' , OUT h'005e' ;
R16 := h'0010' , OUT h'005f' ;
R17 := h'0011' , OUT h'0060' ;
R18 := h'0012' , OUT h'0061' ;
R19 := h'0013' , OUT h'0062' ;
R20 := h'0014' , OUT h'0063' ;
R21 := h'0015' , OUT h'0064' ;
R22 := h'0016' , OUT h'0065' ;
R23 := h'0017' , OUT h'0066' ;
R24 := h'0018' , OUT h'0067' ;
R25 := h'0019' , OUT h'0068' ;
R26 := h'001a' , OUT h'0069' ;
R27 := h'001b' , OUT h'006a' ;
R28 := h'001c' , OUT h'006b' ;
R29 := h'001d' , OUT h'006c' ;
R30 := h'001e' , OUT h'006d' ;
R31 := h'001f' , OUT h'006e' ;

ACH := R31 , OUT h'006f' ;
ACL := R0 , OUT h'0070' ;
AOR := R1 , OUT h'0071' ;
DOR := R15 , OUT h'0072' ;
BC  := R7 , OUT h'0073' ;
```

```
/* all input registers are initialized to zero from RESET */

/* to integrate two different programs 1. BREAKPOINT 2. SINGLE-STEP*/
/* The result of masking R0 with R1 is used to differentiate */
/* between BP and SS. */
/* IF Z = 1 Breakpoint; ELSE Z = 0 Single-Step */

/***************** READ IN FIFO AND TEST FOR BP/SS ********************/
g0:  JMPC FICD g0 , OUT h'0074' ; /*check that the fifo contents is data
LDBPD , RDFIFO , OUT h'0075'; /* FIFO was loaded with h'0 00 007a' */
/* first 0 is FICD ; 00 is B address ; 0 is the test bit ; */
/* 07a is the EPROM breakpoint address. */
/* Load loop counter with same data read from FIFO : LDLCD; */
/* the data written into the CPU is the same as the CPU output bus */

AND R1 R0 , OUT h'0076' ; /* the Z flag is tested in the next cycle */
JMPC Z b0 , OUT h'0077' ;
/* select single-step interrupt */
ESS , OUT h'0078' ;
JMP c0 , OUT h'0079' ; /* skip breakpoint routine */

/***************** BREAKPOINT ************************************/
/* perform alu operations till interrupt comes */
b0:  R2 := R2 + R1 , OUT h'007a' ; /* breakpoint on this address h'07a'
R3 := R3 + R1 , OUT h'007b' ;
R4 := R4 + R1 , OUT h'007c' ; /* breakpoint interrupt comes here */
/* return from ISR to here */
e0 : JMP e0 , OUT h'007d' ;   /* loop forever ; end of breakpoint */


/***************** SINGLE-STEP ************************************/
c0:  R5 := R5 + R1 , OUT h'007e' ; /* execute till interrupt comes */
R6 := R6 + R1 , OUT h'007f' ; /* interrupt should after here */

/* return from single-step ISR to here */
/* enable single-step interrupt and perform an operation */
ENABLE INT6 , R7 := R7 + R1 , OUT h'0080' ; /* the output for R2 */
                              /* should be h'1002' */
R8 := R8 + R1 , OUT h'0081' ; /* interrupt should come here */
/* return from single-step ISR to here */

f0 : JMP f0 , OUT h'0082' ;   /* loop forever */
end ;


/*********************************/
/*SINGLE-STEP SUBROUTINE 04/03/89*/
/* David Fong       Rev. 1.0     */
/* single.mal                    */
/*********************************/

segment single ;
entry single1 ;

single1 :
     /* read out the registers from the ALU */
     /* use the addressing scheme from the FIFO */
```

```
     SETCON h'010' , OUT h'2000' ; /* set ADD bus to output */
     /* to read out AOR to ADD */

     /* loop four times to address the 32 registers */

     FOR 3 , OUT h'2001' ;

     /* FIFO should already be full */

f0 : JMPC FIIR f0 , OUT h'2002' ;  /* loop till FIFO is full*/

     /* check that the first value in the FIFO is a data */
f1 : JMPC FICD f1 , OUT h'2003' ;

     /* loop eight times to empty the FIFO */
          FOR 7 , OUT h'2004' ;

     /* use the FIFO as an address pointer */
     /* the data is not needed; write the data back to CPU */
     /* and output the CPU output to AOR */
     /* the default CPU instruction is add which adds zero and */
     /* the address pointed by the FIFO which is the B address */

          RDFIFO , alu_src = zb , ybus_sel = y_aoreg ,
          OUT h'2005' ;
          ENDFOR , OUT h'2006' ;

     ENDFOR , OUT h'2007' ;

     /* read out the source registers to ADD */
     MOV AOR DIR , OUT h'2008' ; /* 0000 should come out next cyle */
     MOV AOR AIR , OUT h'2009' ; /* 0000 */
     MOV AOR ACH , OUT h'200a' ; /* 001f */
     MOV AOR ACL , OUT h'200b' ; /* 0000 */
     MOV AOR IIR , OUT h'200c' ; /* 0000 */
     MOV AOR BC  , OUT h'200d' ; /* 0007 */

     RET , OUT h'200e' ; /* return to ISR 6 */

end ;


/*********************************/
/* INTERRUPT JUMP TABLE  04/03/89*/
/* David Fong       Rev. 1.0     */
/* intserv.mal                   */
/*********************************/

segment intserv ;
entry int_serv ;
external int0,int1,int2,int3,int4,int5,int6,int7 ;

int_serv :
     JMP int0 , OUT h'0008' ;
     JMP int1 , OUT h'0009' ;
     JMP int2 , OUT h'000a' ;
     JMP int3 , OUT h'000b' ;
```

```
     JMP int4 , OUT h'000c' ;
     JMP int5 , OUT h'000d' ;
     JMP int6 , OUT h'000e' ;
     JMP int7 , OUT h'000f' ;

end ;


/*********************************************************/
/* Interrupt Service Routines      04/03/89              */
/* David Fong                  Rev. 1.0                  */
/* int.mal                                               */
/*********************************************************/


segment int ;
entry int0 , int1 , int2 , int3 , int4 , int5 , int6 , int7 ;
external single1 ;

int0 :
     /* clear all the external interrupts */
     CLI h'00f' , OUT h'0100' ;
     RET , OUT h'0101' ;

int1 :
     /* clear all the external interrupts */
     CLI h'00f' , OUT h'0102' ;
     RET , OUT h'0103' ;

int2 :
     /* clear all the external interrupts */
     CLI h'00f' , OUT h'0104' ;
     RET , OUT h'0105' ;

int3 :
     /* clear all the external interrupts */
     CLI h'00f' , OUT h'0106' ;
     RET , OUT h'0107' ;

int4 :
     /* mask that interrupt */
     DISABLE INT4 , OUT h'0108' ;
     RET , OUT h'0109' ;

int5 :
     /* mask that interrupt */
     DISABLE INT5 , OUT h'010a' ;
     RET , OUT h'010b' ;

int6 :  /* Breakpoint and Single-step ISR */
     /* mask that interrupt */
     DISABLE INT6 , OUT h'010c' ;  /* mask interrupt 6 INT6 */
     CLI h'0ff' , OUT h'010d' ;    /* clear all interrupts */


/************** TEST for Breakpoint/Single-Step **************/
     AND R1 R0 , OUT h'010e' ;
     JMPC Z a0 , OUT h'010f' ; /* if Z=1 then breakpoint,Z=0 SS */
```

```
     CALL single1 , OUT h'0110' ;/* call single step program */
     JMP b0 , OUT h'0111' ; /*finish SS ISR , return to main progr */

a0:  SET HDOE HDSEL0 , OUT h'0112' ; /* set HD to output */
                        /* select DOR to HD output bus*/

     /* move out the source registers to HD */

     MOV DOR DIR , OUT h'0113' ; /* 0000 should come out next cycle*/
     MOV DOR AIR , OUT h'0114' ; /* 0000 */
     MOV DOR AOR , OUT h'0115' ; /* 0001 */
     MOV DOR ACH , OUT h'0116' ; /* 001f */
     MOV DOR ACL , OUT h'0117' ; /* 0000 */
     MOV DOR IIR , OUT h'0118' ; /* 0000 */
     MOV DOR BC  , OUT h'0119' ; /* 0007 */

b0:
     RET , OUT h'011a' ;
int7 :
     /* mask that interrupt */
     DISABLE INT7 , OUT h'011a' ;
     RET , OUT h'011b' ;
end ;


.T
         RCCCCCCCCCIIIIIIIIIIIICWRHHHHHHHHHHHHHHHHHHHHHHHHAAAAAAAAAAAAAAAA
         ECCCCCCCCCOOOOOOOONNNNSRDDDDDDDDDDDDDDDDDDDAAAAAADDDDDDDDDDDDDDDD
         S76543210765432107TTTBBB1111119876543210DDDDDDDDDDDDDDDDDDDDDD
         E                3210   543210          5432101111119876543210
         T                                            543210
         B
TIME
    1    00000000000000000000011100000000000000000000000000000000000000
    2    10000000000000000000011100000000000000000000000000000000000000
# bps0.stl file for single-stepping
# write the single-step mode bit hd12=1
   20    10000000000000000000001000100000000000000000000000000000000000
   21    10000000000000000000011100010000000000000000000000000000000000
# write into FIFO for single-step
   55    10000000000000000000011100000000000000000000ZZZZZZZZZZZZZZZZ
   56    10000000000000000000001000000000000000000000ZZZZZZZZZZZZZZZZ
   57    10000000000000000000011100000000000000000000ZZZZZZZZZZZZZZZZ
   58    10000000000000000000001000000000000000000001ZZZZZZZZZZZZZZZZ
   59    10000000000000000000011100000000000000000001ZZZZZZZZZZZZZZZZ
   60    10000000000000000000001000000000000000000010ZZZZZZZZZZZZZZZZ
   61    10000000000000000000011100000000000000000010ZZZZZZZZZZZZZZZZ
   62    10000000000000000000001000000000000000000011ZZZZZZZZZZZZZZZZ
   63    10000000000000000000011100000000000000000011ZZZZZZZZZZZZZZZZ
   64    10000000000000000000001000000000000000000100ZZZZZZZZZZZZZZZZ
   65    10000000000000000000011100000000000000000100ZZZZZZZZZZZZZZZZ
   66    10000000000000000000001000000000000000000101ZZZZZZZZZZZZZZZZ
   67    10000000000000000000011100000000000000000101ZZZZZZZZZZZZZZZZ
   68    10000000000000000000001000000000000000000110ZZZZZZZZZZZZZZZZ
   69    10000000000000000000011100000000000000000110ZZZZZZZZZZZZZZZZ
   70    10000000000000000000001000000000000000000111ZZZZZZZZZZZZZZZZ
   71    10000000000000000000011100000000000000000111ZZZZZZZZZZZZZZZZ
```

```
  95    1000000000000000000000010000000000000000001000ZZZZZZZZZZZZZZZZ
  96    1000000000000000000001110000000000000000001000ZZZZZZZZZZZZZZZZ
  97    1000000000000000000000010000000000000000001001ZZZZZZZZZZZZZZZZ
  98    1000000000000000000001110000000000000000001001ZZZZZZZZZZZZZZZZ
  99    1000000000000000000000010000000000000000001010ZZZZZZZZZZZZZZZZ
 100    1000000000000000000001110000000000000000001010ZZZZZZZZZZZZZZZZ
 101    1000000000000000000000010000000000000000001011ZZZZZZZZZZZZZZZZ
 102    1000000000000000000001110000000000000000001011ZZZZZZZZZZZZZZZZ
 103    1000000000000000000000010000000000000000001100ZZZZZZZZZZZZZZZZ
 104    1000000000000000000001110000000000000000001100ZZZZZZZZZZZZZZZZ
 105    1000000000000000000000010000000000000000001101ZZZZZZZZZZZZZZZZ
 106    1000000000000000000001110000000000000000001101ZZZZZZZZZZZZZZZZ
 107    1000000000000000000000010000000000000000001110ZZZZZZZZZZZZZZZZ
 108    1000000000000000000001110000000000000000001110ZZZZZZZZZZZZZZZZ
 109    1000000000000000000000010000000000000000001111ZZZZZZZZZZZZZZZZ
 110    1000000000000000000001110000000000000000001111ZZZZZZZZZZZZZZZZ
 135    1000000000000000000000010000000000000000010000ZZZZZZZZZZZZZZZZ
 136    1000000000000000000001110000000000000000010000ZZZZZZZZZZZZZZZZ
 137    1000000000000000000000010000000000000000010001ZZZZZZZZZZZZZZZZ
 138    1000000000000000000001110000000000000000010001ZZZZZZZZZZZZZZZZ
 139    1000000000000000000000010000000000000000010010ZZZZZZZZZZZZZZZZ
 140    1000000000000000000001110000000000000000010010ZZZZZZZZZZZZZZZZ
 141    1000000000000000000000010000000000000000010011ZZZZZZZZZZZZZZZZ
 142    1000000000000000000001110000000000000000010011ZZZZZZZZZZZZZZZZ
 143    1000000000000000000000010000000000000000010100ZZZZZZZZZZZZZZZZ
 144    1000000000000000000001110000000000000000010100ZZZZZZZZZZZZZZZZ
 145    1000000000000000000000010000000000000000010101ZZZZZZZZZZZZZZZZ
 146    1000000000000000000001110000000000000000010101ZZZZZZZZZZZZZZZZ
 147    1000000000000000000000010000000000000000010110ZZZZZZZZZZZZZZZZ
 148    1000000000000000000001110000000000000000010110ZZZZZZZZZZZZZZZZ
 149    1000000000000000000000010000000000000000010111ZZZZZZZZZZZZZZZZ
 150    1000000000000000000001110000000000000000010111ZZZZZZZZZZZZZZZZ
 175    1000000000000000000000010000000000000000011000ZZZZZZZZZZZZZZZZ
 176    1000000000000000000001110000000000000000011000ZZZZZZZZZZZZZZZZ
 177    1000000000000000000000010000000000000000011001ZZZZZZZZZZZZZZZZ
 178    1000000000000000000001110000000000000000011001ZZZZZZZZZZZZZZZZ
 179    1000000000000000000000010000000000000000011010ZZZZZZZZZZZZZZZZ
 180    1000000000000000000001110000000000000000011010ZZZZZZZZZZZZZZZZ
 181    1000000000000000000000010000000000000000011011ZZZZZZZZZZZZZZZZ
 182    1000000000000000000001110000000000000000011011ZZZZZZZZZZZZZZZZ
 183    1000000000000000000000010000000000000000011100ZZZZZZZZZZZZZZZZ
 184    1000000000000000000001110000000000000000011100ZZZZZZZZZZZZZZZZ
 185    1000000000000000000000010000000000000000011101ZZZZZZZZZZZZZZZZ
 186    1000000000000000000001110000000000000000011101ZZZZZZZZZZZZZZZZ
 187    1000000000000000000000010000000000000000011110ZZZZZZZZZZZZZZZZ
 188    1000000000000000000001110000000000000000011110ZZZZZZZZZZZZZZZZ
 189    1000000000000000000000010000000000000000011111ZZZZZZZZZZZZZZZZ
 190    1000000000000000000001110000000000000000011111ZZZZZZZZZZZZZZZZ
# write into FIFO second time around for single-step
 240    1000000000000000000000011100000000000000000000ZZZZZZZZZZZZZZZZ
 241    1000000000000000000000010000000000000000000000ZZZZZZZZZZZZZZZZ
 242    1000000000000000000001110000000000000000000000ZZZZZZZZZZZZZZZZ
 243    1000000000000000000000010000000000000000000001ZZZZZZZZZZZZZZZZ
 244    1000000000000000000001110000000000000000000001ZZZZZZZZZZZZZZZZ
 245    1000000000000000000000010000000000000000000010ZZZZZZZZZZZZZZZZ
 246    1000000000000000000001110000000000000000000010ZZZZZZZZZZZZZZZZ
 247    1000000000000000000000010000000000000000000011ZZZZZZZZZZZZZZZZ
```

```
248   10000000000000000000011100000000000000000000011ZZZZZZZZZZZZZZZZ
249   10000000000000000000011000000000000000000000100ZZZZZZZZZZZZZZZZ
250   10000000000000000000011100000000000000000000100ZZZZZZZZZZZZZZZZ
255   10000000000000000000010000000000000000000000101ZZZZZZZZZZZZZZZZ
256   10000000000000000000011100000000000000000000101ZZZZZZZZZZZZZZZZ
257   10000000000000000000010000000000000000000000110ZZZZZZZZZZZZZZZZ
258   10000000000000000000011100000000000000000000110ZZZZZZZZZZZZZZZZ
259   10000000000000000000010000000000000000000000111ZZZZZZZZZZZZZZZZ
260   10000000000000000000011100000000000000000000111ZZZZZZZZZZZZZZZZ
285   10000000000000000000010000000000000000000001000ZZZZZZZZZZZZZZZZ
286   10000000000000000000011100000000000000000001000ZZZZZZZZZZZZZZZZ
287   10000000000000000000010000000000000000000001001ZZZZZZZZZZZZZZZZ
288   10000000000000000000011100000000000000000001001ZZZZZZZZZZZZZZZZ
289   10000000000000000000010000000000000000000001010ZZZZZZZZZZZZZZZZ
290   10000000000000000000011100000000000000000001010ZZZZZZZZZZZZZZZZ
291   10000000000000000000010000000000000000000001011ZZZZZZZZZZZZZZZZ
292   10000000000000000000011100000000000000000001011ZZZZZZZZZZZZZZZZ
293   10000000000000000000010000000000000000000001100ZZZZZZZZZZZZZZZZ
294   10000000000000000000011100000000000000000001100ZZZZZZZZZZZZZZZZ
295   10000000000000000000010000000000000000000001101ZZZZZZZZZZZZZZZZ
296   10000000000000000000011100000000000000000001101ZZZZZZZZZZZZZZZZ
297   10000000000000000000010000000000000000000001110ZZZZZZZZZZZZZZZZ
298   10000000000000000000011100000000000000000001110ZZZZZZZZZZZZZZZZ
299   10000000000000000000010000000000000000000001111ZZZZZZZZZZZZZZZZ
300   10000000000000000000011100000000000000000001111ZZZZZZZZZZZZZZZZ
325   10000000000000000000010000000000000000000010000ZZZZZZZZZZZZZZZZ
326   10000000000000000000011100000000000000000010000ZZZZZZZZZZZZZZZZ
327   10000000000000000000010000000000000000000010001ZZZZZZZZZZZZZZZZ
328   10000000000000000000011100000000000000000010001ZZZZZZZZZZZZZZZZ
329   10000000000000000000010000000000000000000010010ZZZZZZZZZZZZZZZZ
330   10000000000000000000011100000000000000000010010ZZZZZZZZZZZZZZZZ
331   10000000000000000000010000000000000000000010011ZZZZZZZZZZZZZZZZ
332   10000000000000000000011100000000000000000010011ZZZZZZZZZZZZZZZZ
333   10000000000000000000010000000000000000000010100ZZZZZZZZZZZZZZZZ
334   10000000000000000000011100000000000000000010100ZZZZZZZZZZZZZZZZ
335   10000000000000000000010000000000000000000010101ZZZZZZZZZZZZZZZZ
336   10000000000000000000011100000000000000000010101ZZZZZZZZZZZZZZZZ
337   10000000000000000000010000000000000000000010110ZZZZZZZZZZZZZZZZ
338   10000000000000000000011100000000000000000010110ZZZZZZZZZZZZZZZZ
339   10000000000000000000010000000000000000000010111ZZZZZZZZZZZZZZZZ
340   10000000000000000000011100000000000000000010111ZZZZZZZZZZZZZZZZ
365   10000000000000000000010000000000000000000011000ZZZZZZZZZZZZZZZZ
366   10000000000000000000011100000000000000000011000ZZZZZZZZZZZZZZZZ
367   10000000000000000000010000000000000000000011001ZZZZZZZZZZZZZZZZ
368   10000000000000000000011100000000000000000011001ZZZZZZZZZZZZZZZZ
369   10000000000000000000010000000000000000000011010ZZZZZZZZZZZZZZZZ
370   10000000000000000000011100000000000000000011010ZZZZZZZZZZZZZZZZ
371   10000000000000000000010000000000000000000011011ZZZZZZZZZZZZZZZZ
372   10000000000000000000011100000000000000000011011ZZZZZZZZZZZZZZZZ
373   10000000000000000000010000000000000000000011100ZZZZZZZZZZZZZZZZ
374   10000000000000000000011100000000000000000011100ZZZZZZZZZZZZZZZZ
375   10000000000000000000010000000000000000000011101ZZZZZZZZZZZZZZZZ
376   10000000000000000000011100000000000000000011101ZZZZZZZZZZZZZZZZ
377   10000000000000000000010000000000000000000011110ZZZZZZZZZZZZZZZZ
378   10000000000000000000011100000000000000000011110ZZZZZZZZZZZZZZZZ
379   10000000000000000000010000000000000000000011111ZZZZZZZZZZZZZZZZ
380   10000000000000000000011100000000000000000011111ZZZZZZZZZZZZZZZZ
```

```
.T
        RCCCCCCCCIIIIIIIIIIIICWRHHHHHHHHHHHHHHHHHHHHHHHHAAAAAAAAAAAAAAAA
        ECCCCCCCCOOOOOOOONNNNSRDDDDDDDDDDDDDDDDDDAAAAAADDDDDDDDDDDDDDDD
        S7654321076543210TTTTBBB1111119876543210DDDDDDDDDDDDDDDDDDDDDDD
        E                 3210   543210          5432101111119876543210
        T                                              543210
        B
TIME
 1       0000000000000000000001110000000000000000000000000000000000000000
 2       1000000000000000000001110000000000000000000000000000000000000000
20       1000000000000000000000010000000011110100000000000000000000000000
21       1000000000000000000001110000000011110100000000000000000000000000
53       1000000000000000000001112ZZZZZZZZZZZZZZZZ0000000000000000000000000
# bps1.stl uses Z=1 for breakpoint ISR; HD12=0;


                *********The bps0.out file **********

*************************************************************************

                            O U T P U T   T A B L E

  P A C S I M     Ver. 1.09                      Tue Apr 04 15:43:42 1989
*************************************************************************

      CCC OOOO A AAAA AAAA FFFIB PPP LLL BBBB
      PPP CCCC D DDDD OOOO IIINR CCC CCC CCCC
      CCC 1173 O DDDD RRRR CIOTP 173 173 1173
      173 51:: E 1173 1173 DRRRT 1:: 1:: 51::
      1:: ::40   51:: 51::     E :40 :40 ::40
      :40 18     ::40 ::40     Q 8   8   18
      8   2      18   18       U         2
                 2    2        L
TIME
   1  000 0000 0 0000 0000 00001 000 000 0000
   2  000 0000 0 0000 0000 00001 000 000 0000
   3  000 0002 0 0000 0000 01000 001 000 0000
   4  001 0001 0 0000 0000 01000 002 000 0000
   5  002 0000 0 0000 0000 01000 050 000 0000
   6  050 0050 0 0000 0000 01000 051 000 0000
   7  051 0051 0 0000 0000 01000 052 000 0000
   8  052 0052 0 0000 0000 01000 053 000 0000
   9  053 0053 0 0000 0000 01000 054 000 0000
  10  054 0054 0 0000 0000 01000 055 000 0000
  11  055 0055 0 0000 0000 01000 056 000 0000
  12  056 0056 0 0000 0000 01000 057 000 0000
  13  057 0057 0 0000 0000 01000 058 000 0000
  14  058 0058 0 0000 0000 01000 059 000 0000
  15  059 0059 0 0000 0000 01000 05a 000 0000
  16  05a 005a 0 0000 0000 01000 05b 000 0000
  17  05b 005b 0 0000 0000 01000 05c 000 0000
  18  05c 005c 0 0000 0000 01000 05d 000 0000
  19  05d 005d 0 0000 0000 01000 05e 000 0000
  20  05e 005e 0 0000 0000 01000 05f 000 0000
  21  05f 005f 0 0000 0000 01000 060 000 0000
  22  060 0060 0 0000 0000 01100 061 000 0000
  23  061 0061 0 0000 0000 01100 062 000 0000
```

```
  24  062 0062 0 0000 0000 01100 063 000 0000
  25  063 0063 0 0000 0000 01100 064 000 0000
  26  064 0064 0 0000 0000 01100 065 000 0000
  27  065 0065 0 0000 0000 01100 066 000 0000
  28  066 0066 0 0000 0000 01100 067 000 0000
  29  067 0067 0 0000 0000 01100 068 000 0000
  30  068 0068 0 0000 0000 01100 069 000 0000
  31  069 0069 0 0000 0000 01100 06a 000 0000
  32  06a 006a 0 0000 0000 01100 06b 000 0000
  33  06b 006b 0 0000 0000 01100 06c 000 0000
***Due to the length of the file,the rest of the output is not shown ***


                **********The bps1.out file *************

*************************************************************************

                          O U T P U T   T A B L E

P A C S I M     Ver. 1.09                          Mon Apr 03 13:08:15 1989
*************************************************************************

      CCC OOOO M CC DI BBB B HHHH LLL BBBB
      PPP CCCC D CC ON RRR R DDDD CCC CCCC
      CCC 1173 O 73 RT EEE P 1173 173 1173
      173 51:: E ::  R AAA T 51:: 1:: 51::
      1:: ::40   40    KKK E ::40 :40 ::40
      :40 18           RRR Q 18   8   18
      8   2            EEE U 2        2
                       GGG L
                       973
                       :::
                       840
TIME
   1  000 0000 0 00 00 000 1 0000 000 0000
   2  000 0000 0 00 00 000 1 0000 000 0000
   3  000 0002 0 00 00 000 0 0000 000 0000
   4  001 0001 0 00 00 000 0 0000 000 0000
   5  002 0000 0 00 00 000 0 0000 000 0000
   6  050 0050 0 00 00 000 0 0000 000 0000
   7  051 0051 0 00 00 000 0 0000 000 0000
   8  052 0052 0 00 00 000 0 0000 000 0000
   9  053 0053 0 00 00 000 0 0000 000 0000
  10  054 0054 0 00 00 000 0 0000 000 0000
  11  055 0055 0 00 00 000 0 0000 000 0000
  12  056 0056 0 00 00 000 0 0000 000 0000
  13  057 0057 0 00 00 000 0 0000 000 0000
  14  058 0058 0 00 00 000 0 0000 000 0000
  15  059 0059 0 00 00 000 0 0000 000 0000
  16  05a 005a 0 00 00 000 0 0000 000 0000
  17  05b 005b 0 00 00 000 0 0000 000 0000
  18  05c 005c 0 00 00 000 0 0000 000 0000
  19  05d 005d 0 00 00 000 0 0000 000 0000
  20  05e 005e 0 00 00 000 0 007a 000 0000
  21  05f 005f 0 00 00 000 0 007a 000 0000
  22  060 0060 0 00 00 000 0 007a 000 0000
  23  061 0061 0 00 00 000 0 007a 000 0000
```

24	062	0062	0	00	00	000	0	007a	000	0000
25	063	0063	0	00	00	000	0	007a	000	0000
26	064	0064	0	00	00	000	0	007a	000	0000
27	065	0065	0	00	00	000	0	007a	000	0000
28	066	0066	0	00	00	000	0	007a	000	0000
29	067	0067	0	00	00	000	0	007a	000	0000
30	068	0068	0	00	00	000	0	007a	000	0000
31	069	0069	0	00	00	000	0	007a	000	0000
32	06a	006a	0	00	00	000	0	007a	000	0000
33	06b	006b	0	00	00	000	0	007a	000	0000
34	06c	006c	0	00	00	000	0	007a	000	0000
35	06d	006d	0	00	00	000	0	007a	000	0000
36	06e	006e	0	00	00	000	0	007a	000	0000
37	06f	006f	0	00	00	000	0	007a	000	0000
38	070	0070	0	00	00	000	0	007a	000	0000
39	071	0071	0	00	00	000	0	007a	000	0000
40	072	0072	0	00	00	000	0	007a	000	0000
41	073	0073	0	00	00	000	0	007a	000	0000
42	074	0074	0	00	00	000	0	007a	000	0007
43	075	0075	0	00	00	07a	0	007a	000	0007
44	076	0076	0	00	00	07a	0	007a	000	0007
45	077	0077	0	00	00	07a	1	007a	000	0007
46	07a	007a	0	00	00	07a	0	007a	000	0007
47	07b	007b	0	00	01	07a	0	007a	000	0007
48	00e	000e	0	00	01	07a	0	007a	000	0007
49	10c	010c	0	00	00	07a	0	007a	000	0007
50	10d	010d	0	00	00	07a	0	007a	000	0007
51	10e	010e	0	00	00	07a	0	007a	000	0007
52	10f	010f	0	00	00	07a	0	007a	000	0007
53	112	0112	1	00	00	07a	0	000f	000	0007
54	113	0113	1	00	00	07a	0	000f	000	0007
55	114	0114	1	00	00	07a	0	0000	000	0007
56	115	0115	1	00	00	07a	0	0000	000	0007
57	116	0116	1	00	00	07a	0	1000	000	0007
58	117	0117	1	00	00	07a	0	001f	000	0007
59	118	0118	1	00	00	07a	0	0000	000	0007
60	119	0119	1	00	00	07a	0	0000	000	0007
61	11a	011a	1	00	00	07a	0	0007	000	0007
62	07c	007c	1	00	00	07a	0	0007	000	0007
63	07d	007d	1	00	00	07a	0	0007	000	0007
64	07d	007d	1	00	00	07a	0	0007	000	0007
65	07d	007d	1	00	00	07a	0	0007	000	0007
66	07d	007d	1	00	00	07a	0	0007	000	0007
67	07d	007d	1	00	00	07a	0	0007	000	0007
68	07d	007d	1	00	00	07a	0	0007	000	0007
69	07d	007d	1	00	00	07a	0	0007	000	0007
70	07d	007d	1	00	00	07a	0	0007	000	0007

Programmable Peripheral Application Note 010

PAC1000 Introduction

By Chris Jay and David Fong

Abstract

The PAC1000 programmable peripheral controller is the first of a generation of devices intended for applications in high end embedded control. Understanding the device architecture and using its support tools require some practical experience before a full system design is attempted.

This application note is intended to introduce the device and its architecture along with the support software tools to the systems designer. Finally, some simple applications are leveled at common problems found in system design.

Introduction

The PAC1000 has many applications in digital systems where high speed processing, interface or control is required. The two roles of the device are in a standalone mode where the PAC1000 is programmed to control data flow to or from other systems, or as a high speed peripheral working with a host microprocessor. Frequently, many systems designers cannot find the ideal solution to their requirements in a standard chip. The designer may look at creating the required function from discrete logic, a combination of a number of PAL/EPLD devices, Programmable Gate Array (PGA) products or standard gate array. In each alternative, the designer is trying to reduce the chip count of the system solution and hence increase its reliability and reduce assembly costs.

The discrete TTL or CMOS logic solution to a systems design is considered by some to be an old fashioned approach but still popular with many digital design engineers. However, designs using this technology can quickly escalate in chip count as the development progresses and once a system is designed it is very difficult to modify because the finished printed circuit board contains devices that cannot be re-programmed or altered in any way. Also, a revision or system upgrade will require a new printed circuit board design.

The PAL/EPLD solution reduces the chip count over a solution that uses discrete logic but still many devices are used because the PAL/EPLD products are not very register intensive. Small subsystems such as FIFO or a STACK require a number of PAL/EPLD devices and additionally require some additional chips. An alternative solution would be to use additional dedicated chips like FIFO, ALU and SRAM, leaving the PLD/EPLD devices to handle the glue, interface and small state machine functions. The Programmable Gate Array brings the system down to a possible acceptable level but system logic still has to be defined and routed in the logic cells and a number of PGA devices have to be designed such that they all work together. Nevertheless, in the case of the programmable solution, subsystems such as STACK, ALU, REGISTER FILES etc., might still need to be configured in the gates and registers of these devices. This can cause an escalation in the quantity of these chips used in the final system, because PLDs and PGAs are not good vehicles for integration at the subsystem to system level. In a gate array design the turn-around time is longer than the programmable solution, and because the device is not re-programmable there is a high level of risk in going to a gate array solution. Also, the high 'up front' Non Recurring Engineering charges NRE can rule out the use of gate array.

The Programmable Standalone Controller offers the most likely solution to the problem facing the systems designer. Very often both the PAC1000 is used with programmable logic devices to effect an overall solution. For example in some modes of operation PLDs are used for address decoders to select and gate the host interface control lines such as CSB, RDB, and WRB. By bringing the package count of the system down to its lowest

Introduction (Cont.)

level the design cycle time reduces, so minimizing the overall time to market of the final product. The reason for this is that the PAC1000 already contains the subsystems necessary for a fully functional system design, and being programmable, it can be adapted to perform most functions required from systems devices.

The PAC1000 comprises elements such as FIFO, ALU, register files, STACK, microcode store, loop and breakpoint counters, special registers and interface logic all interconnected by a general purpose internal bus structure. The instructions that control data flow are contained in the EPROM section of the microcontrol store. These instructions are entered into the system by the designer as assembly or high level language code. There also exists a microcode entry level for those designers who are used to microprogrammable designs. Designing with the PAC's software support tools is very similar to writing code for microprocessors. The end result is an assembled listing which can be simulated prior to programming into the PAC1000 device's on chip EPROM. The difference between microprocessors, conventional microcontrollers and the PAC1000 device is found its ability to execute instructions in parallel, and to offer the designer a flexible architecture. Microcontrollers and microprocessors function on single operations of execution, but the PAC1000 executes three instructions in parallel during the current clock cycle. In this way the PAC1000 device needs fewer EPROM locations to store the code which performs a given function. In addition high functional speeds can be obtained because the device can execute those instructions at the clock rate of the system.

PAC1000 Device Architecture

The PAC1000 device architecture can be divided into three subsystems, see Figure 1a; a CPU section that is similar to those found in microprocessors, a host interface, and a programmable instruction control unit. The instruction register can be clearly identified with its three output sections of control, output and CPU Operation Definition. Figure 1b illustrates a more detailed diagram of the system than Figure 1a, clearly identifying the sub structures of the three subsystems. The different sections of the PAC1000 are interconnected to each other by internal buses and convey data and instructions to and from each other. Communication to and from the outside world is achieved through various input and output registers, and a Command/Data FIFO.

The Control Unit

The control unit is constructed around a 1K deep 64-bit wide EPROM, see Figure 1b. The 64-bit wide instructions are programmed in the EPROM section and are accessed and executed on each clock cycle. The input RESET causes the PAC1000 to access and execute the first instruction at location 000H of EPROM. On each execution cycle, the Instruction Register shown in Figure 1a will contain three control operatives, a next address instruction to the control section, an output instruction and CPU instruction. The other inputs to the control unit include interrupts and condition codes. There are four external and four internal interrupts that can be enabled under programmed control. These can generate a branch to an interrupt service routine that results from a rising edge applied to the external interrupt input. For interrupts INT0, INT1, INT2, and INT3 there are four locations 008H, 009H 00AH and 00BH respectively. These are the vectored addresses at which processing will continue in the presence of one of these active interrupts. At the interrupt location a jump to an interrupt service routine should be inserted. For example, the occurrence of INT0 will divert processing to location 008H, that location may contain a JMP 100H, where 100H is the address where the service routine for INT0 should reside. The internally generated interrupts are INT4, INT5, INT6 and INT7 which divert processing to locations 0CH, 0DH, 0EH and 0FH respectively. Details of their allocated function is given in the PAC1000 data sheet. In addition there are eight condition code inputs CC[7:0], shown alongside the INT[3:0] inputs in Figure 1b. These inputs can be tested individually under program control. The combination of Next Instruction Definition, Interrupt and Condition Code

The Control Unit (Cont.)

input direct the flow of the program and hence the execution of instructions contained in the EPROM section. The CASE logic is used in the controller section to enable CASE statements to be executed on condition code groups. The eight condition code inputs may be divided into two four bit groups. Case group zero CG0 comprises CC0, CC1, CC2 and CC3. Case group 1 CG1 comprises CC4, CC5, CC6 and CC7. A further two case groups CG2 and CG3 test flag registers (see Table 1). These condition code inputs may be tested individually or tested in a group. When tested in a four bit group, a one-of-sixteen branch will occur, as specified by the CASE instruction.

The current status of the PAC1000 is kept in the sixteen bit status register. STAT0–STAT11 give twelve status bits with four extra bit locations for future development. Table 2 shows the assignment of each register.

Figure 1a. PAC1000 Programmable Peripheral Controller Single Cycle Control Architecture

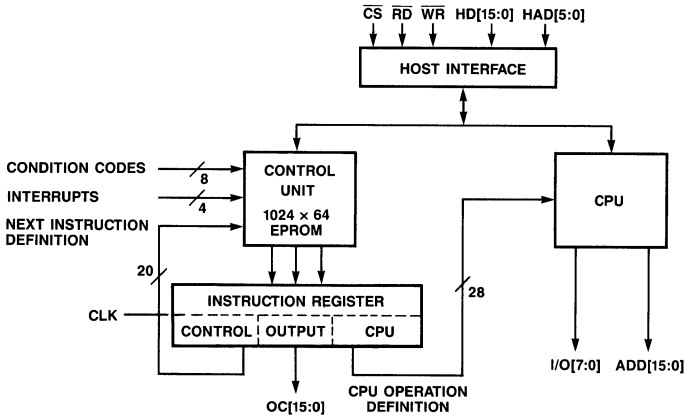
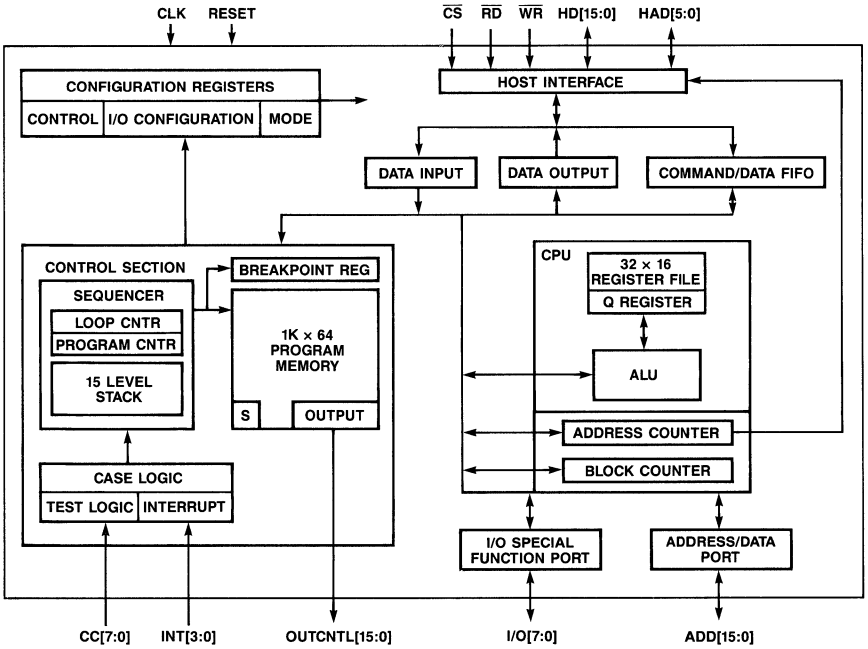


Figure 1b. PAC1000 Programmable Peripheral Controller Block Diagram



4

Table 1. CASE Group Assignments

Condition Code	CASE
CC0, CC1, CC2, CC3	CASE Group 0
CC4, CC5, CC6, CC7	CASE Group 1
S, O, Z, CY.	CASE Group 2
INTR, BCZ, FIOR, FICD.	CASE Group 3
FIXP, ACO, STKF, FIIR, DOR, INTR	N/A

Table 2. Status Register

0	0	WSI Reserved	S11	S10	S9	S8	S7	S6	S5	S4	S3	S2	S1	S0

S11 — Security Bit, High is Active Security On, Low is No Security.
S10 — Scan Mode, High is Active On, Low is No Scan Mode.
S9 — FIXP FIFO Exception Occurs When a Command is Written, a Low Means No Exception.
S8 — FIIR FIFO Input Ready When There is at Least One Location Vacant.
S7 — CY Set High When the Result of a CPU Operation Generated a Carry.
S6 — Z Set High When the Result of a CPU Operation is Zero.
S5 — O Set When an Overflow Has Occurred During a Two's Complement Operation.
S4 — S Sign Bit Set to One When the Result is a Negative Number.
S3 — Stack Full Flag. Set When the Stack is Full.
S2 — Breakpoint Flag is Set When the Address in the Breakpoint Register is Equal to the Address in the Program Counter.
S1 — BCZ is Set When the Block Counter Reaches Zero.
S0 — ACO Address Counter All Ones Flag is Set When the Address Counter Reaches the Maximum Count.

The Control Unit (Cont.)

A single internal counter is provided for loop control, this is part of the control section, and is shown in Figure 1b. If a FORLOOP is executed the loop counter is loaded and the instructions within that loop are executed until the counter has decremented to zero. The loading of this counter is transparent to the designer in the respect that the FORLOOP instruction automatically performs loading and counting.

A fifteen level stack is incorporated to hold the return address of the main program when a subroutine call or interrupt service routine is being executed. The address of the next sequential instruction to be executed is pushed onto the stack. The stack is also used for LOOP NESTING. There is only one loop counter in the PAC1000 but nested FORLOOP instructions are possible because the current contents of the loop counter is saved in the stack when the next subsequent loop in the next is entered. When leaving the loop the stack is popped to return the old count back into the loop counter thus preserving its original contents. When the stack becomes full a status flag STKF is set in the sixteen bit status register and an interrupt level 7 is generated.

To enable a debugging facility a register called the breakpoint register is included in the microcode section. When the contents of the program counter is equal to that of the breakpoint register an interrupt level six is generated. For debugging purposes a level six interrupt service routine should be written to perform diagnostic tests within the system.

Host Interface

The host interface section has been designed to easily integrate into a CPU based system. When the PAC1000 is used in the peripheral mode, the flow of data or commands to its internal registers may be achieved through an internal FIFO. Standard microprocessor signals of chip select CSB, read RDB and write WRB (active LOW CS,

Host Interface (Cont.)

RD and WR) are accompanied by a sixteen bit Host Data and a six bit Host Address bus. Table 3 gives the conditions governing the mode setting for both standalone and peripheral mode. The logic condition of HDSEL0 and HDSEL1 in the control register will determine the mode of the PAC1000 operation. Bit positions in this register can be set or reset under program control.

A detailed block diagram of the PAC1000 is given in Figure 2 which illustrates the internal structure of the control section, processor section and interface. Data flow from the host processor data inputs HD0–HD15 to the internal 16-bit bus can be achieved through the FIFO section. The FIFO is eight locations deep and twenty-two bits wide. To transfer data words to the registers in the CPU section the host processor uses the chip select, write and HAD inputs. The address of the register is set up on the five HAD lines (this selects one of 32 registers) then the write and chip select lines are driven LOW. The data on the HD lines plus the register address is loaded into the FIFO. An additional bit called the FICD bit is loaded through HAD5 at the same time as address HAD[0–4] and the host data lines HD[0–15]. This is the FIFO Command/Data bit and must be LOW to signify that the sixteen bit word on HD[0–15] is data. If it is set HIGH, the least significant ten bits of that data will be used as an address pointer to the microcoded EPROM. In this way the host system can direct PAC1000 processing to a defined microcoded address. This is a powerful feature that enables dynamic context switching of PAC1000 under supervision of the host processor. The FIFO exception flag FIXP will be set if the information residing in the FIFO was misdirected (if it were treated as a control word when the FICD flag labeled it as data or if the opposite condition prevailed).

Using the FIFO is the only method in which the host can communicate with the PAC1000 using the active LOW chip select CSB and the write input WRB. The DOR and DIR are Data Output and Data Input registers and are available to convey data to and from the internal sixteen bit bus but do not respond to CSB and WRB. The DIR would be used in a synchronous system because, when it is enabled by setting the DIREN flag (see Table 4), data is latched on the rising edge of each clock signal. The data contents of the DOR register may be directed to the host data outputs if all inputs CSB, WRB and RDB are inactive and HDSEL0 and HDSEL1 are 1 and 0 respectively, see Table 3. The use of the DIR and DOR register is intended more for synchronous communication whereas the FIFO is intended primarily for asynchronous systems or synchronous peripheral interface. The flags FIIR and FIOR are the FIFO Input Ready and FIFO Output Ready respectively, these flags can be tested so no overwriting of data will occur. Figure 3 shows the I/O Port and Special Functions. The FIIR register can be directed to the output I/O_7 through a multiplexer so it can be tested externally by the host system.

Table 3. Host Interface Function Table

HDSEL0	HDSEL1	$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	HAD5	HAD[0–4]	HD[15–0]	OPERATION
0	0	0	1	0	0	Register Address	Data	Write Data to FIFO
0	0	0	1	0	1	X	Command	Write Command to FIFO
0	0	0	0	1	0	00100	X	Reset FIFO
0	0	0	0	1	0	00011	X	Reset Status Register
0	0	0	0	1	0	00010	X	Read Program Counter
0	0	0	0	1	0	00001	X	Read Status Register
0	0	0	0	1	0	00000	X	Read Data Output Register
1	0	1	1	1	X	X	X	Data Output Register
0	1	1	1	1	X	X	X	Status Register
1	1	1	1	1	X	X	X	Program Counter

Figure 2. PAC1000 Detailed Block Diagram

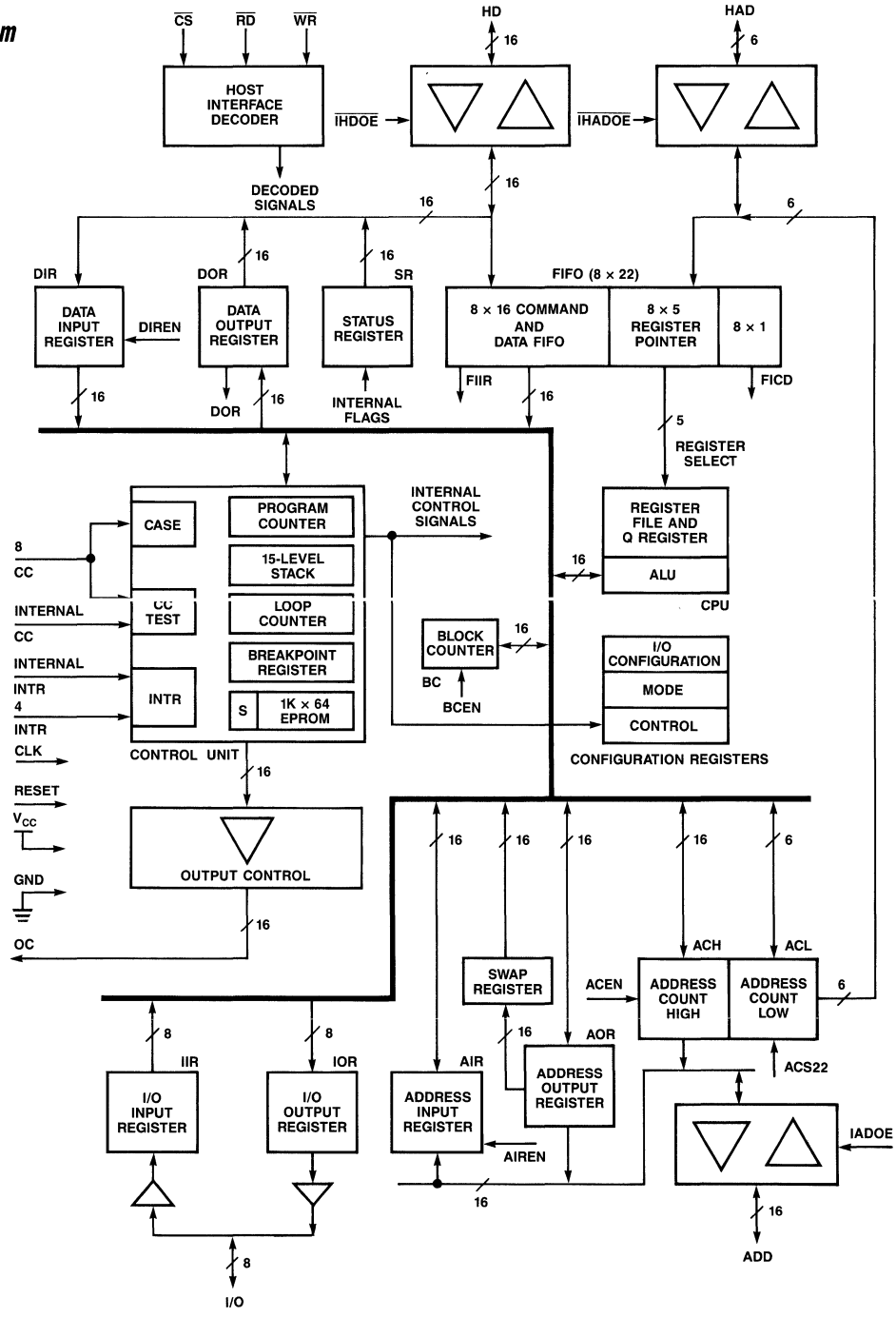


Figure 3. I/O Port and Special Functions

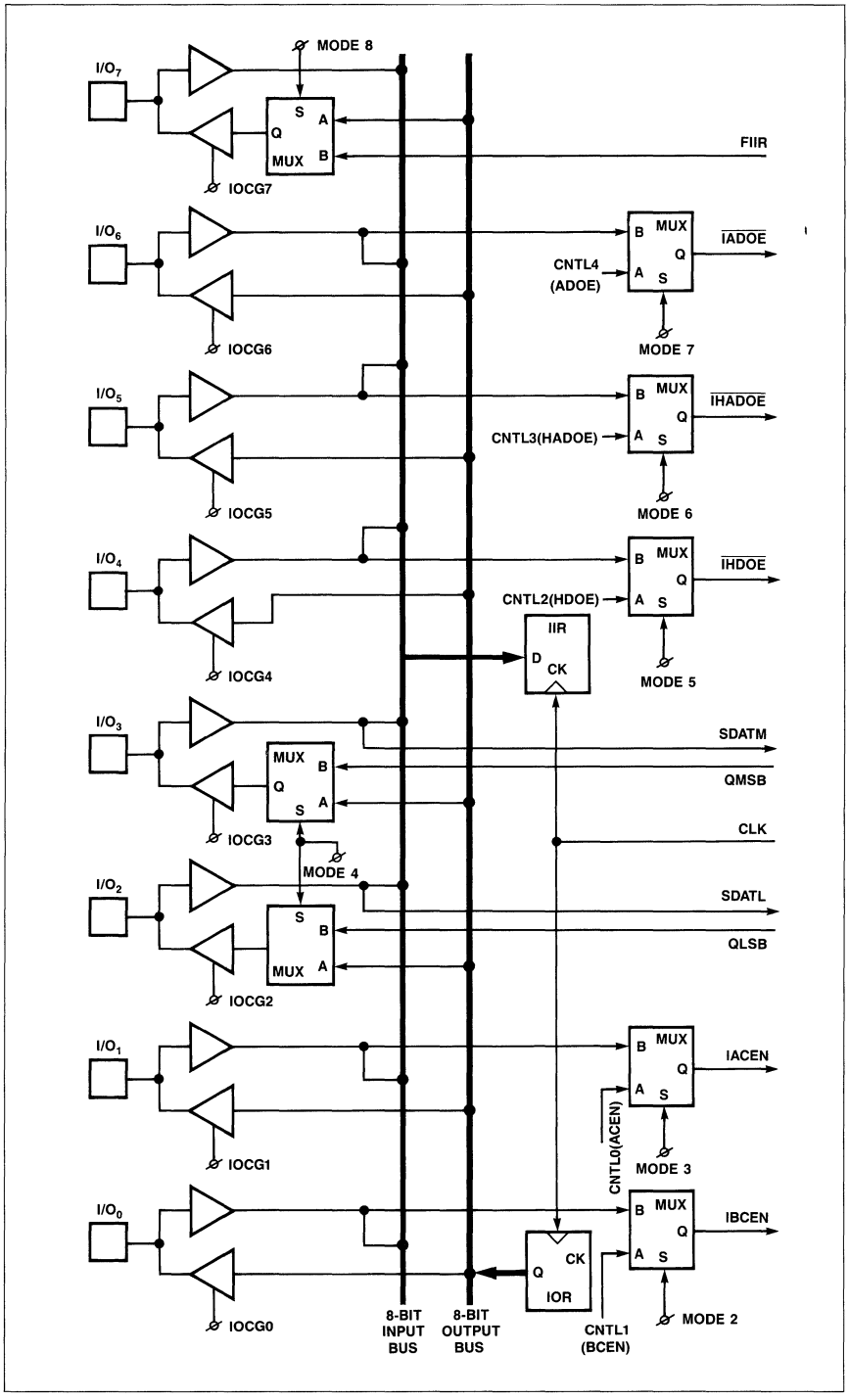


Table 4. Control Register

CTRL9	CTRL8	CTRL7	CTRL6	CTRL5	CTRL4	CTRL3	CTRL2	CTRL1	CTRL0
ASEL	AIREN	DIREN	HDSEL1	HDSEL0	ADOE	HADOE	HDOE	BCEN	ACEN

ASEL — Selects Which Source Will Write to the Address Bus
1 = Address Counter. 0 = Address Output Register.

AIREN — Enables/Disables Writing to the Address Input Register by the Address Bus.
1 = Enabled. 0 = Disabled.

DIREN — Enables/Disables Writing to the Data Input Register.
1 = Enabled. 0 = Disabled.

HDSEL1
HDSEL0 — Decoded to Select Which Source Will be Connected to the Host Data Bus (See Table 3.).

ADOE — Selects Direction of the Address Bus
1 = Output. 0 = Input.

HADOE — Selects Direction of Host Address Bus (HAD).
1 = Output. 0 = Input.

HDOE — Selects Direction of Host Data Bus for Next Clock Cycle.
1 = Output. 0 = Input.

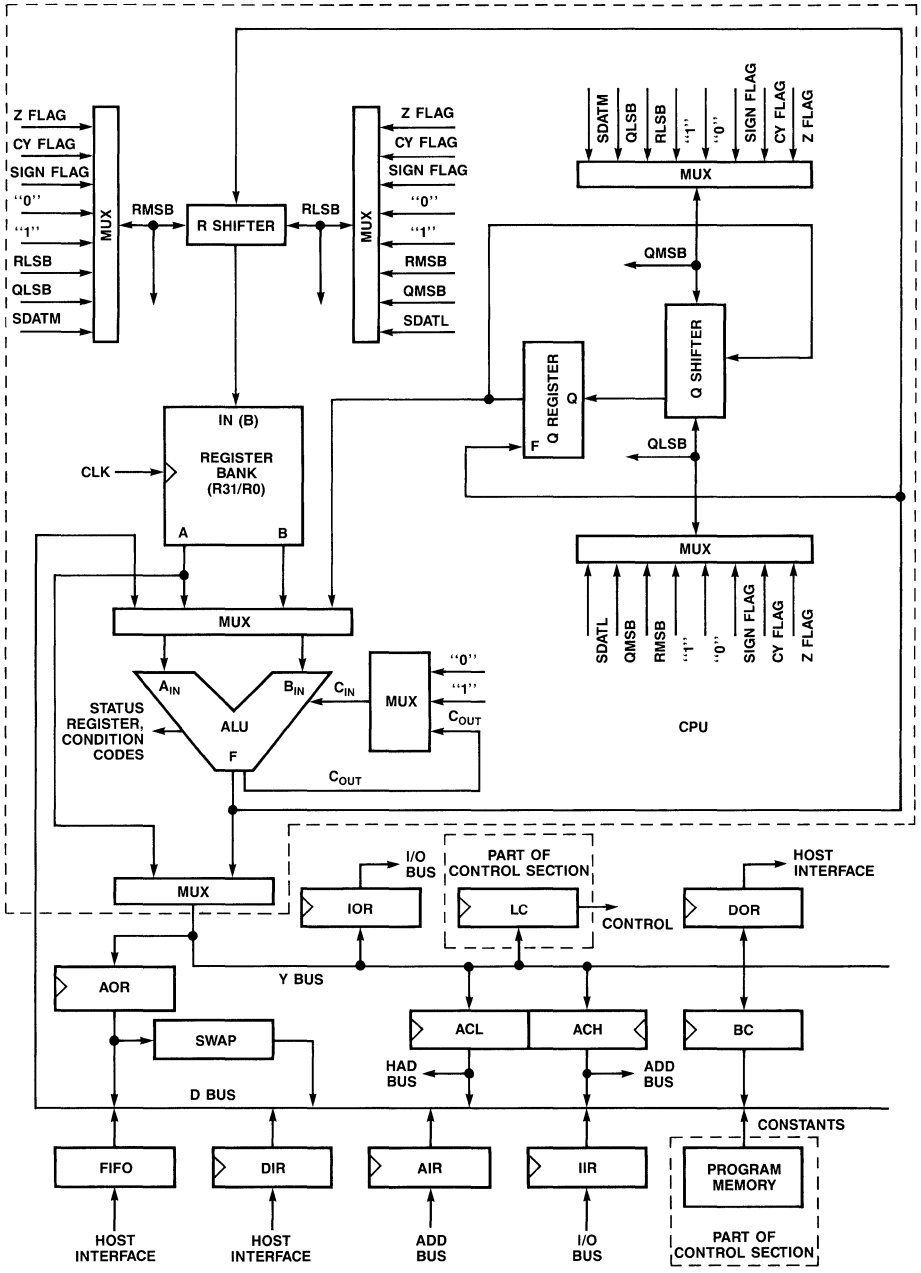
BCEN — Enables/Disables Block Counter Before Next Clock Edge.
1 = Enabled. 0 = Disabled.

ACEN — Enables/Disables Address Counter Before Next Clock Edge.
1 = Enabled. 0 = Disabled.

Central Processing Unit

The section that deals with data processing is the central processing unit. This comprises a sixteen bit wide ALU with a 32 x 16 bit register file. One other special purpose register Q and an R shifter circuit make up this section. The Q register is sixteen bits wide and can be used for data shifting. Figure 4 shows the ALU and register structure of the CPU section. The ALU is in the path of the register outputs such that arithmetic and logic functions may be executed on the contents of any one of the 32 general registers. The output of the ALU passes data back to the selected register through the R shifter. In this logic circuit, data may be shifted either left or right, one position, before being written back into the register file. The output of the ALU can also drive data to registers such as the DOR register. A multiplexer can select either the ALU or the R0–R31 register output. The loop counter LC can be loaded from this multiplexer enabling the contents of a register to determine how many program loops are to be executed. This loop counter can be loaded from the EPROM to give a fixed number of loops or from a register at program 'run time.' In this event, the number of times a loop is executed can be made programmable. Other registers on this bus are AOR, Address Output Register, the IOR, Input Output Register, the ACL and ACH low and high address counters and the BC Block Counter. The ACL counter has a six bit resolution and the ACH counter has sixteen. When enabled by ACEN, the ACH counter will increment on the rising edge of each clock cycle. The default value is for a sixteen bit count. To enable a twenty-two bit count where the ACL takes on the six least significant of the twenty-two bits. The ACS22 flag must be set, to enable the clocking of these counters. This is transparent to the software because once enabled the counters will clock at the system clock rate. However, they can be turned on and off from the microcoded instruction of enable SET ACEN, or disable RESET ACEN, also counting can be influenced by register loading.

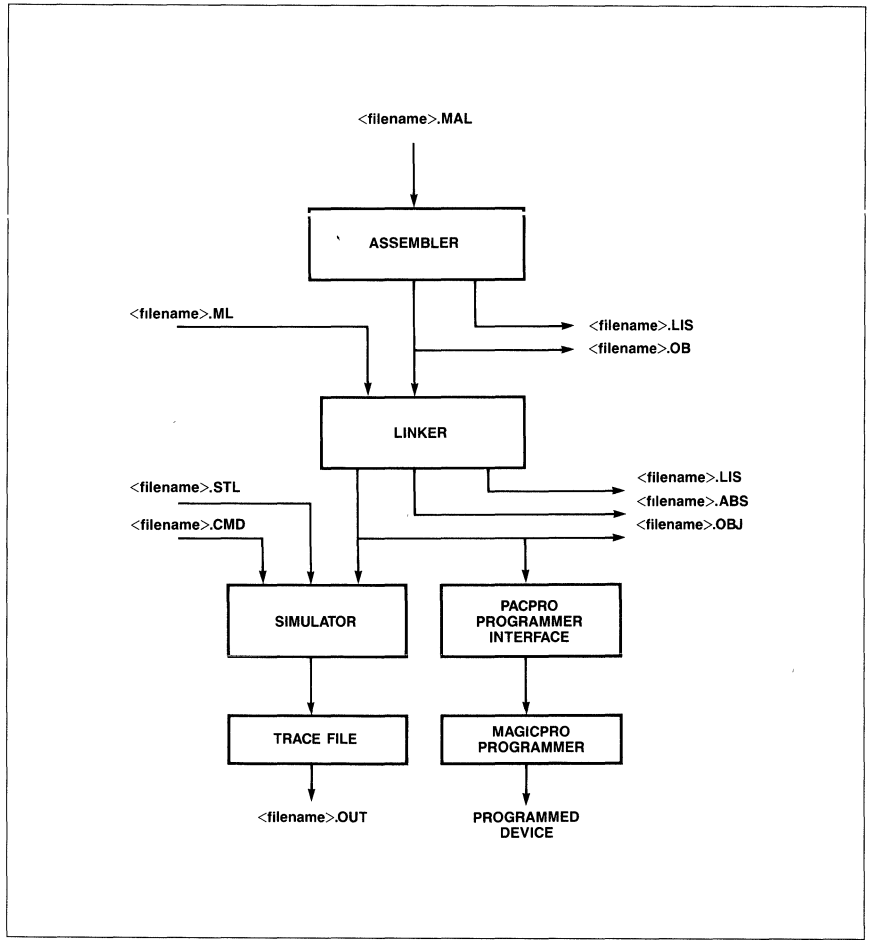
Figure 4. PAC1000 ALU and Registers Structure



Support Software

The PAC1000 device is supported with development software that can run in an IBM PC/XT or AT computer. The main tools that the designer will use are the assembler, the linker and the simulator. These support programs are run from a WSI menu called WISPER that has been designed to make software development a simple process. The designer can select the assembler from the menu and assemble his source program. After assembly the program must be linked. The linker program is designed for those system designers who build their software up from a number of modules. Figure 5 illustrates the flow from original source code entry through the linker to a simulated output. The linker will take these modules and combine them into one object program. On completion of assembly and linking the program may be checked by the simulator. The use of the simulator removes the need for EPROM programming and in-circuit testing during the design cycle and gives the designer a fairly high level of confidence that the program will function as intended. The simulator will take the bit pattern format that was generated during assembly and apply a command and stimulus file to the program. The result will be a series of waveforms that appear on the screen of the PC and is similar to that of a logic analyzer display. A table of vectors is also generated for those signals that are traced from the command file. These vectors can be printed out for analysis and verification.

Figure 5. Program Flow From Assembly Input to Simulated Output



Microcoded EPROM Section

A further aid to the design entry is the ability to mix high level, assembler and microcode mneumonics so designers can use the entry level that they feel the most comfortable with. Most of the applications example given below are written in a high level 'C' like language but some assembler instructions are also incorporated.

In systems applications such as Direct Memory Access (DMA), it is required to output the contents of a counter to address memory and then increment it. This is implemented in the PAC1000 high level language syntax as:

```
AOR := R0    ; /*CONTENTS OF R0
               GOES INTO THE AOR*/

R0 := ++R0   ; /*REGISTER R0 IS
               INCREMENTED BY ONE*/
```

For efficiency these two instructions may be combined into one line of code, which is executed in one clock cycle:

```
AOR := R0 := ++ R0 ;
  /*COMBINING THE TWO OPERATIONS*/
```

The contents of R0 will be passed to the Address Output Register and will be incremented by the ALU.

Where AOR is the address output register and R0 is one of the thirty-two, 16-bit general purpose registers. The '/*' symbol delimits the comment field boundary.

With a PAL/EPLD/PGA approach the designer would be required to spend much valuable time configuring a loadable binary counter, with a 3-State output capability.

In applications such as digitizer/plotter systems, x,y coordinates have to be quickly summed or subtracted many times to register cursor movements and position. This requires repetitive arithmetic operations. In this application vector addition on two or more sixteen bit words can be defined as two instructions:

```
R0 := R0 + R1 ;
AOR := R0 ;
```

Combining these instructions together:

```
AOR := R0 := R0 + R1 ;
```

With conventional programmable logic an ALU function would have to be designed or a dedicated custom chip used with the programmable logic part used as the data I/O controller. The key point of this issue is that complex logic functions are simply written as a few single lines of statements. Moreover, a combination of functions may be grouped in a single line. These include a microcontrol directive such as a branch, call to subroutine or JUMP on condition, an ALU function such as increment or add, and an output control command. There are sixteen output control lines which can be driven active on each clock cycle. The composite of the three commands are:

```
LABEL:   JMPNC CC7 LABEL ,
         R0 := R0 + 1 , OUT 'HOLD' ;
```

The function of this line of code would be to wait until the condition code input of CC7 went active before the next instruction is executed. At the same time the contents of R0 would be incremented and the output control lines would be driven with a sixteen bit code called HOLD. An equates option 'equ' is used to define uniquely a sixteen bit pattern called HOLD. The assembler encodes an equate statement to allow meaningful words to be used in output control statements. Some examples of this are:

```
HOLD equ H'FFFF' ;
  /* HOLD IS SET AS HEX FFFF */

ENBL equ H'EFFF' ;
  /* ENBL IS SET AS HEX EFFF */
```

The equates directive should be declared at the start of the program before any actual code is written.

4

Applications Programs

The depth of the microcontrol store is 1K of 64-bit wide words. One 64-bit instruction is executed on each clock cycle. The instruction word is subdivided into three commands: an output control command, a command to the processor section and a next address command to the microcoded memory. Figure 1a shows the Instruction Register with its contents of control, output and CPU commands. The control unit will also respond to condition code inputs and interrupts. An example of output control and response to condition codes is in a handshake loop. The output stimulus can be to set one of the control outputs OC[15:0] and wait for a response to a condition code input CC[7:0]. Under program control a conditional JUMP to a location could result if the bit tested were set. Otherwise linear programming could continue.

The first applications program below demonstrates the use of condition code zero CC0 to test for a start condition. When the input is LOW, the program loops continually testing CC0. When the host raises CC0, the program performs a double precision addition. The sum is available at the data output register DOR.

```
segment pacdes01 :

/* PROGRAM TO PERFORM DOUBLE PRECISION ADDITION ON THE REGISTER*/
/* CONTENTS OF R1,R0: R3,R2 THE CARRY OF THE LEAST SIGNIFICANT */
/* WORD ADDITION IS CONTAINED IN THE CP REGISTER AND IS USED IN*/
/* THE SECOND HALF OF THE 32 BIT ADDITION.                     */
/*                                                             */
/*              PIN FUNCTIONAL DESIGNATIONS.                   */
/*                      INPUTS.                                */
/*                                                             */
/*              CC0 - ACTIVE HIGH - START 32-BIT ADDITION      */
/*              /CS - ACTIVE LOW  - PAC1000 CHIP SELECT        */
/*              /RD - ACTIVE LOW  - READ A REGISTER FROM HOST  */
/*              HAD[5:0] - INPUTS TO SELECT DOR REGISTER FROM  */
/*              HOST INTERFACE                                 */
/*                                                             */

HOLD:    JMPNC  CC0 HOLD :       /*WAIT FOR START CONDITION  */

         R0 := H'F830' ;         /*LOAD REGISTERS WITH DATA  */
         R1 := H'982F' ;         /*R0 AND R2 CONTAIN THE     */
         R2 := H'A309' ;         /*LEAST SIGNIFICANT WORD OF */
         R3 := H'4500' ;         /*THE 32 BIT LONG WORD AND  */
                                 /*R1 AND R3 CONTAIN THE MOST*/
                                 /*SIGNIFICANT WORD          */

              R5 := R1 ;
              R4 := R0 :
              DOR := R0 := R0 + R2 ; /*LOAD DOR REGISTER*/
              R1 := R1 + R3 + CP ;

LOOP1:   JMPNC DOR LOOP1 :       /* WAIT FOR HOST TO READ DOR */

              DOR := R1 :        /* LOAD MOST SIG WORD INTO DOR */

LOOP2:   JMPNC DOR LOOP2 :       /* WAIT FOR HOST TO READ DATA */

FIN:     JMP HOLD :              /*END OF THE CYCLE*/

end :
```

Applications Programs (Cont.)

The program adds the contents of R0 and R2, then R1 and R3 and the CARRY bit. In the next design example, double precision subtraction is performed and this time the CY flag will hold the borrow bit. This design example is more practical than the example above because instead of performing arithmetic on fixed values the register file may be loaded from a source. The configuration of the PAC1000 is in the peripheral mode and data is loaded into the FIFO. CC0 is monitored and, when active, is a signal to the PAC1000 that data has been loaded. The FIFO is unloaded into the registers by the series of instructions:

```
FOR 3    ; /*EXECUTE THE LOOP
           FOUR TIMES*/
RDFIFO   ; /*UNPACK DATA FROM
           THE FIFO*/
ENDFOR ; /*END THE FORLOOP */
```

This section of the program performs a read operation on the FIFO four times. In any FORLOOP N, where N is an integer value, the number of times the loop is executed is N + 1 times.

```
segment pacdes02 ;

/*PROGRAM TO PERFORM DOUBLE PRECISION SUBTRACTION ON REGISTER  */
/*CONTENTS R1, R0 ; R3, R2 THE BORROW FLAG IS CONTAINED IN THE */
/*CP REGISTER DURING THE SECOND HALF OF A 32-BIT SUBTRACTION   */
/*                                                            */
/*       PIN FUNCTIONAL DESIGNATIONS                          */
/*              INPUTS                                        */
/*       CC0 - ACTIVE HIGH - START PROGRAM                    */
/*       /CS - ACTIVE LOW  - PAC1000 CHIP SELECT              */
/*       /WR - ACTIVE LOW  - FIFO WRITE                       */
/*       /RD - ACTIVE LOW  - READ A REGISTER FROM HOST INTERFACE*/
/*       HAD[5:0] - INPUTS TO SELECT A REGISTER FROM THE HOST */
/*                          INTERFACE                         */
/*       HD[15:0] - DATA INPUTS TO FIFO THROUGH HOST INTERFACE */
/*                                                            */

HOLD:   JMPNC   CC0 HOLD ; /*WAIT FOR START CONDITION EMPTY    */
        FOR 3  ;           /*THE FOUR LOCATIONS OF THE FIFO   */
        RDFIFO ;           /*LOADED THROUGH THE HOST INTERFACE */
        ENDFOR ;           /*SECTION OF THE PAC1000          */
                R5 := R1 ;              /*SAVE R1 CONTENTS IN R5*/
                R4 := R0 ;              /*SAVE R0 CONTENTS IN R4*/
                DOR := R0 := R0 - R2 ; /*SUBTRACT LSW PROPAGATE*/
                R1 := R1 - R3 - CP ;   /*THE BORROW INTO THE CP*/

                DOR := R0 ;             /*LOAD DOR WITH MSW    */

LOOP1:  JMPNC DOR LOOP1 ;

                DOR := R1 ;             /*LOAD DOR WITH MSW    */

LOOP2:  JMPNC DOR LOOP2 ;

        JMP HOLD ;                      /*END OF PROGRAM       */
end;
```

Applications Programs (Cont.)

The next program shows a multiply routine. Although there is no dedicated multiplier in the PAC1000, multiplication can be achieved by shifting and adding. The MUL instruction is a MACRO command that is expanded when assembled into a loop of shift and add instructions. The RDFIFO instruction is used to pass the data from the host to the PAC, which is configured as a peripheral. In the example the contents of R0 and R1 are multiplied and the product is available in registers R1 and R2, where R2 contains the most significant word and R1 the least significant.

```
segment pacdes03 ;

HOLD:   JMPNC    CC0 HOLD ;      /*WAIT FOR START CONDITION EMPTY*/
        FOR 1    ;               /*THE TWO LOCATIONS OF THE FIFO */
        RDFIFO ;                 /*LOADED THROUGH THE HOST INTER-*/
        ENDFOR ;                 /*-FACE SECTION OF THE PAC1000  */
        MUL R2 R1 R0 ;
                DOR := R2 ;        /*REGISTER. THE PRODUCT IN THE */
LOOP1:  JMPNC DOR LOOP1 ;          /*DATA OUTPUT REGISTER         */
                DOR := R1 ;                 /*                   */
SELF:   JMP HOLD ;                          /*END OF PROGRAM       */
end:
```

In the following example, the contents of registers R2 and R1 is divided by the contents of register R0. The most significant word of the 32-bit long word is held in register R2 and the least significant 16 bits are stored in R1. The result of the divide operation leaves the quotient in the Q register and any remainder in register R2.

```
segment pacdes04 ;

HOLD:   JMPNC    CC0 HOLD ;      /*WAIT FOR START CONDITION EMPTY*/
        FOR 1    ;               /*THE TWO LOCATIONS OF THE FIFO */
        RDFIFO ;                 /*LOADED THROUGH THE HOST INTER-*/
        ENDFOR ;                 /*-FACE SECTION OF THE PAC1000  */
        DIV R2 R1 R0 ;
                DOR := Q   ;                /*OUTPUT THE REMAINDER*/
LOOP1:  JMPNC DOR LOOP1 ;
                DOR := R2 ;                 /*OUTPUT THE QUOTIENT. */
SELF:   JMP SELF ;                          /*END OF PROGRAM       */
end:
```

The files generated so far can be entered into the assembler and two files <filename>.LIS and <filename>.OB may be generated as shown in Figure 5. The latter object file must be linked before the final object file is available for programming into the PAC1000's EPROM. The link program <filename>.ML performs this function and is shown below.

```
load pacdes04 ;
place pacdes04 ;
end ;
```

This design example only used one program but many sub-modules may be linked together to form a single executable program. It is possible to simulate the design after linking. The necessary inputs to the simulator are the <filename>.OBJ, <filename>.STL and <filename>.CMD. The latter two files are the input stimulus file and the input command file (see Figure 5). The stimulus file is used to drive inputs such as address, data and condition codes. The command file lists which signals should be traced for observation. Examples of the stimulus file and command file are given below.

The command file shown below will instruct the simulator to set an output trace on the Current value of the Program Counter, CPC. The Condition Code zero input, the write, and the chip select lines are also traced. The simulator also enables a trace to be invoked on registers as well as input

Applications Programs (Cont.)

or output pins. The Q register is traced along with host data, loop counter, and registers R0, R1, and R2. The final trace is set on the host data output register. At the end of the stimulus file, the run instruction informs the simulator to run the driving signals for 140 cycles. The final instruction invokes a View Trace Waveform instruction, so the waveforms may be observed on the PC screen.

```
OPEN STIMULUS PACDES04
SET TRACE CPC
SET TRACE CC0
SET TRACE WRB
SET TRACE CSB
SET TRACE RDB
SET TRACE Q
SET TRACE HD
SET TRACE LC
SET TRACE R0
SET TRACE R1
SET TRACE R2
SET TRACE HDOR
OPEN TRACE PACDES04
RUN 140
V T W
```

The stimulus file is used to apply active signals to inputs of the design. At specific time points conditions are established. For example the statement:

.S CC0 0@1 1 @40

means that the input condition code zero CC0 should become a logic state LOW at time point one and a logic HIGH condition 40 cycles later. A three-state condition can be applied by typing the letter Z in place of logic '1' or '0.' The stimulus file is completed to drive all inputs and applied to the simulator during run time.

```
.S RESETB 0 @ 1 1 @ 2 ;
.S CC0 0@1 1@40 ;
.S WRB 1@1 0@2 1@8 0@12 1 @19 ;
.S CSB 1@1 0@2 1@9 0@11 1@18 0@120 1@129 0@131 1@139 ;
.S RDB 1@1 0@121 1@129 0@131 1@139 ;
.S HAD0 0@1 1@10 0@24 ;
.S HAD1 0@1 ;
.S HAD2 0@1 ;
.S HAD3 0@1 ;
.S HAD4 0@1 ;
.S HAD5 0@1 ;
# WRITE A 7 INTO R0 AND 31 INTO R1
.S HD0 0@1 1@10 Z@70 ;
.S HD1 1@1 Z@70 ;
.S HD2 0@1 1@10 Z@70 ;
.S HD3 0@1 1@10 Z@70 ;
.S HD4 0@1 1@10 Z@70 ;
.S HD5 0@1 Z@70 ;
.S HD6 0@1 Z@70 ;
.S HD7 0@1 Z@70 ;
.S HD8 0@1 Z@70 ;
.S HD9 0@1 Z@70 ;
.S HD10 0@1 Z@70 ;
.S HD11 0@1 Z@70 ;
.S HD12 0@1 Z@70 ;
.S HD13 0@1 Z@70 ;
.S HD14 0@1 Z@70 ;
.S HD15 0@1 Z@70 ;
```

The comment field is denoted by a '#' sign.

Case Statement Logic

The ability of the PAC1000 to perform case statement logic has already been discussed but the following program excerpt illustrates how to encode the case statement. The program will execute when condition code 7 is active high, then case group CG0 is tested for one of sixteen possible states. CG0 comprises CC0, CC1, CC2 and CC3. Sixteen registers are initialized and the output code is driven with zero. When CC7 goes HIGH the CG0 input is tested and the register contents that are equal to the state of the CG0 input is transferred to the AOR outputs.

```
segment pacdes05 ;

        /* illustrate the use of multiway branching */

        R0 := 0 ;
        R1 := 1 ;
        R2 := 2 ;
        R3 := 3 ;
        R4 := 4 ;
        R5 := 5 ;
        R6 := 6 ;
        R7 := 7 ;
        R8 := 8 ;
        R9 := 9 ;
        R10 := 10 ;
        R11 := 11 ;
        R12 := 12 ;
        R13 := 13 ;
        R14 := 14 ;
        R15 := 15 ;
        WHILE CC7 ;
        SWITCH CG0 ;
                CASE 0 , GOTO NEXT , AOR := R0  ;
                CASE 1 , GOTO NEXT , AOR := R1  ;
                CASE 2 , GOTO NEXT , AOR := R2  ;
                CASE 3 , GOTO NEXT , AOR := R3  ;
                CASE 4 , GOTO NEXT , AOR := R4  ;
                CASE 5 , GOTO NEXT , AOR := R5  ;
                CASE 6 , GOTO NEXT , AOR := R6  ;
                CASE 7 , GOTO NEXT , AOR := R7  ;
                CASE 8 , GOTO NEXT , AOR := R8  ;
                CASE 9 , GOTO NEXT , AOR := R9  ;
                CASE 10 , GOTO NEXT , AOR := R10 ;
                CASE 11 , GOTO NEXT , AOR := R11 ;
                CASE 12 , GOTO NEXT , AOR := R12 ;
                CASE 13 , GOTO NEXT , AOR := R13 ;
                CASE 14 , GOTO NEXT , AOR := R14  ;
                CASE 15 , GOTO NEXT , AOR := R15  ;
        ENDSWITCH ;
        NEXT : OUT 0 ;
        ENDWHILE ;
         OUT h'FFFF' ;
        LOOPX :
                GOTO LOOPX ;
        end ;
```

Simple DMA Controller for Memory to Memory Transfer

The software designs discussed so far have been based on arithmetic functions but an important feature of how to use the FIFO in the host interface section of the PAC1000 for the communication of data will enable the reader to develop ideas into more complex programs. The FIFO Output Ready flag is used in a loop to read the data into the registers. The output codes are used to create signals to control read, write, latch, output enable and bus acknowledge signals. A summary of these signals is given in Table 5 each time an instruction is executed. These signals are generated to accompany the memory addresses which control the DMA cycle.

Figure 6a shows a generic system solution where the PAC1000 sits on the address and data bus of a microprocessor and memory interface. The PAC1000 is mapped into the system with a PLD decoder and an external latch is used to catch data on read and write cycles. It is possible to use the internal DIR and DOR for this purpose but a faster solution would use an external component. Also, if the bus were greater than sixteen bits, an external latch would be required. This mode where data does not enter the PAC1000 device is called the 'fly by' mode.

Figure 6b shows the timing waveform derived from the program simulation. Active LOW WRB and CSB inputs to ADD1, ADD2 and ADD3 will write to the registers. The Source Address Register R0, the Destination Address Register R1 and the transfer counter R2 are all loaded through the FIFO. At time point 1, the registers become loaded. At time 2, CC7 is set HIGH to indicate transfer can commence. The response from the PAC1000 is an active LOW output from output control OC14 to inform the microprocessor that DMA activity is taking place. This occurs at time point 3. OC14 stays LOW during DMA activity but goes HIGH after the transfer is complete (at time point 4). Three transfers have taken place and the microprocessor is free to regain control of the bus.

```
segment pacdes06;

/*THE PROGRAM ILLUSTRATES A SIMPLE DMA DESIGN WHICH  */
/*READS THE DATA FROM SUCCESSIVE MEMORY LOCATIONS    */
/*ADDRESSED BY THE CURRENT CONTENTS OF R0 THEN WRITES*/
/*THAT DATA TO LOCATIONS ADDRESSED BY THE CONTENTS   */
/*OF R1. BOTH REGISTERS ARE INCREMENTED AFTER THE    */
/*READ/WRITE CYCLE. R2 IS A TRANSFER COUNTER THAT IS */
/*DECREMENTED AFTER EACH TRANSFER. WHEN R2 IS ZERO   */
/*ALL TRANSFER ACTIVITY CEASES AND A NEW DEVICE WAITS*/
/*FOR A NEW DMA CYCLE.                               */

/*              PIN FUNCTIONAL DESIGNATIONS.         */
/*                      OUTPUTS.                     */

/*      OC15 - LATCH ENABLE.....ACTIVE LOW.          */
/*      OC14 - BUS TAKEN.......ACTIVE LOW.           */
/*      OC13 - WRITE ENABLE.....ACTIVE LOW.          */
/*      OC12 - READ ENABLE.....ACTIVE LOW.           */
/*      OC11 - LATCH OUTPUT ENABLE....ACTIVE LOW.    */
/*      AOR  - 16 BIT ADDRESS OUTPUT..ACTIVE TRUE.   */

/*                      INPUTS.                      */

/*      CC7 - ACTIVE HIGH - INITIATE DMA ACTIVITY.   */
/*      HD  - ACTIVE TRUE - 16 DATA INPUTS.          */
/*      HAD - ACTIVE TRUE - REGISTER ADDRESS INPUTS  */
/*      /CS - ACTIVE LOW  - PAC1000 SELECT           */
/*      /WR - ACTIVE LOW  - WRITE TO PAC1000 FIFO    */
/*      /RD - ACTIVE LOW  - READ NOT USED            */

/*                  LIST OF EQUATES.                 */

READ    equ     H'AFFF'; /*ACTIVE LOW READ,TRANSFER  */
                         /*ENABLE,AND BUS BUSY       */
```

Figure 6a. PAC1000 as a Simple DMA Controller

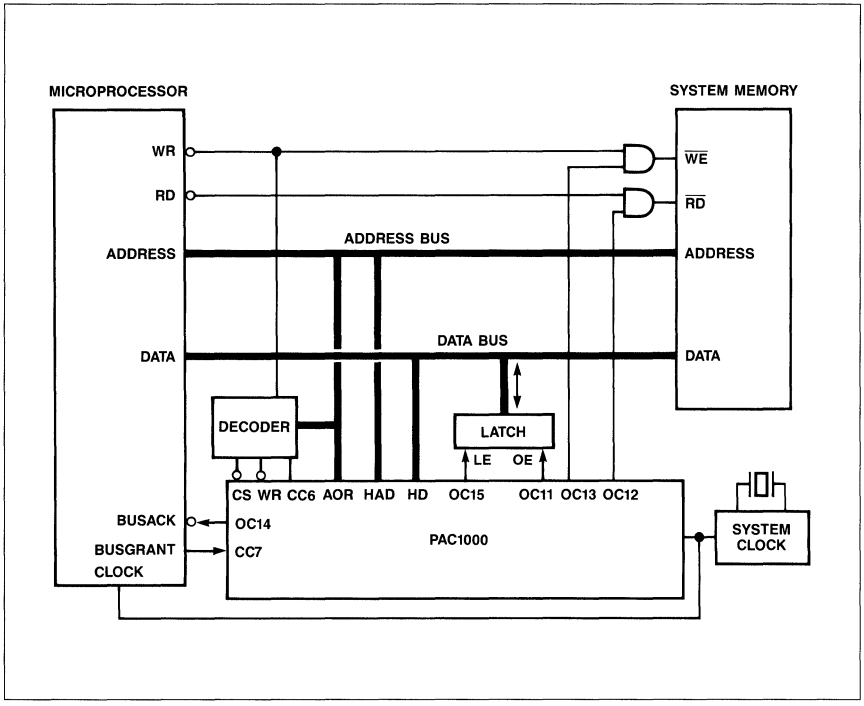


Figure 6b. System Waveforms

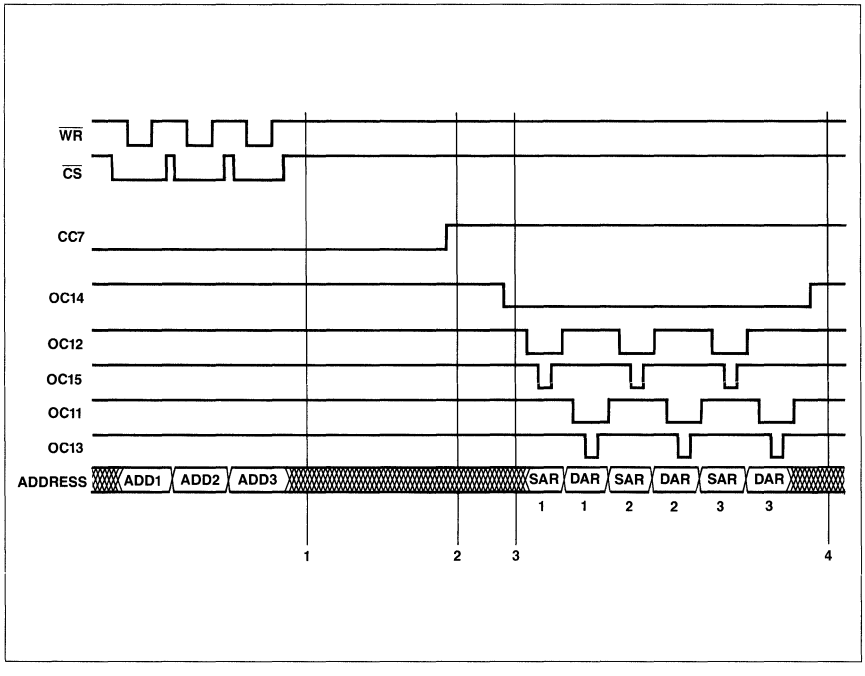


Table 5. Output Condition Assignment Codes for the DMA Controller Application

	OC15	OC14	OC13	OC12	OC11	OC10-OC0
INIT	1	1	1	1	1	All High
READ	1	0	1	0	1	All High
OENBL	1	0	1	1	0	All High
WRITE	1	0	1	0	0	All High
ENBLE	1	0	1	1	1	All High
LATCH	0	0	1	0	1	All High

OC15 = Active Low Latch Command
OC14 = Active Low DMA in Progress
OC13 = Active Low Write Signal
OC12 = Active Low Read Signal
OC11 = Active Low Output Enable

```
LATCH   equ     H'2FFF';      /*ACTIVE LOW READ,TRANSFER          */
                              /*ENABLE,LATCH ENABLE,AND           */
                              /* BUS BUSY                         */
OENBL   equ     H'B7FF';      /*ACTIVE LOW TRANSFER ENABLE        */
                              /*OUTPUT ENABLE,AND BUS             */
                              /* BUSY                            */
WRITE   equ     H'97FF';      /*ACTIVE LOW WRITE,TRANSFER         */
                              /*OUTPUT ENABLE,AND BUS             */
                              /* BUSY                            */
INIT    equ     H'FFFF';      /*INITALIZE ALL OUTPUTS HIGH        */
ENBLE   equ     H'BFFF';      /*ACTIVE LOW ENABLE TRANSFER        */
                              /*SIGNAL,AND BUS BUSY               */

/*                  PROGRAM START                                 */

START:  OUT INIT;             /*INITALIZE OUTPUT CODES TO CC0-15*/
LOOP1:  RESET ADOE ;          /*SET THE ADDRESS BUFFERS INPUTS  */
        FOR 2 ;               /*SET READ FIFO LOOP TO 3         */
HOLD0:  JMPNC FIOR HOLD0 ;    /*WAIT FOR ACTIVE FIOR FLAG       */
        RDFIFO ;              /*READ FIFO INTO THE REGISTER FILE*/
        ENDFOR ;              /*ALL THREE WORDS READ END LOOP   */

HOLD1:  JMPNC CC7 HOLD1 ;     /*ACTIVE CC7 BUSACK SIGNAL INPUT  */
        SET ADOE ;            /*SET ADDRESS BUFFER AS OUTPUT    */
                              /*FOR DMA CYCLES                  */

        FOR , R2 := R2 , OUT ENBLE ;/*START DATA TRANSFERS     */
        AOR := R0 ;                  /*OUTPUT SOURCE ADDRESS   */
        R0 := ++ R0 , OUT READ ;     /*OUTPUT ACTIVE READ      */
        OUT LATCH ;                /*AND LATCH DATA ON READ    */
        OUT READ ;                 /*HOLD READ LINE ACTIVE     */
        AOR := R1;                 /*OUTPUT DESTINATION ADDRESS*/
        R1 := ++ R1 , OUT OENBL ;  /*ENABLE LATCH OUTPUT       */
        OUT WRITE ;                /*PERFORM WRITE CYCLE       */
        OUT OENBL ;                /*DISABLE WRITE BEFORE OE   */
        OUT ENBLE ;                /*END OF SINGLE TRANSFER    */
        ENDFOR ;                   /*END OF TRANSFER CYCLE     */

HALT:   GOTO LOOP1 , OUT INIT ;    /*RETURN TO PROGRAM START   */
end;
```

FIFO DRAM Controller

The next PAC1000 design example illustrates how to use the device as a FIFO DRAM Controller. See Figure 7a for device implementation.

If the DRAMs are 64K devices, only the least significant byte of the AOR register need be used (that is ADD0–ADD7). The system could easily be upgraded to handle 256K or 1M bit DRAMs by wiring in address bits A8 and A9 but additional PAC1000 software would need to be written to accommodate the FIFO status counter. About 45 lines of code are used to enable the PAC1000 to handle REFRESH, READ and WRITE activity. The design uses the output control lines to provide RAS, CAS and WRITE signals to the DRAM and additional signals to give busy status during read, write and refresh activity. The whole system responds to input condition codes CC0 and CC1 as RQWRITE request to write and RQREAD request to read respectively. During active read, write and refresh cycles, three signals BUSYWR, BUSYRD and BUSYRFSH which go active LOW an additional composite signal which goes LOW when the FIFO is in any of these conditions. The system design also incorporates an UP/DOWN status counter which increments on write activity and decrements on read activity. This counter is tested to provide information to the outside world that the FIFO is full, empty or neither full or empty. The FULL, EMPTY and ACTIVE flags can be read from the IO0 and IO1 and give information to the outside world about the status of the FIFO.

The waveforms associated with read, write and refresh activity are shown in Figures 7b, 7c and 7d respectively. These waveforms were created from the PACDES08.OUT vector tables generated from the simulator. Table 6 illustrates the assignment of the output conditions which drive the various functions RAS, CAS, RFSH WR etc.,

It is recommended that high current buffer circuits be used to interface the outputs of the PAC1000 to the inputs of the memory chips used in both the DMA and FIFO applications.

```
 segment pacdes08 :

/*LIST OF EQUATES.*/
/*CONDITTION CODE OUTPUTS*/

RASW     equ      H'55FF' ;        /*WRITE RAS OUTPUT   */
RASR     equ      H'79FF' ;        /*READ RAS OUTPUT    */
RFSH     equ      H'7CFF' ;        /*REFRESH OUTPUT     */
CASW     equ      H'15FF' ;        /*WRITE CAS OUTPUT   */
CASR     equ      H'39FF' ;        /*READ CAS OUTPUT    */
ENDWR    equ      H'35FF' ;        /*END OF WRITE OUPUT*/
INIT     equ      H'FFFF' ;

ZERO     equ      H'0000' ;        /*ZERO COUNT*/
FULL     equ      H'FD'   ;        /*FULL FLAG */
EMPTY    equ      H'FE'   ;        /*EMPTY FLAG*/
ACTVE    equ      H'FF'   ;        /*ACTIVE     */
MAX      equ      H'FFFF' ;        /*MAX COUNT */

RQWRITE  equ      CC0 ;            /*REQUEST TO WRITE*/
RQREAD   equ      CC1 ;            /*REQUEST TO READ */

                              /*PROGRAM START*/

START:   OUT INIT ;.                /*INITALIZE OUTPUT CODES*/
                                    /*INITALIZE REGISTERS   */
R0 := H'0000'                  ;    /*ROW ADDRESS WRITE     */
R1 := H'0000'                  ;    /*COLUMN ADDRESS WRITE  */
R2 := H'0000'                  ;    /*ROW ADDRESS READ      */
R3 := H'0000'                  ;    /*COLUMN ADDRESS READ   */
```

FIFO DRAM Controller (Cont.)

```
                R4 := H'FFFF'          ;    /*REFRESH COUNTER      */
                R5 := H'0000'          ;    /*STATUS COUNTER       */

                OUTPUT IO0 IO1 ;            /*SER IO0 AND IO1 TO   */
                SET ADOE , OUT INIT ;       /*OUTPUT, ADOE INPUT   */
                IOR := EMPTY  ;             /*FIFO IS EMPTY        */
                GOTO TEST ;                 /*TEST REQUEST TO      */
                                            /*READ/WRITE           */

LOOP:           AOR := R4 ;                 /*OUTPUT REFRESH CTR   */
                OUT RFSH ;                  /*PERFORM REFRESH      */
                R4 := ++ R4 , OUT INIT ;    /*INCREMENT RFSH CTR   */
                                            /*CLEAR OUTPUT         */

TEST:           IF RQWRITE;                 /*IF REQUEST TO WRITE  */
                 AOR := R0 , OUT INIT ;     /*OUTPUT WRITE ADDR    */
                 R5 := ++ R5 ;              /*INCREMENT STATUS     */
                 OUT RASW ;                 /*OUTPUT RAS WRITE     */
                 AOR := R1 ;                /*OUTPUT CAS ADDR      */
                 R1 := ++ R1 ;              /*INCREMENT CAS ADDR   */
                 OUT CASW ;                 /*OUTPUT CAS ADDR      */
                 OUT ENDWR ;                /*END WRITE CYCLE      */
                 OUT INIT ;                 /*FINISH WRITE CYCLE   */
                ENDIF ;

                 IF R1 == 256 ;             /*TEST FOR 256 COLUMNS*/
                  R0 := ++ R0 ;             /*INCREMENT ROW        */
                 ENDIF ;                    /*IF 256               */

                 IF RQREAD;                 /*IF REQUEST TO READ   */
                 AOR := R2 , OUT INIT ;     /*OUT ROW READ ADDRESS*/
                 R5 := -- R5 ;              /*DECREMENT STATUS     */
                 OUT RASR ;                 /*OUTPUT RAS READ      */
                 AOR := R3 ;                /*OUTPUT CAS ADDRESS   */
                 R3 := ++ R3 , OUT CASR ;   /*INCREMENT CAS ADD    */
                 OUT CASR ;                 /*STRETCH CAS          */
                 OUT INIT ;                 /*FINISH READ CYCLE    */
                ENDIF ;

                IF R3 == 256 ;              /*TEST FOR 256 COLUMNS*/
                 R2 := ++ R2 ;              /*INCREMENT ROW        */
                ENDIF ;                     /*IF EQUAL TO 256      */

                R6 := R5 ;                  /*SAVE STATUS COUNTER  */
                R6 := MAX - R5 ;            /*TEST FOR MAX COUNT   */
                IF Z ;                      /*IF MAXIMUM           */
                IOR := FULL ;               /*SET OUTPUT FULL FLAG*/
                GOTO LOOP ;                 /*GOTO REFRESH LOOP    */
                ENDIF ;                     /*END TEST             */

                R6 := R5 ;                  /*SAVE STATUS COUNTER  */
                R6 := ZERO - R6 ;           /*TEST FOR ZERO COUNT  */
                IF Z ;                      /*IF ZERO              */
                IOR := EMPTY ;              /*SET EMPRY FLAG       */
                GOTO START;                 /*RESTART PROGRAM      */
                ENDIF ;                     /*ELSE                 */

                IOR := ACTVE ;              /*THE SYSTEM IS NOT    */
                GOTO LOOP ;                 /*FULL OR EMPTY        */
end;
```

Table 6. Output Condition Assignment Codes for the PAC FIFO DRAM Controller Design

	OC15	OC14	OC13	OC11	OC10	OC9	OC8	OC12, OC7–OC0
INIT	1	1	1	1	1	1	1	All High
RASW	0	1	0	1	0	1	0	All High
CASW	0	0	0	0	1	0	1	All High
ENDW	0	0	1	0	1	0	1	All High
RASR	0	1	1	1	0	0	1	All High
CASR	0	0	1	1	0	0	1	All High
RFSR	0	1	1	1	1	0	0	All High

OC15 = Active Low RAS
OC14 = Active Low CAS
OC13 = Active Low Write
OC11 = Active Low BUSYWR
OC10 = Active Low BUSYRD
OC9 = Active Low Busy
OC8 = Active Low BUSYRFSH

Figure 7a. Using a PAC as a FIFO DRAM Controller

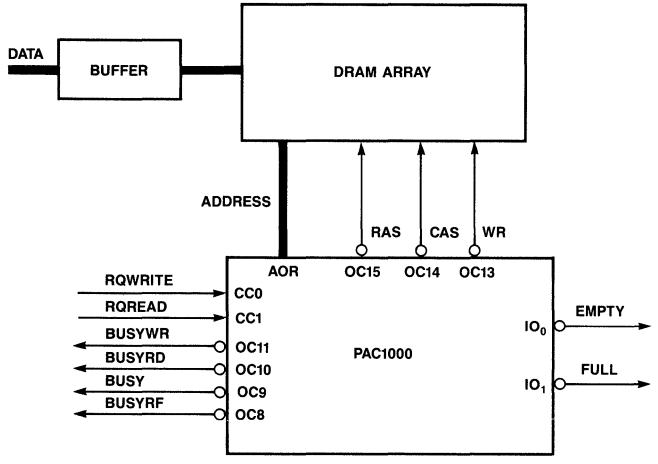
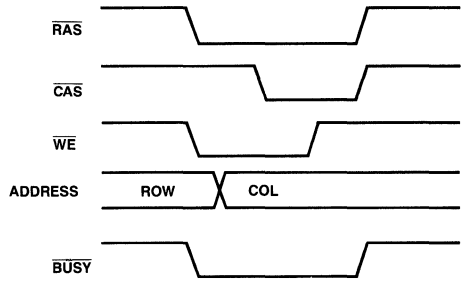
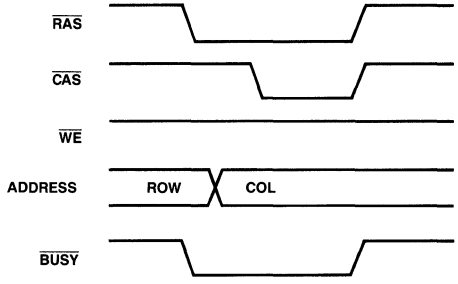


Figure 7b.



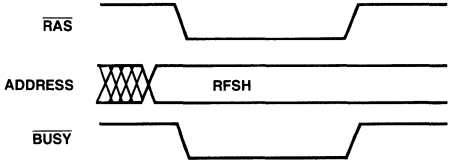
In response to a request to read one early write cycle will take place. $\overline{RAS}$ will latch in the row address and the WE line goes low. The column address is set up followed by the falling edge of $\overline{CAS}$. The WE input is taken inactive followed by $\overline{RAS}$ and $\overline{CAS}$. During the whole cycle the busy signal is active.

Figure 7c.



In response to a request to read one read cycle will take place. The $\overline{RAS}$ and $\overline{CAS}$ signals latch in the row and column addresses respectively but the WE input is inactive throughout the cycle. The $\overline{BUSY}$ signal is active throughout the whole cycle.

Figure 7d.



To refresh the memory the PAC will output a refresh count to be strobed into the DRAMs by an active low $\overline{RAS}$ transition.

Programmable UART

The PAC1000 contains no UART for serial data but parallel to serial conversion is possible through the Q register and I/O Port 2 and 3. The following program illustrates the designer how to create a UART function in the PAC1000 with about 40 lines of instructions. The PAC1000 device will receive data in parallel from the host system. The FIFO is used to interface to the host and transfer data into the registers. The program will take the seven bits of ASCII code and calculate the parity, then add a parity bit. The result is serialized and framing bits are applied. The data, one parity bit, one start bit and two stop bits are serially clocked out of the Q register into Port 3. The handshake signals of Data Terminal Ready and Data Set Ready are built into the program.

```
segment pacdes09 ;

/*THIS PROGRAM ILLUSTRATES THE PARALLEL TO SERIAL          */
/*CHANNEL CONVERSION OF THE PAC1000 TO THE PERIPHERAL      */
/*BUS OF THE SYSTEM                                        */
/*                                                        */
/*              PIN FUNCTIONAL DESIGNATIONS.              */
/*                      OUTPUTS.                          */
/*                                                        */
/*      OC12 - DTR - DATA TERMINAL READY....ACTIVE LOW.   */
/*      OC13 - RHD - RECEIVED HOST DATA.....ACTIVE LOW.   */
/*      OC14 - DONE.........................ACTIVE LOW.   */
/*      OC15 - ABORT........................ACTIVE LOW.   */
/*      IO3 - TxD - TRANSMITTED DATA........ACTIVE LOW.   */
/*                                                        */
/*                      INPUTS.                           */
/*                                                        */
/*      CC0 - DSR - DATA SET READY.........ACTIVE HIGH.   */
/*      CC1 - START TRANSMITTING...........ACTIVE HIGH.   */
/*      HD  - ACTIVE TRUE - 16 DATA INPUTS.               */
/*      HAD - ACTIVE TRUE - REGISTER ADDRESS INPUTS       */
/*      /CS - ACTIVE LOW  - PAC1000 SELECT                */
/*      /WR - ACTIVE LOW  - WRITE TO PAC1000 FIFO         */
/*                                                        */

INIT    equ  H'FFFF'; /*INITALIZE ALL OUTPUTS HIGH       */
RHD     equ  H'DFFF'; /*ACKNOWLEDGE RECEIVING HOST DATA  */
DTR     equ  H'EFFF'; /*DATA TERMINAL READY              */
DONE    equ  H'BFFF';
ABORT   equ  H'8FFF'; /*TELL HOST THAT DATA WAS CORRUPTED*/


/* R21 - H'0060' - MASK REGISTER FOR EVEN PARITY          */
/* R20 - H'00E0' - MASK REGISTER FOR ODD PARITY           */
/* R19 - H'0002' - CONSTANT TO DIVIDE THE 32-BIT VALUE    */
/*                 IN RX R16                              */
/* R18 - H'0000' - COUNTER OF THE NUMBER OF ONES IN THE   */
/*                 DATA                                   */
/* R17 - H'FFFF' - A CONSTANT TO MASK WITH DATA           */
/* R16 - H'0000' - A CONSTANT TO MASK WITH DATA           */
/* R8  -           WORKING REGISTER FROM RO               */
/* RO  -           ORIGINAL DATA FROM HOST SYSTEM         */
/* Q   -           REGISTER TO SHIFT OUT DATA TO THE      */
/*                 SERIAL PORT                            */
```

Programmable UART (Cont.)

```
begn:   R21 := H'0060' , OUT INIT ;  /*SET OC[15:0] HIGH*/
              R20 := H'00E0' ;
              R19 := H'0002' ;
              R18 := H'0000' ;
              R17 := H'FFFF' ;
              R16 := R18 ;
              Q := R18 ; /*  INITIALIZE Q TO ZERO'S   */


/*            WAIT FOR HOST TELLS PAC1000              */
/*            TO START TRANSMITTING DATA               */

stndby: JMPNC CC1 stndby ;
        JMPC FICD abort ;
        RDFIFO , OUT RHD ; /* READ FIFO DATA INTO RO    */
                           /*TELL HOST THAT DATA WAS    */
                           /*READ CORRECTLY             */

/*********************************************************/
/*            FORMAT OF DATA RECEIVED                    */
/*      FIFODA[15:0]                                     */
/*      15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0   */
/*      0  0  0  0  0  0  0  0  0 D1 D2 D3 D4 D5 D6 D7   */
/*********************************************************/

/*      SWAP THE HIGHER AND LOWER BYTES                  */

                                /* SET OC TO NON-        */
        AOR := RO , OUT INIT ; /* FUNCTIONING MODE       */
        R8 := SWPV ;           /* MODE SWAP TO SHIFT     */
        RO := SWPV ;           /* LATER SWAP NOW         */

/*      SHIFT DATA                                       */

            FOR 7 ;
              R8:= R8 << 0 ;
              IF S ;
                R18 := ++ R18 ;  /*INCREMENT COUNTER*/
              ENDIF ;
            ENDFOR ;

/*       CHECK FOR EVEN/ODD PARITY                       */

        DIV R16 R18 R19 ; /* DIVIDE R18 R16 BY 2        */
        OR Q 0 ;          /* CHECK IF REMAINDER IS ZERO */
        IF Z ;            /* IF Z=1 THEN JUMP TO PARITY */
                          /* (EVEN PARITY)              */
                          /* IF Z = 0 THEN (ODD PARITY) */
        OR RO R21 ;  /* MERGE MASK BITS FOR EVEN PARITY */
        ELSE ;
        OR RO R20 ; /*  MERGE MASK BITS FOR ODD PARITY  */
        ENDIF ;     /* RO IS NOW FORMATTED CORRECTLY FOR*/
                    /* SERIAL SHIFTING                 */
        Q := RO ;   /* LOAD RO TO Q TO SHIFT OUT TO IO3 */

/*      CHECK THAT RECEIVING END IS READY                */
```

Programmable UART (Cont.)

```
wait:    IF CC0 ;    /*IF RECEIVER READY SET IO3 TO OUTPUT */
         CONFIGURE SIO ; /*AND SET MODE TO SHIFT Q TO IO3 */
         OUTPUT IO3 , OUT DTR ; /*DRIVE DTR TO ZERO THIS  */
                                /*TELLS THE RECEIVER THAT */
                                /*THE TRANSMITTER IS READY*/


/* SHIFT OUT THE 1 START BIT,7-BITS OF DATA,1 PARITY AND */
/* 2 STOP BITS . THEREFORE SHIFT 11 TIMES */

         LDLC 10 ;          /*LOAD 10 INTO LOOP COUNTER FOR */
                            /*A SHIFT OF 11 THEN FILL WITH  */
                            /*ZEROS                         */
lp:      LOOPNZ lp , QR8 := Q << 0 R8 << 0 ;
         ELSE;
         JMP wait ;  /*IF RECEIVER IS NOT READY THEN WAIT*/
         ENDIF ;
         OUT DONE ;  /*TELL HOST THAT PAC1000 IS DONE     */
         JMP begn ;  /* START AGAIN FOR NEXT DATA         */

/*       ABORT DATA READ AND TELL HOST ABOUT IT           */

abort:   JMP begn , OUT ABORT ;

end;
```

Summary

The PAC1000 programmable peripheral controller incorporates many features that enable a high speed design to be quickly realized. Its reprogrammability has enabled many designers to go to printed circuit board layout early in the design cycle. Moreover, because the system logic is programmable into the on-chip EPROM, modifications can be made at a later time without having to change printed circuit board artwork. In fact over discrete and PAL/EPLD type solutions the printed circuit board artwork is considerably less complex because a greater degree of circuit complexity containing much interconnect has migrated into the instructions encoded in the EPROM section of the chip.

To learn how to use the PAC1000 is a relatively quick process for most systems designers have designed with microprocessors and microcontrollers. This is because they understand the writing of assembly or high level code. With the support of WSI's user friendly software tools, an engineer can be designing with the PAC1000 in less than a week. This contrasts with the many and diverse schematic capture, net translation, placing and routing, annotation and back-annotation packages that support EPLD and PGA devices. These products subject the designer to a multiplicity of software tools that he must become familiar with. This results in generating a long learning curve that can easily be avoided with the PAC1000 and WSI's software support.

The result of using the PAC1000 device and software tools virtually guarantees the fastest route possible from initial conception to the final design of a complex high performance system.

Programmable Peripheral
Application Note 012
Testing 8 Dual-Port RAM Memories with the PAC1000 Programmable Peripheral Controller

By Karen Spesard

Abstract

The PAC1000 16-Bit Programmable Peripheral Controller is a member of WSI's Programmable Peripheral family. It can be used in a variety of different applications requiring high-performance as well as high integration because of it's control architecture, user-configurability, and flexibility. This application note describes the use of the PAC1000 in the stand-alone mode as a Memory Tester for eight dual-port static RAMs with interrupts.

Each of the eight dual-port RAMs, with 2K x 8 bits shared memory, is accessed from both ports and tested by the PAC1000 for all possible functional failures. The PAC1000 simple interface to the dual-port memory is discussed, as well as the dual-port memory test conditions and timing considerations. Finally, examples of program code are given to illustrate how easy the PAC100 is to use.

Introduction

Dual-port static RAMs are typically used to simplify communication between processors in computer systems. They have become popular in recent years due to the fact that they allow simultaneous read and write accesses to the same memory providing the capability for two devices to communicate with each other without the need for any special data communication hardware between the devices. These devices could, for instance, be an I/O controller and a CPU or two CPUs working on separate but related tasks. This contrasts with the DMA (dual memory access) approach where a single memory is shared between CPUs and/or one or more I/O devices and where hardware arbitration logic is always necessary.

Testing multiple dual-port static RAM memories efficiently, however, has often proved difficult because two sets of devices can control each memory independently and access any word in memory simultaneously. This includes the case where both devices are accessing the same memory location at the same time. Arbitration is required to insure against this case which is usually handled in the memory hardware. For example, most dual-port RAMs have address detection logic and a cross-coupled arbitration latch to provide a busy signal for the address that arrived last, so writing to the busy port is deterred. As a result, testing multiple memories for these cases requires processors or controllers capable of providing the necessary control signals, memory addresses, and data in real time.

The PAC1000 Programmable Peripheral Controller is well suited for this type of application. It provides a single-chip user-configurable test interface for up to 8 dual-port memories at one time, eliminating the need for discrete implementations of PLDs, ALUs, SRAMs, and Register files. It has 64K EPROM program store on-chip (1K x 64 bits) as well as a microsequencer, a 16-bit ALU and register set, and programmable I/O ports. The PAC1000 also has the capability of controlling very fast systems, generating addresses to memory, feeding the system data, and responding to interrupts, all at one time. In fact, its architecture allows it to be able to execute three parallel operations (Control, Output, and CPU) every clock cycle, making timing predictable and increasing throughput significantly. See Figure 1 for the PAC1000 single cycle control architecture and Figure 2 for a simplified block diagram of the PAC1000.

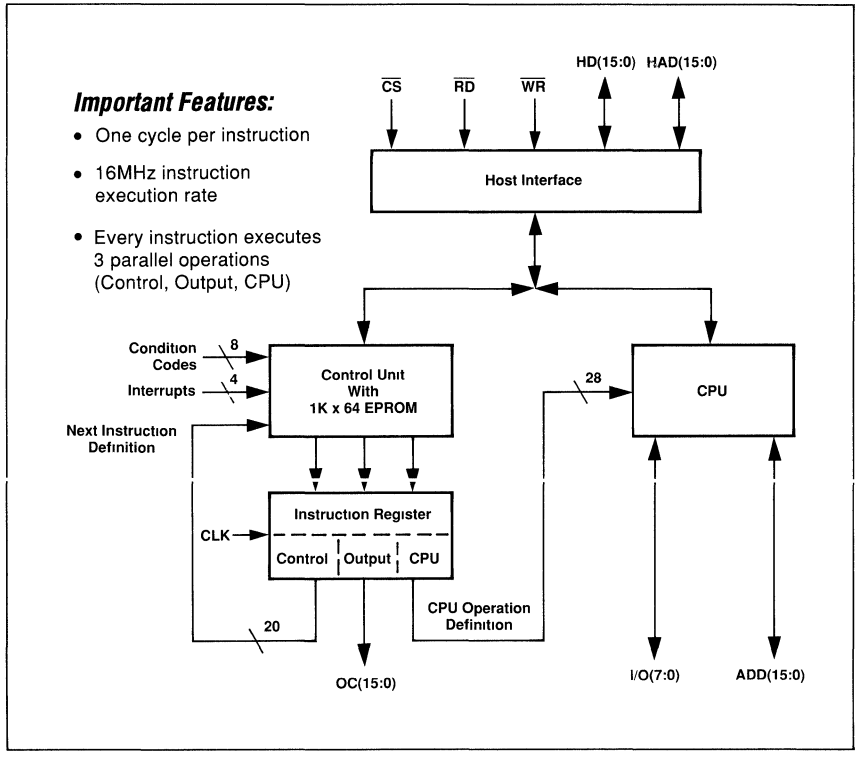
A typical instruction containing three parallel operations illustrates the efficiency of the PAC1000 in this application. For example, during a dual-port RAM access, the sequencer section of the PAC1000 can check for the $\overline{BUSY}$ signal or the end of a loop, the output control section can generate the $\overline{CS}$ signals for each of the RAMs, and the CPU can

Introduction (Cont.)

generate the $\overline{RD}$, $\overline{WR}$, and $\overline{OE}$ strobes or calculate and produce the next address all during the same instruction cycle simultaneously.

In addition, of course, the PAC1000 can also be used to control other system functions while testing the dual port memories. In most cases, the PAC1000 can perform intelligent DMA control and I/O control protocols at the same time. And, when switched to the peripheral mode, it can off-load other tasks, as well, from a host processor.

Figure 1. PAC1000 Single Cycle Control Architecture



PAC1000 to Dual-Port SRAM Interface

The circuit diagram of a typical system configuration using the PAC1000 as a memory tester is shown in Figure 3 for eight 2K x 8 dual-port static RAMs for a total memory depth of 16K x 8 bits. Each dual-port RAM has two complete and independent sets of address, data, and read/write control signals and shares the same set of memory cells. The PAC1000 memory tester interfaces directly with each dual-port static RAM without the need for any external glue logic.

Specifically, the 16-bit PAC1000 interfaces to the dual-port static RAMs as two I/O or CPU devices would interface, with one exception: the PAC1000 has two 16-bit and one 6-bit user-configurable address/data buses, whereas two CPU or I/O devices would have two distinct address buses and two distinct data buses. Therefore, the PAC1000 buses need to be split. This is handled by combining the 16-bit address bus and 6-bit host address bus of the PAC1000 and configuring them as a 22-bit address bus. This 22-bit bus, in turn, is split into two 11-bit buses for accessing both left and right ports of 2K x 8 memory simultaneously. For each memory, ADD(15:5) corresponds to $A(10:0)_L$, and ADD(4:0)/HAD(5:0) corresponds to $A(10:0)_R$. (See Figure 4.) Likewise, the 16-bit data bus is split between both ports of memory. Thus, HD(15:8) corresponds to $I/O(7:0)_L$ and HD(7:0) corresponds to $I/O(7:0)_R$.

PAC1000 Dual-Port SRAM Interface (Cont.)

To select one of the dual-port memory ports, 16 output control lines, OC(15:0), are individually connected to the chip enables of the dual-port RAMs – one OC line per port. The left ports of memory share OC(15:8) lines and the right ports of memory share OC(7:0) lines. The eight read/write and output enable control pins of the dual-port memories also interface directly to the PAC1000 through it's input/output port, I/O(7:0). Here, the I/O7 and I/O6 pins are tied to each of the eight $R/\overline{W}_L$ signals and $R/\overline{W}_R$ signals, respectively. Also, the I/O5 and I/O4 pins are tied to each of the eight dual-port memory $\overline{OE}_L$ and $\overline{OE}_R$ signals, respectively.

The remaining pins to be discussed are the right and left $\overline{BUSY}$ and $\overline{INTR}$ signals of the dual-port memories. Each group of four memories have their right or left $\overline{BUSY}$ or $\overline{INTR}$ pins tied together and connected to the PAC1000 condition code inputs. CC7 and CC6 correspond to the $\overline{BUSY}$ left and right signals of dual port RAMs #1–4, and CC5 and CC4 correspond to the $\overline{BUSY}$ left and right signals of dual port RAMs #5–8. Likewise, CC3 and CC2 interface to the $\overline{INT}$ left and right signals of dual port RAMs #1–4 and CC1 and CC0 to the $\overline{INT}$ left and right signals of dual port RAMs #5–8.

Figure 2. PAC1000 Block Diagram

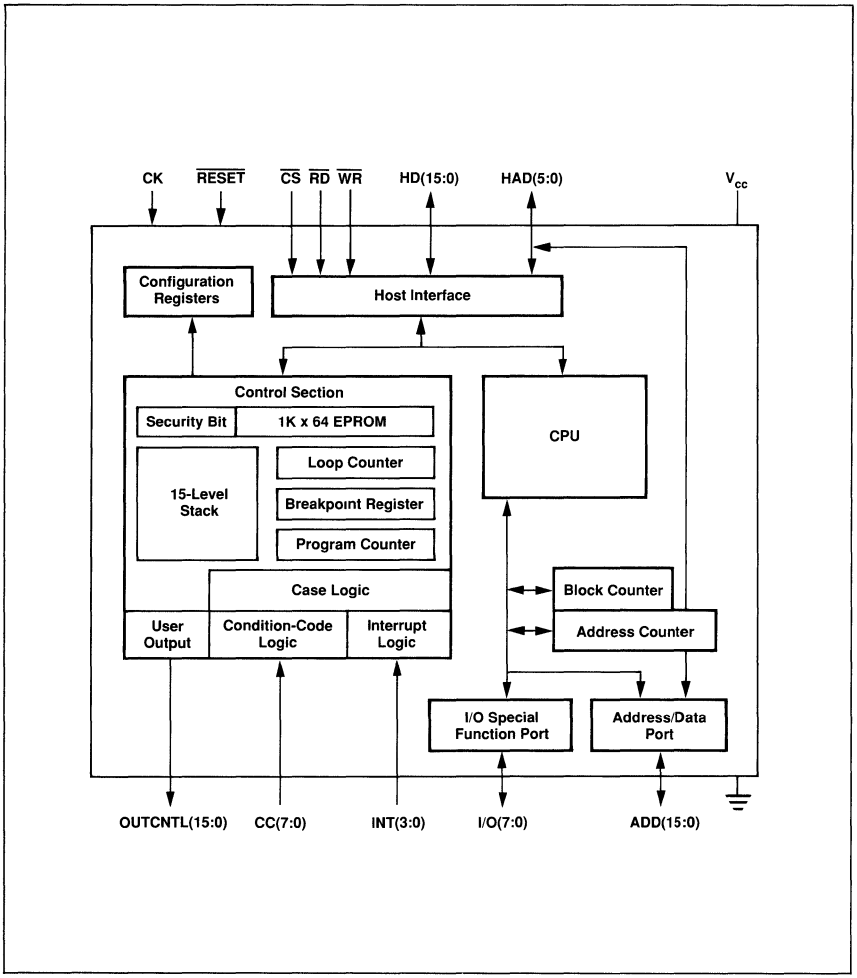


Figure 3. PAC1000 Configured as a Dual-Port RAM Memory Tester

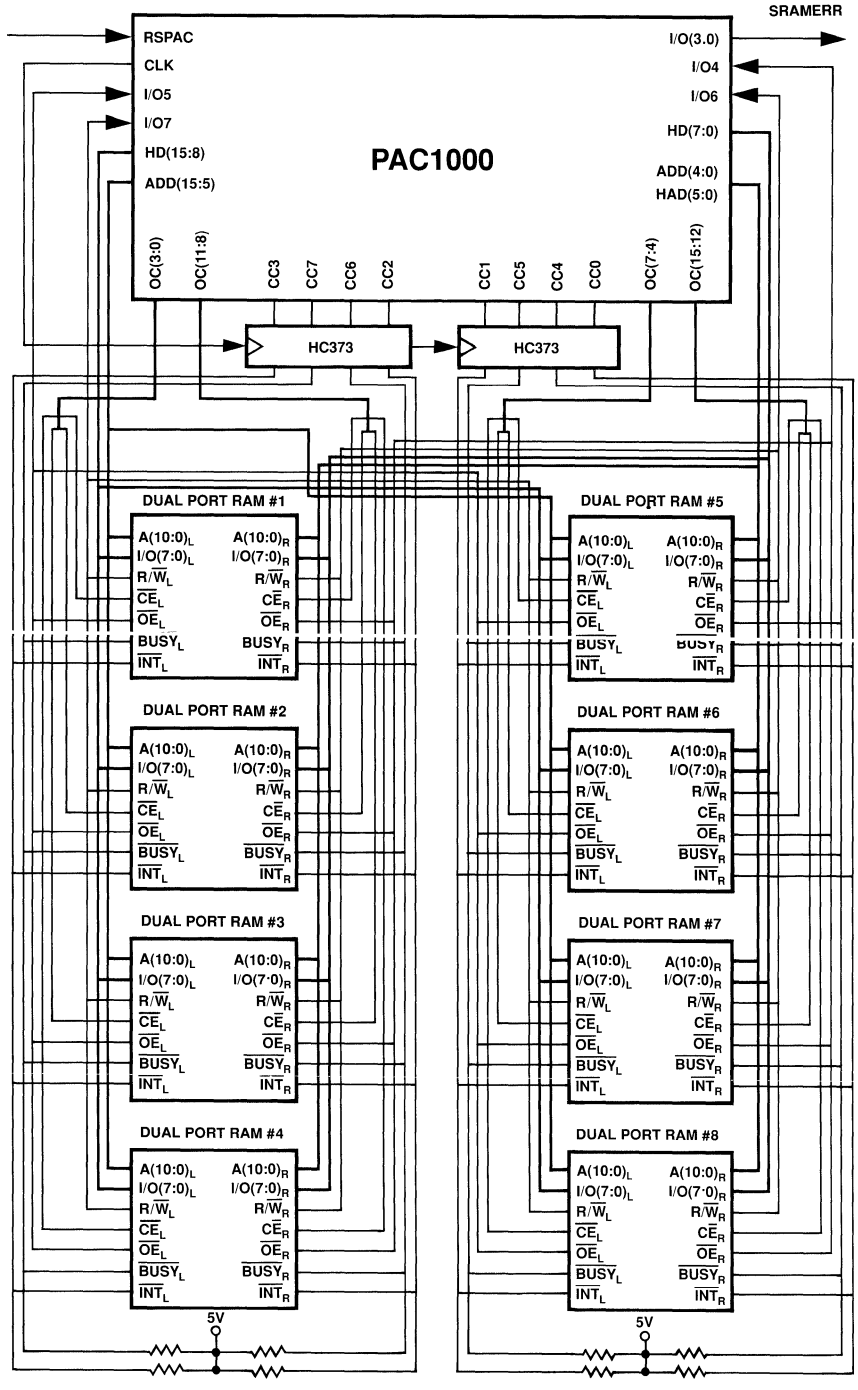


Figure 4. Address Splitting for Dual-Port RAM Memory Testing

PAC1000 Address/Data Bus	ADD15	ADD14	ADD13	ADD12	ADD11	ADD10	ADD9	ADD8	ADD7	ADD6	ADD5
Address Counter High	ACH15	ACH14	ACH13	ACH12	ACH11	ACH10	ACH9	ACH8	ACH7	ACH6	ACH5
Left Port of Dual Port RAM	$A10_L$	$A9_L$	$A8_L$	$A7_L$	$A6_L$	$A5_L$	$A4_L$	$A3_L$	$A2_L$	$A1_L$	$A0_L$
PAC1000 Address/Data Bus Host Address Bus	ADD4	ADD3	ADD2	ADD1	ADD0	HAD5	HAD4	HAD3	HAD2	HAD1	HAD0
Address Counter High/Low	ACH4	ACH3	ACH2	ACH1	ACH0	ACL5	ACL4	ACL3	ACL2	ACL1	ACL0
Right Port of Dual Port RAM	$A10_R$	$A9_R$	$A8_R$	$A7_R$	$A6_R$	$A5_R$	$A4_R$	$A3_R$	$A2_R$	$A1_R$	$A0_R$

NOTES: Address buses can be written from a 16-bit or 22-bit Address Counter (16-bit ACH or 22-bit ACH/ACL) or from a 16-bit Address Output Register In this application, the address bus is driven by the 22-bit Address Counter

Functional Description

The two basic operational modes for the PAC1000 are either as a stand-alone controller or as a memory-mapped peripheral to a host processor. The PAC1000, as a dual-port static RAM memory tester, is configured in the standalone mode. In this mode, the PAC1000 has complete control over the bus at all times; moving data from the 16-bit data bus back and forth to/from the left and right sides of the memory, generating addresses from the 22-bit address counter through the 16-bit address/data and 6-bit host address buses to the 2K x 8 memory, generating the control signals to the memory, and monitoring/responding to the memory's status and control signals.

The PAC1000 dual-port memory tester performs basically like two separate I/O or CPU devices, writing and reading from each port of the dual-port static RAM through the same memory. In this application, the PAC1000 tests dual-port memory for all possible fault conditions. For example, one of the tests the PAC1000 performs is for the case when both ports of the dual-port memory attempt to access the same memory cell location (writing or reading) at basically the same time. If this happens, one of the ports of memory, through hardware arbitration, is inhibited from being accessed and is supposed to receive a $\overline{BUSY}_{L/R}$ signal. To test for this condition, the signal needs to reach the device and the device needs to respond in real time.

The PAC1000 can respond to the $\overline{BUSY}$ lines generated by either the left or right sides of dual-port memory, in one cycle through its condition code inputs. These condition code inputs, CC7–CC4, are used because each condition can be tested for true or not true simultaneously by the PAC1000. And since the condition code logic is part of the sequencer, a decision can be made within the next cycle on how to respond.

The $\overline{INTR}$ signals generated by the dual-port memory provide another case for testing. When the left side port writes into the top odd address (7FF) or the right side port writes into the top even address (7FE) of the memory chip, the interrupt latch is set and the interrupt line to the opposite side port is supposed to be activated. These top two addresses serve as flag bits or interrupt generators and the activated interrupt signal gives permission for the interrupting CPU to use the memory. An interrupt latch is cleared when the opposite side port reads from the same address (e.g. to clear the right port after the left port writes into 7FF, the right side port reads from address 7FF). The interrupts are designed to save system designers from having to design in extra logic, and to allow one CPU to interrupt the other.

To test for the functionality of the $\overline{INTR}$ signals generated by the dual-port memory, these signals, like the $\overline{BUSY}$ signals, are also tied to the PAC1000's condition code inputs. Again, the PAC1000's condition code inputs allow it to respond to the dual-port memory interrupts in real time.

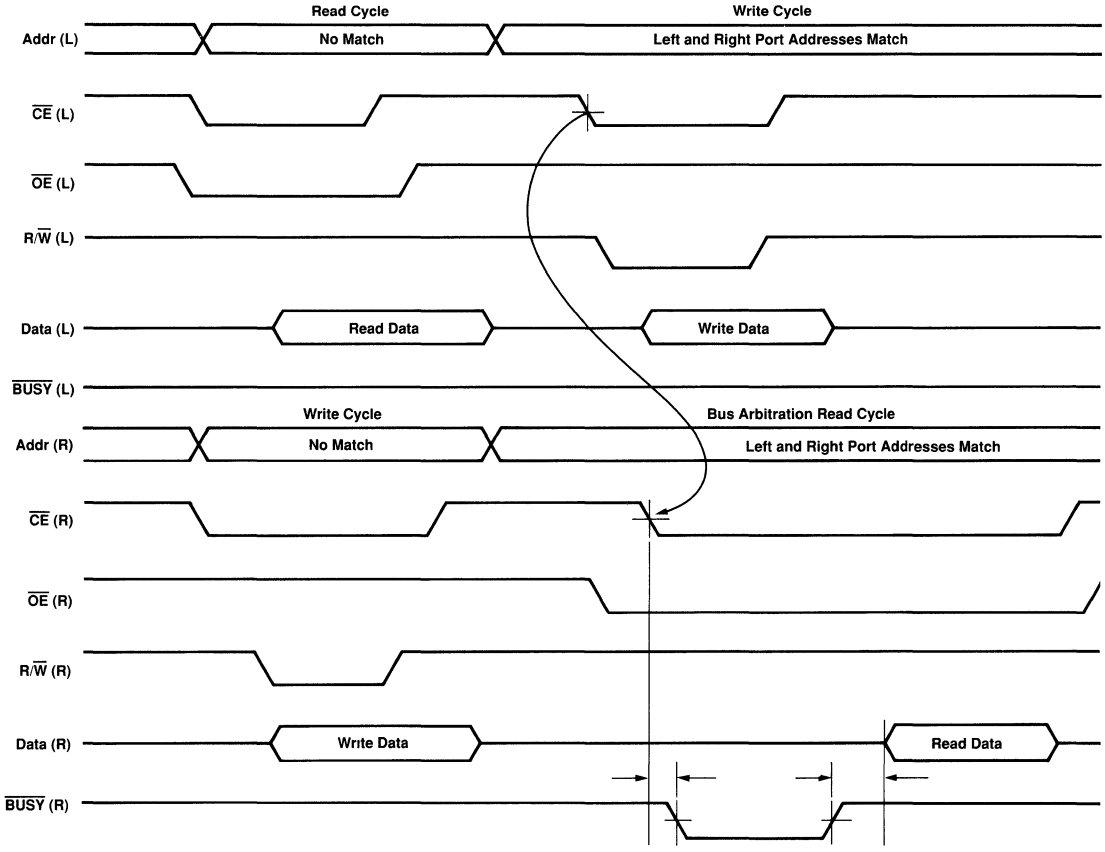
Pin Descriptions

Symbol	Type	Pin Name and Function
$I/O(7:0)_L$	I/O	HD(15:8)–This is part of the 16-bit data bus which is used to transfer data to/from the left ports of the dual-port memory.
$I/O(7:0)_R$	I/O	HD(7:0)–This is the other part of the 16-bit data bus which is used to transfer data to/from the right ports of the dual-port memory.
$A(10:0)_L$	O	ADD(15:5)–This is part of the address/data bus that will address each 2K memory through the ACH from the left side of dual-port memory.
$A(10:6)_R$	O	ADD(4:0)–This part of the address/data bus will address a portion of each 2K memory through the ACH from the right side of dual-port memory.
$A(5:0)_R$	I/O	HAD(5:0)–This is the bidirectional host address bus that in the stand-alone mode is configured as an output. As part of the 22-bit address counter, (ADD(15:0) and HAD(5:0)) it is used here to output the lower 6 address lines through the ACL which address the right side of dual-port memory.
$R/\overline{W}_L$ and $R/\overline{W}_R$	I/O	These Read/Write signals are tied to I/O(7:6) in the PAC1000 and are used as outputs. They control the read/write function of each side of dual-port memory in conjunction with the other control signals.
$\overline{OE}_L$ and $\overline{OE}_R$	I/O	These Output Enable signals are tied to I/O(5:4) in the PAC1000 and are used as outputs. They control the read function of each side of dual-port memory in conjunction with the other control signals.
$\overline{CE}(15:8)_L$	O	These dual-port memory Chip Selects select one of the eight left memory ports. They are tied to OC(15:8) in the PAC1000.
$\overline{CE}(7:0)_R$	O	These dual-port memory Chip Selects select one of the eight right memory ports. They are tied to OC(7:0) in the PAC1000.
$\overline{BUSY}_L$ and $\overline{BUSY}_R$	I	These active low Busy signals are driven by the dual-port memories and are monitored by the PAC1000 condition code inputs CC(7:6). If one becomes active, the PAC1000 will hold off accessing the other port until it becomes inactive.
$\overline{INTR}_L$ and $\overline{INTR}_R$	I	These active low Interrupt signals are driven by the dual-port memories and are monitored by the PAC1000 condition code inputs CC(5:4). If one becomes active, then one "CPU" has interrupted the other giving it permission to use the memory.
CLK	I	The Clock input to the PAC1000. It also latches the condition codes to ensure the proper set-up time.
– – –	I	$\overline{RSPAC}$. Reset is an asynchronous input signal that initializes the state of the PAC1000. It must be held low for at least two clock cycles.

Timing Considerations

The timing waveforms associated with read, write, and bus arbitration cycles are shown in Figure 5. These waveforms were created from the simulation results which were generated from the PACSIM simulator. The timing is relatively straightforward. Each dual port RAM is accessed similarly to a standard SRAM, except that the BUSY flag needs to be monitored in case of address contention. If the left and right port addresses match during a memory access, then the dual port memory arbitrates between the two ports and decides which port will be chosen. The port not chosen activates its BUSY signal and must wait until the busy goes away before completing the read or write cycle.

Figure 5. Timing Waveforms



Dual-Port Static RAM Test Conditions

There are many ways to test dual-port static RAMs. The following cases illustrate the tests that were devised which should thoroughly cover all of the fault conditions possible for dual-port static RAMs with Interrupts.

Test Case #1: Data Integrity Test for Right Ports

Write alternating bit pattern data (aa/55) to all locations of the eight memories at once through the right port. Then, read all locations, one memory at a time, through the right port, verifying the data is correct.

Test Case #2: Data Integrity Test for Left Ports

Write alternating bit pattern data (66/99) to all locations of the eight memories at once through the left port. Then, read all locations, one memory at a time, through the left port, verifying the data is correct.

Test Case #3: Data Integrity Test for Right/Left Ports

Write alternating bit pattern data (cc/33) for one location in SRAM #1 through the right port and immediately read that same location in memory through the left port, verifying the data is correct. Then test SRAM #2, etc.

Test Case #4: Data Integrity Test for Both Left/Right Ports

Write alternating bit pattern data (aa/55) for one address location in SRAM #1 through the left port and immediately read that same location in memory through the right port, verifying the data is correct. Then test SRAM #2, etc.

Test Case #5: Address Connections Test for Both Ports

Write address at current addresses (which differ by 1) in both left and right ports at the same time (e.g., write left address 00 at address 000 and write right address 01 at address 001 in one cycle, etc.) Then read these locations and check for correct address and continue. When finished with SRAM#1, test SRAM#2.

Test Case #6: Data Connections Test for Both Ports

Write "running 1's" in both the left and right ports (at addresses that differ by 1) at the same time (e.g., write left data 01 at address 000 and write right data 01 at address 001, etc.) Then read these locations and check for correct data. Next shift data left one bit and write to next sequential addresses (e.g., write left data 02 at address 002 and write right data 02 at address 003, etc.) and continue until SRAM #1 tested, then test for SRAM #2, etc.

Test Case #7: Dual-Port Address Arbitration Test/Busy Signal Test for Both Ports

Write to same location of SRAM #1 and #5 at the same time. Monitor busy signal. If Busy, then wait until not busy and continue, otherwise set error flag. Continue for SRAMs #2 and #6, etc.

Test Case #8: Dual-Port Interrupt Activity Test

Write to right port memory location h'7FE' and check for INT_L latch set. Then read from left port and check for INT_L clear. Do the same for INT_R at memory location h'7FF'. After testing SRAMs #1 and #5, go on and test SRAMs #2 and #6, etc.

All the algorithms described above are internally realized by software. Code implementation for each of these cases can be found in Appendix 1. The code listings contain sufficient explanations that let the reader understand the subjects they describe.

Simulation

The preceeding algorithms have been assembled with no errors. After assembly, and before simulation and programming, the files must be linked. An example of the linker program which links separate sub-programs and places them at a predefined location is shown in Appendix 2.

After the above files have been linked, the program can be simulated and parts can be programmed. An additional input file is needed before simulation can begin. That is the stimulus file. The command file is another input file that can be useful.

The stimulus file is used to drive inputs such as address, data, condition codes and control signals, an example of which is shown in Appendix 3. The command file is an optional batch file that contains a series of valid PACSIM commands, also shown in Appendix 3.

The PAC1000 functional simulator, PACSIM, records the state of specified signals at each cycle. Simulation results of some of the above algorithms are shown in Appendix 4.

PAC1000 Resources Usage

Using the PAC1000 as a dual-port memory tester in this application, utilizes many of the resources available on the chip. However, it does not take advantage of the part being used in the peripheral mode or slave configuration, where the host processor can request a command or download data to be used at a later time.

In addition, as seen in Figure 4, there are several pins on the PAC1000, such as I/Os, and interrupts which are not used. These pins can be taken advantage of by performing other operations in parallel, without any performance degradation during dual-port memory testing.

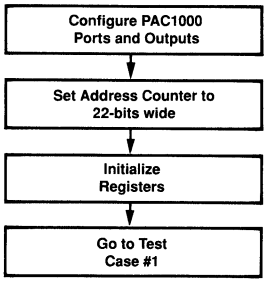
Summary

The PAC1000 architecture is unique in that it enables the part to be configured in a wide range of applications. As exemplified in the circuit diagram where virtually no interface glue logic was required and, in the program code where tasks were handled in very few one-cycle instructions, the PAC1000 enables easy system interfacing as well as efficient task handling. So, whether the PAC1000 performs as an intelligent I/O Controller or as a simple Dual-Port Memory Tester, its flexibility provides the high-level of control that today's circuit designers need in many high-performance systems.

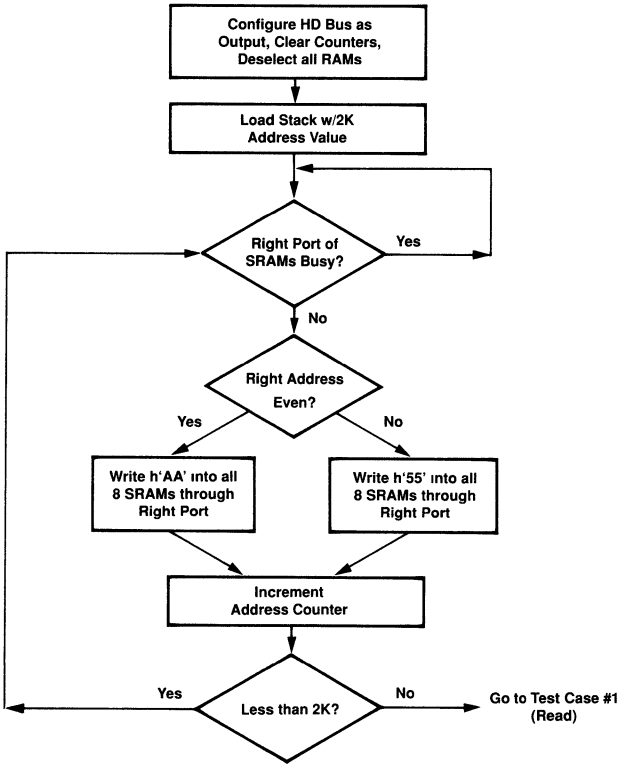
Notes

Appendix 1. PAC1000 Program Flow Charts

Initialization:

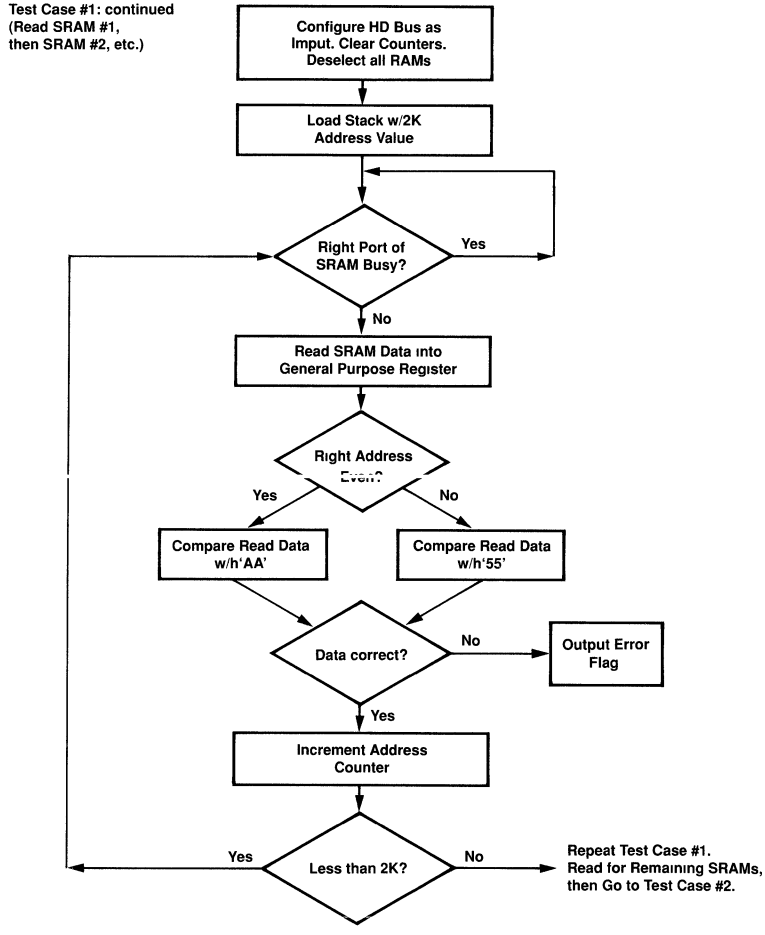


Test Case #1: Write alternating bit pattern data to all locations of each Dual Port RAM through its right port.



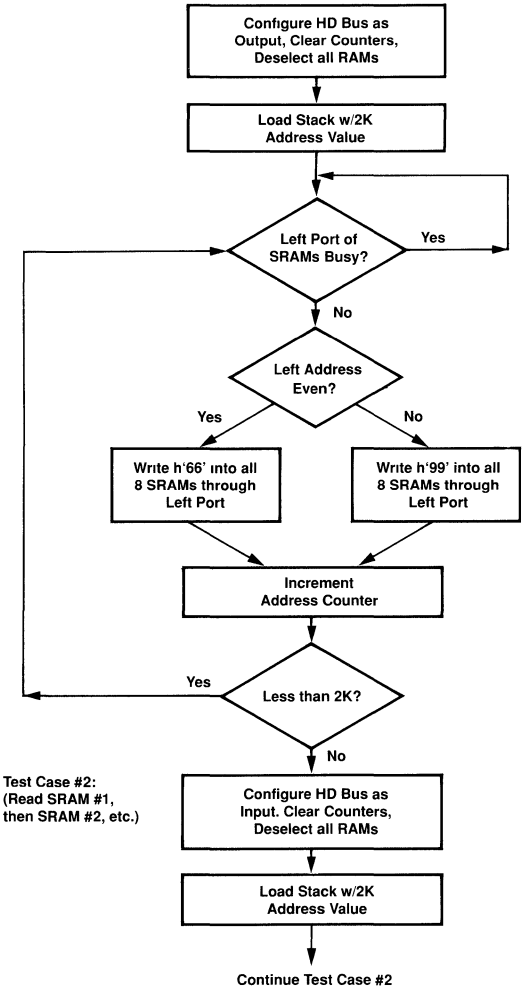
4

Appendix 1. PAC1000 Program Flow Charts (Cont.)

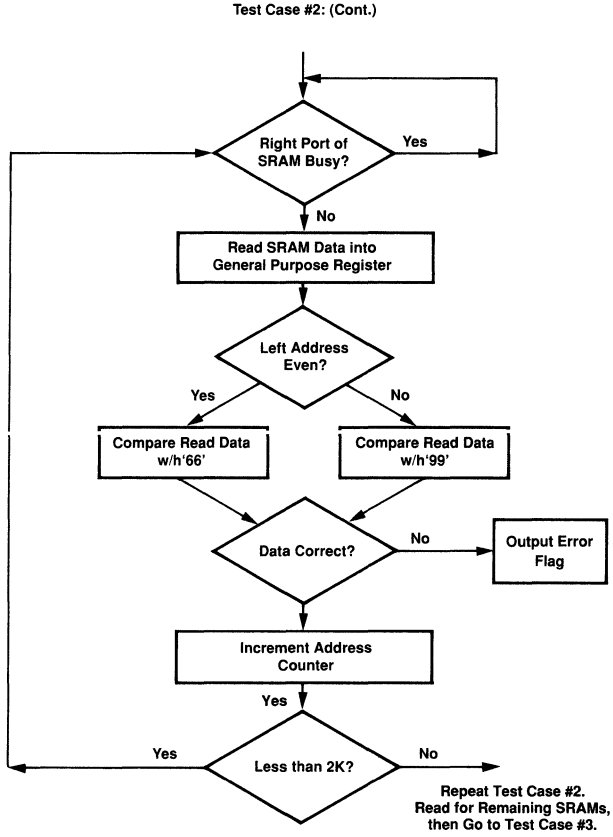


Appendix 1. PAC1000 Program Flow Charts (Cont.)

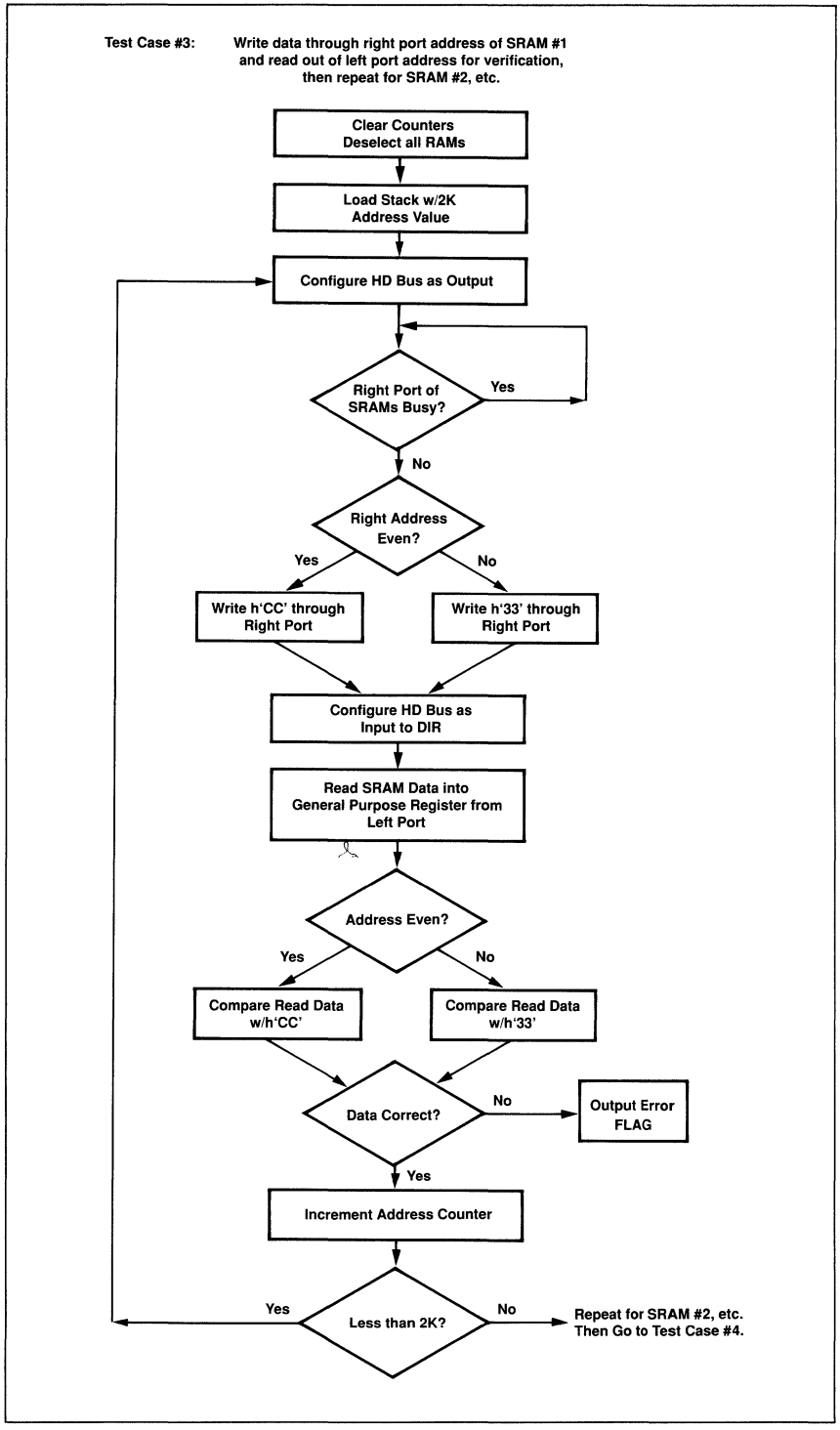
Test Case #2: Write alternating bit pattern data to all locations of each Dual Port RAM through its left port.



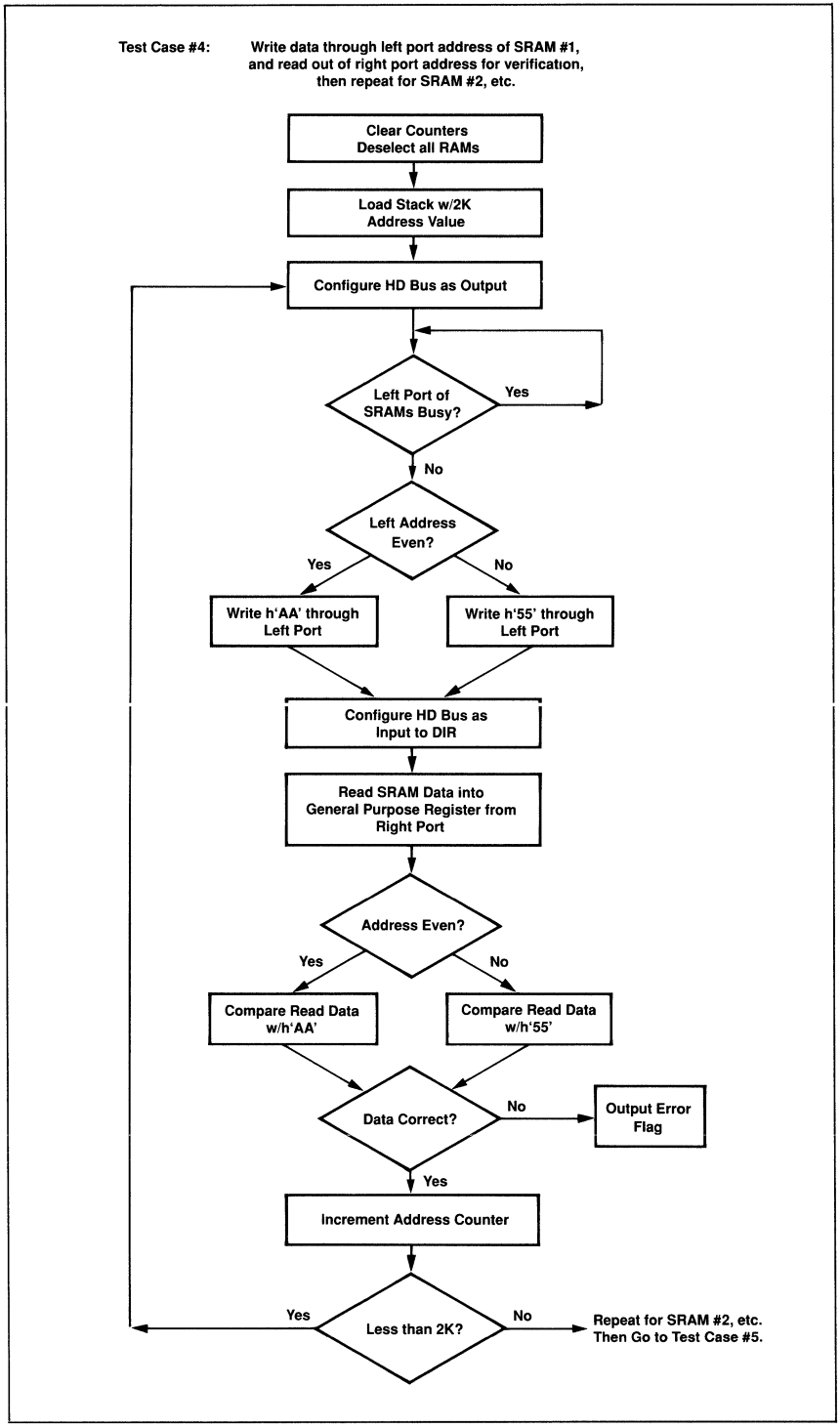
Appendix 1. PAC1000 Program Flow Charts (Cont.)



Appendix 1. PAC1000 Program Flow Charts (Cont.)

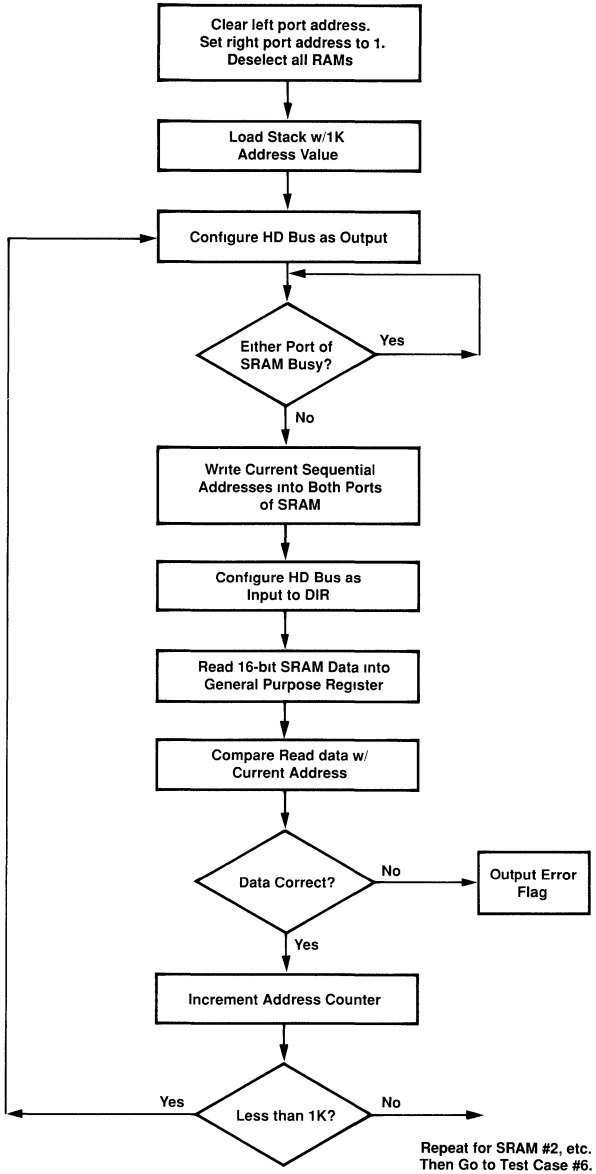


Appendix 1. PAC1000 Program Flow Charts (Cont.)

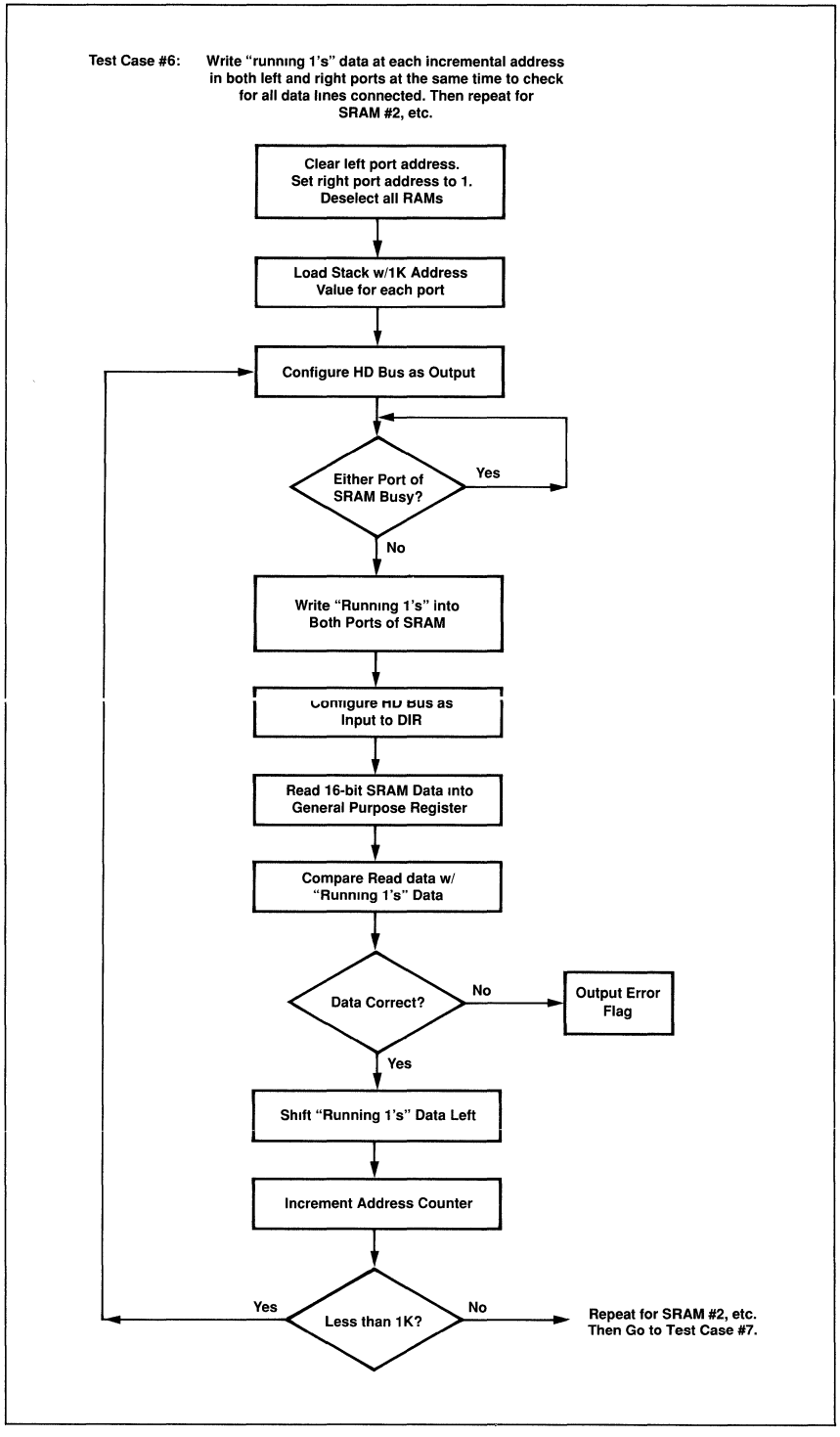


Appendix 1. PAC1000 Program Flow Charts (Cont.)

Test Case #5: Write sequential address data at current address in both left and right ports at the same time to check for address lines not connected. Then repeat for SRAM #2, etc.

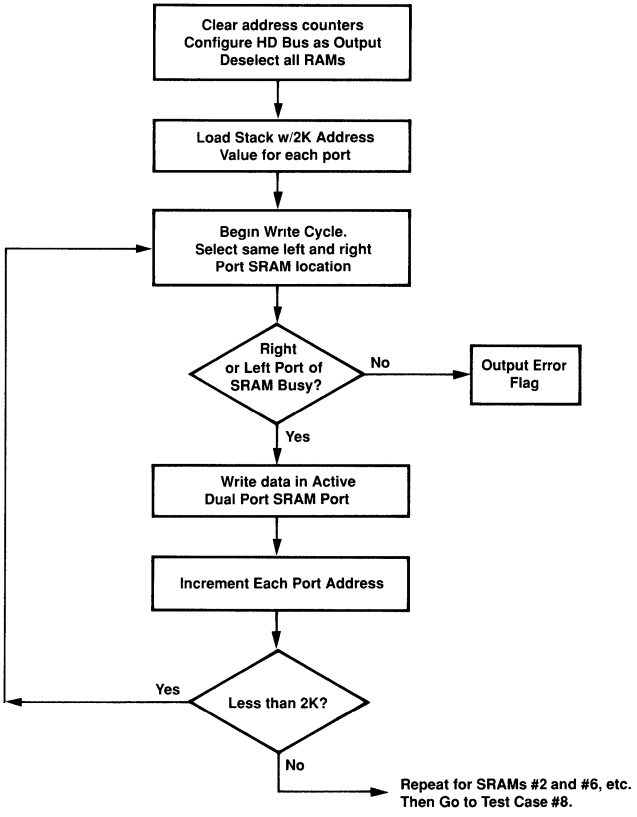


Appendix 1. PAC1000 Program Flow Charts (Cont.)



Appendix 1. PAC1000 Program Flow Charts (Cont.)

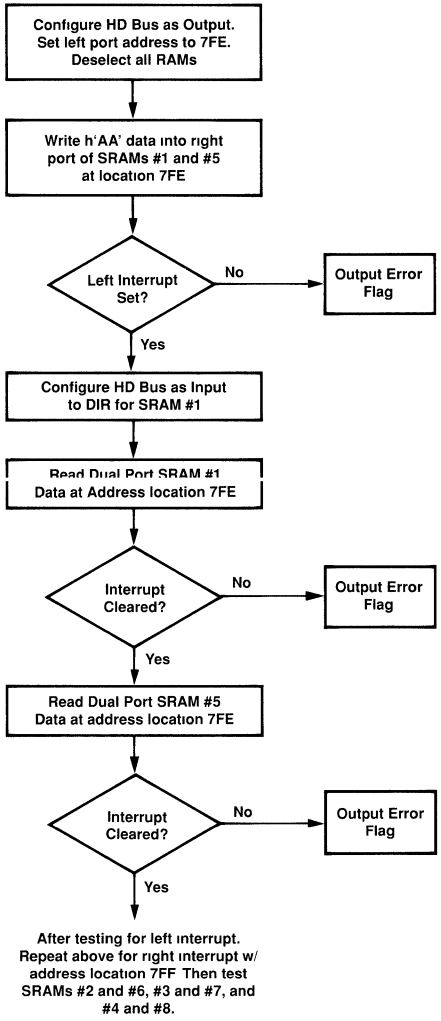
Test Case #7: Try to write data at same address of each port for SRAMs #1 and #5. Test for Active Busy Signal. Continue for all addresses. Then repeat for SRAMs #2 and #6, #3 and #7, and #4 and #8.



4

Appendix 1. PAC1000 Program Flow Charts (Cont.)

Test Case #8: Test Right and Left interrupt flag functions of each Dual Port SRAM for SRAMs #1 and #5. Then repeat for SRAMs #2 and #6, #3 and #7, and #4 and #8.



Appendix 1. PAC1000 Program Illustrations (Assembly Code)

```
segment dual_port_memory_test, abs(0);
entry Beginning, Start;

org h'0000';

include 'dpmtst.equ';

include 'dpmtst.als';

/*************************************************************************/
/*Initialization                                                         */
/*************************************************************************/

Beginning:

        JMP Start, OUT SRAMNO;

org h'0010';

Start:  SET ASEL ADOE HADOE HDSEL0, OUT SRAMNO;  /*Select
        address counter as source that will write to address bus,
        select address bus direction as output, select host address
        bus direction as output, and select the data output
        register to be connected to the host data bus.*/

        ACSIZE 22;
        MOV IOR h'ff'; /*Select address counter to be 22-bits wide*/
        SETIO h'ff';

        DATAA  := h'00aa';   /*(R0) Alternate bit pattern test data*/
        DAT55  := h'0055';   /*(R1) for writing to dual-port*/
        DAT66  := h'6600';   /*(R2) static RAMs*/
        DAT99  := h'9900';   /*(R3)*/
        DATCC  := h'00cc';   /*(R4)*/
        DAT33  := h'0033';   /*(R5)*/
        DATAA2 := h'aa00';   /*(R6)*/
        DATAA3 := h'aaaa';   /*(R7)*/
        DAT552 := h'5500';   /*(R8)*/
        DATCC2 := h'cc00';   /*(R9)*/
        DAT332 := h'3300';   /*(R10)*/

        SRMDIS := h'ff';     /*(R11) SRAM port disable OE, R/W*/
        SRMERR := h'f0';     /*(R12) SRAM error*/
        GPR := h'0000';      /*(R13) general read register*/

        INTLS := h'1F3E';    /*(R14) left interrupt set address (7FE)*/
        INTRC := h'1F3F';    /*(R15) right interrupt clear address (7FF)*/
        INTRS := h'FFFF';    /*(R16) right interrupt set address (7FF)*/
        INTLC := h'FFDF';    /*(R17) left interrupt clear address (7FE)*/

        SRLRPWR := h'003f';  /*(R18) left/write port enable write*/
        SRLRPRD := h'00f3';  /*(R19) left/write port enable read*/
        SRRPWR := h'00bf';   /*(R20) right port enable write*/
        SRLPWR := h'007f';   /*(R21) left port enable write*/
        SRRPRD := h'00ef';   /*(R22) right port enable read*/
```

4

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
        SRLPRD := h'00df';    /*(R23) left port enable read*/

        RUN1S := h'0101';     /*(R24) initialization for "running 1's" test*/
        INC02 := h'0002';     /*(R25) increment address counter*/
        INC40 := h'0040';     /*(R26) increment address counter*/

        M70   := h'00ff';     /*(R27) mask for D(7:0)*/
        M158  := h'ff00';     /*(R28) mask for D(15:8)*/

        INC20 := h'0020';     /*(R29) increment address counter*/
        SET1  := h'0001';     /*(R31) initialization for address counters*/

/*TEST CASE #1*/
/*************************************************************************/
/*This section writes alternating bit pattern data to all locations     */
/*of each dual port SRAM through its right port.                         */
/*************************************************************************/

        SET HDOE, CLR ACH, OUT SRAMNO;
        LDLC h'1F';           /*ACH ctr is output on ADD(4:0) MSB*/
        RESET DIREN, CLR ACL;
LOOP2:  PLDLC h'3F';          /*ACL ctr is output on HAD(5:0) LSB*/
LOOP1:  SWITCH CG1, AND 0 ACL SET1, OUT SRAM1_8R;
                              /*Check if right port of SRAMs not busy (CG6 and CG4),
                              Select right SRAM port and generate WR strobe, check
                              for odd or even address.*/
         CASE 00, GOTO LOOP1;/*CG7=0, CG6=0, CG5=0, CG4=0*/
         CASE 01, GOTO LOOP1;/*CG7=0, CG6=0, CG5=0, CG4=1*/
         CASE 02, GOTO LOOP1;/*CG7=0, CG6=0, CG5=1, CG4=0*/
         CASE 03, GOTO LOOP1;/*CG7=0, CG6=0, CG5=1, CG4=1*/
         CASE 04, GOTO LOOP1;/*CG7=0, CG6=1, CG5=0, CG4=0*/
         CASE 05, GOTO NEXT1;/*CG7=0, CG6=1, CG5=0, CG4=1*/
         CASE 06, GOTO LOOP1;/*CG7=0, CG6=1, CG5=1, CG4=0*/
         CASE 07, GOTO NEXT1;/*CG7=0, CG6=1, CG5=1, CG4=1*/
         CASE 08, GOTO LOOP1;/*CG7=1, CG6=0, CG5=0, CG4=0*/
         CASE 09, GOTO LOOP1;/*CG7=1, CG6=0, CG5=0, CG4=1*/
         CASE 10, GOTO LOOP1;/*CG7=1, CG6=0, CG5=1, CG4=0*/
         CASE 11, GOTO LOOP1;/*CG7=1, CG6=0, CG5=1, CG4=1*/
         CASE 12, GOTO LOOP1;/*CG7=1, CG6=1, CG5=0, CG4=0*/
         CASE 13, GOTO NEXT1;/*CG7=1, CG6=1, CG5=0, CG4=1*/
         CASE 14, GOTO LOOP1;/*CG7=1, CG6=1, CG5=1, CG4=0*/
         CASE 15, GOTO NEXT1;/*CG7=1, CG6=1, CG5=1, CG4=1*/
NEXT1:   ENDSWITCH;

        IF Z, MOV IOR SRRPWR;
          MOV DOR DATAA;      /*write h'aa' data into SRAM right port*/
        ELSE;
          MOV DOR DAT55;      /*write h'55' data into SRAM right port*/
        ENDIF;

        MOV IOR SRMDIS;       /*end write cycle*/
        LOOPNZ LOOP1,  ACL := ++ ACL, OUT SRAMNO;
                              /*Increment HAD(5:0)-ACL address counter,
                              deselect WR and CS*/
        POPLC, ACH := ++ ACH;
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
        /*Increment ADD(4:0)-ACH address counter*/
      LOOPNZ LOOP2;        /*end of outer loop*/

/*************************************************************************/
/*This section reads and checks data from dual port SRAM #1              */
/*through its right port.  After checking SRAM #1, then check SRAM#2 etc*/
/*************************************************************************/

      RESET HDOE, CLR ACL; /* Clear ACL, Enable writing to Data Input Register*/
      SET DIREN, CLR ACH;  /* Clear ACH, Select Host data to be used as an
                           input*/
      LDLC h'1F';          /*ACH ADD(4:0) MSB*/
LOOP4: PLDLC h'3F';        /*ACL HAD(5:0) LSB*/
LOOP3: IF NOT CC6, MOV IOR SRRPRD, OUT SRAM1R;
                           /*Enable OE for read, Enable CS to select SRAM*/
        JMP LOOP3;         /*Right SRAM Busy*/
      ENDIF;

      AND GPR M70 DIR;     /*Read dual port RAM data and write it into GPR*/
      AND Q ACL SET1;      /*Check for odd or even address*/
      IF Z, MOV IOR SRMDIS, OUT SRAMNO;
                           /*End read cycle, release /OE and /CE*/
        CMP GPR DATAA;     /*Check for data integrity*/
      ELSE;
        CMP GPR DAT55;     /*read data into PAC*/
      ENDIF;

      JMPC Z ERROR;
      LOOPNZ LOOP3, ACL := ++ ACL;
                           /*increment address for next read*/
      POPLC, ACH := ++ ACH;
      LOOPNZ LOOP4;        /*end of outer loop*/

/*TEST CASE #2*/
/*************************************************************************/
/*This section writes alternating bit pattern data to all locations     */
/*of each dual port SRAM through its left port.                          */
/*************************************************************************/

      CLR ACH, SET HDOE, OUT SRAMNO;
                           /*clear ACH counter*/
      LDLC h'7FF';         /*ACH ADD(15:5) MSB*/
      RESET DIREN, CLR ACL;/*clear ACL counter*/
LOOP19: SWITCH CG1, AND Q ACH INC20, OUT SRAM1_8L;
                           /*Check for left port of SRAMs not busy (CG7 and CG5), Sele
                           left SRAM port and generate WR strobe, check for odd
                           or even address.*/
       CASE 00, GOTO LOOP19;/*CG7=0, CG6=0, CG5=0, CG4=0*/
       CASE 01, GOTO LOOP19;/*CG7=0, CG6=0, CG5=0, CG4=1*/
       CASE 02, GOTO LOOP19;/*CG7=0, CG6=0, CG5=1, CG4=0*/
       CASE 03, GOTO LOOP19;/*CG7=0, CG6=0, CG5=1, CG4=1*/
       CASE 04, GOTO LOOP19;/*CG7=0, CG6=1, CG5=0, CG4=0*/
       CASE 05, GOTO LOOP19;/*CG7=0, CG6=1, CG5=0, CG4=1*/
       CASE 06, GOTO LOOP19;/*CG7=0, CG6=1, CG5=1, CG4=0*/
       CASE 07, GOTO LOOP19;/*CG7=0, CG6=1, CG5=1, CG4=1*/
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
          CASE 08, GOTO LOOP19;/*CG7=1, CG6=0, CG5=0, CG4=0*/
          CASE 09, GOTO LOOP19;/*CG7=1, CG6=0, CG5=0, CG4=1*/
          CASE 10, GOTO NEXT2; /*CG7=1, CG6=0, CG5=1, CG4=0*/
          CASE 11, GOTO NEXT2; /*CG7=1, CG6=0, CG5=1, CG4=1*/
          CASE 12, GOTO LOOP19;/*CG7=1, CG6=1, CG5=0, CG4=0*/
          CASE 13, GOTO LOOP19;/*CG7=1, CG6=1, CG5=0, CG4=1*/
          CASE 14, GOTO NEXT2; /*CG7=1, CG6=1, CG5=1, CG4=0*/
          CASE 15, GOTO NEXT2; /*CG7=1, CG6=1, CG5=1, CG4=1*/
NEXT2:    ENDSWITCH;

        IF Z, MOV IOR SRLPWR;
          MOV DOR DAT66;      /*write h'66' data into SRAM*/
        ELSE;
          MOV DOR DAT99;      /*write h'99' data into SRAM*/
        ENDIF;

        MOV IOR SRMDIS;       /*end write cycle*/
        LOOPNZ LOOP19, ADD ACH INC20, OUT SRAMNO;
                              /*add h'0020' to ACH - inc left port address*/

/*************************************************************************/
/*This section reads and checks data from dual port SRAM #1              */
/*through its left port                                                  */
/*************************************************************************/

        RESET HDOE, CLR ACL; /*Clear ACL, Enable writing to Data Input Register*/
        SET DIREN, CLR ACH;  /*Clear ACH, Select Host data to be used as an
                             input*/
        LDLC h'7FF';         /*ACH ADD(15:5) MSB*/
LOOP20: IF NOT CC7, MOV IOR SRLPRD, OUT SRAM1L;
                             /*Enable OE for read, Enable CS to select SRAM*/
          JMP LOOP20; /*Left SRAM Busy*/
        ENDIF;

        AND GPR M158 DIR;    /*Read dual port RAM data and write it into GPR*/
        AND Q ACH INC20;     /*Check for odd or even address*/
        IF Z RDEVEN9, MOV IOR SRMDIS, OUT SRAMNO;
                             /*End read cycle, Release OE and CS*/
          CMP GPR DAT66;
        ELSE;
          CMP GPR DAT99;
        ENDIF;

        JMPC Z ERROR;
        LOOPNZ LOOP20, ADD ACH INC20;
                             /*increment address*/

/*TEST CASE #3*/
/*************************************************************************/
/* This section alternately writes data through a right port address and */
/* reads out of the left port address for verification, then increments  */
/* each address of dual port SRAM#1. Repeat for other SRAMs.             */
/*************************************************************************/
/*writing*/
        CLR ACH, OUT SRAMNO;
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
        CLR ACL;
        LDLC h'1F';          /*ACH ADD(4:0) MSB - LOAD Stack w/2K */
LOOP29: PLDLC h'3F';         /*ACL HAD(5:0) LSB - address*/
LOOP30: SET HDOE;
        RESET DIREN;
        IF NOT CC7, AND Q ACL SET1, OUT SRAM1R;
                             /*Select right or left SRAM port and generate WR
                             strobe, check for odd or even address.*/
          JMP LOOP30;        /*Left SRAM Busy*/
        ENDIF;

        IF Z, MOV IOR SRRPWR;
         MOV DOR DATCC;      /*write h'cc' data into SRAM*/
        ELSE;
         MOV DOR DAT33;      /*write h'33' data into SRAM*/
        ENDIF;

        MOV IOR SRMDIS;      /*Release R/W*/

/*reading*/
        RESET HDOE, OUT SRAMNO;
                             /*Selects Host data to be used as an input*/
        SET DIREN;           /*Enables writing to Data Input Register*/
        MOV IOR SRLPRD, OUT SRAM1L;
                             /*Enable OE for read, Enable CS to select SRAM*/
        AND GPR M158 DIR;    /*Read dual port RAM data and write it into GPR*/
        AND Q ACH INC20;     /*Check for odd or even address*/
        IF Z, MOV IOR SRMDIS, OUT SRAMNO;
                             /*End read cycle, Release OE*/
          CMP GPR DATCC2;
        ELSE;
          CMP GPR DAT332;
        ENDIF;

        JMPC Z ERROR, ADD ACH INC20;
                             /*Increment ADD(15:5) by h'0020' (ACH address
                             ctr)- for read*/

        LOOPNZ LOOP30, ACL := ++ ACL;
                             /*Increment HAD(5:0)-ACL address counter for write,
                             deselect WR and CS*/
        POPLC, ACH := ++ ACH;/*Increment ADD(4:0)-ACH address counter for write*/
        LOOPNZ LOOP29;       /*end of outer loop*/

/*TEST CASE #4*/
/*************************************************************************/
/* This section alternately writes data through a left port address and  */
/* reads out of the right port address for verification, then increments */
/* each address of dual port SRAM#1.                                     */
/*************************************************************************/
/*writing*/
        CLR ACH, OUT SRAMNO;
                             /*clear ACH counter*/
        CLR ACL;
                             /*clear ACL counter*/
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
        LDLC h'1F';          /*ACH ADD(4:0) MSB*/
LOOP45: PLDLC h'3F';         /*ACL HAD(5:0) LSB*/
LOOP46: SET HDOE;
        RESET DIREN;
        IF NOT CC7, AND Q ACH INC20, OUT SRAM1L;
                             /*Check for Left SRAM Busy, Select left SRAM port
                             and generate WR strobe. check for odd or even address.*/
          JMP LOOP46;        /*Left SRAM Busy*/
        ENDIF;

        IF Z, MOV IOR SRLPWR;
         MOV DOR DATAA2;      /*write h'66' data into SRAM*/
        ELSE;
         MOV DOR DAT552;      /*write h'99' data into SRAM*/
        ENDIF;

        MOV IOR SRMDIS;      /*end write cycle*/

/*reading*/
        RESET HDOE, OUT SRAMNO;
                             /*Enable writing to Data Input Register*/
        SET DIREN;           /*Select Host data to be used as an input*/
        MOV IOR SRRPRD, OUT SRAM1R;
                             /*Enable OE for read, Enable CS to select SRAM*/
        AND GPR M70 DIR;     /*Read dual port RAM data and write it into SRMDIS*/
        AND Q ACL SET1;      /*Check for odd or even address*/
        IF Z RDEVEN25, MOV IOR SRMDIS, OUT SRAMNO;
                             /*End read cycle, Release OE and CS*/
          CMP GPR DATAA;
        ELSE;
          CMP GPR DAT55;
        ENDIF;

        JMPC Z ERROR, ADD ACH INC20;
                             /*Increment ADD(15:5) h'0020 for ACH address ctr
                             for write*/

        LOOPNZ LOOP46, ACL := ++ ACL;
                             /*Increment HAD(5:0)-ACL address counter for read*/
        POPLC, ACH := ++ ACH;/*Increment ADD(4:0)-ACH address counter for read*/
        LOOPNZ LOOP45;       /*end of outer loop*/

/*TEST CASE #5*/
/*************************************************************************/
/* Testing for address lines not connected to SRAM#1 by writing          */
/* sequential addresses at current address in both left and right        */
/* ports at the same time and reading for accuracy.                      */
/*************************************************************************/
/*writing*/
        CLR ACH, OUT SRAMNO;
                             /*clear ACH counter*/
        MOV ACL SET1;
                             /*set ACL counter to 1*/
        LDLC h'0F';          /*ACH ADD(4:0) MSB*/
LOOP61: PLDLC h'1F';         /*ACL HAD(5:0) LSB*/
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
LOOP62: SET HDOE;
        RESET DIREN;
        SWITCH CG1, MOV IOR SRLRPWR, OUT SRAM1LR;
                              /*Check for ports not busy (CG7 or CG6), Select right AND
                              left SRAM port and generate WR strobe*/
         CASE 00, GOTO LOOP62;/*CG7=0, CG6=0, CG5=0, CG4=0*/
         CASE 01, GOTO LOOP62;/*CG7=0, CG6=0, CG5=0, CG4=1*/
         CASE 02, GOTO LOOP62;/*CG7=0, CG6=0, CG5=1, CG4=0*/
         CASE 03, GOTO LOOP62;/*CG7=0, CG6=0, CG5=1, CG4=1*/
                              /*Left and Right ports of SRAM#1 Busy, wait until not
                              Busy*/
         CASE 04, GOTO LOOP62;/*CG7=0, CG6=1, CG5=0, CG4=0*/
         CASE 05, GOTO LOOP62;/*CG7=0, CG6=1, CG5=0, CG4=1*/
         CASE 06, GOTO LOOP62;/*CG7=0, CG6=1, CG5=1, CG4=0*/
         CASE 07, GOTO LOOP62;/*CG7=0, CG6=1, CG5=1, CG4=1*/
                              /*Left port of SRAM#1 Busy, wait until not Busy*/
         CASE 08, GOTO LOOP62;/*CG7=1, CG6=0, CG5=0, CG4=0*/
         CASE 09, GOTO LOOP62;/*CG7=1, CG6=0, CG5=0, CG4=1*/
         CASE 10, GOTO LOOP62;/*CG7=1, CG6=0, CG5=1, CG4=0*/
         CASE 11, GOTO LOOP62;/*CG7=1, CG6=0, CG5=1, CG4=1*/
                              /*Right port of SRAM#1 Busy, wait until not Busy*/
         CASE 12, GOTO NEXT3;/*CG7=1, CG6=1, CG5=0, CG4=0*/
         CASE 13, GOTO NEXT3;/*CG7=1, CG6=1, CG5=0, CG4=1*/
         CASE 14, GOTO NEXT3;/*CG7=1, CG6=1, CG5=1, CG4=0*/
         CASE 15, GOTO NEXT3;/*CG7=1, CG6=1, CG5=1, CG4=1*/
NEXT3:  ENDSWITCH;

        MOV DOR SET1;         /*write address into SRAM*/
        MOV IOR SRMDIS;

/*reading*/
        RESET HDOE, OUT SRAMNO;
                              /*Enable writing to Data Input Register*/
        SET DIREN;            /*Select Host data to be used as an input*/
        MOV IOR SRLRPRD, OUT SRAM1LR;
                              /*Enable OE for read, Enable CS to select both ports
                              of SRAM*/
        MOV GPR DIR;          /*Read dual port RAM data and write it into SRMDIS*/
        MOV IOR SRMDIS, OUT SRAMNO;
                              /*End read cycle, Release OE and CS*/
          CMP GPR SET1;
        JMPC Z ERROR, ADD ACH INC40;
                              /*Increment ADD(15:5) h'0040' for ACH address ctr
                              for write*/
        ADD SET1 h'0202';     /*Increment data to correspond to address*/
        LOOPNZ LOOP62, ADD ACL INC02;
                              /*Increment by 2 HAD(5:0)-ACL address counter
                              for read*/
        POPLC, ACH := ++ ACH;/*Increment ADD(4:0)-ACH address counter for read*/
        LOOPNZ LOOP61;        /*end of outer loop*/
        SET1 := h'0001';

/*TEST CASE #6*/
/*************************************************************************/
/* Testing for data lines not connected for SRAM#1 by writing "running   */
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
/* 1's at each incremental address in both left and right ports at the   */
/* same time.                                                            */
/*************************************************************************/
/*writing*/
        CLR ACH, OUT SRAMNO;
                            /*clear ACH counter*/
        MOV ACL SET1;
                            /*set ACL counter to 1*/
        LDLC h'0F';         /*ACH ADD(4:0) MSB*/
LOOP64: PLDLC h'1F';        /*ACL HAD(5:0) LSB*/
LOOP65: SET HDOE;
        RESET DIREN;
        SWITCH CG1, MOV IOR SRLRPWR, OUT SRAM1LR;
                            /*Check for ports not busy (CG7 and CG6), Select right AND
                            SRAM port and generate WR strobe*/
         CASE 00, GOTO LOOP65;/*CG7=0, CG6=0, CG5=0, CG4=0*/
         CASE 01, GOTO LOOP65;/*CG7=0, CG6=0, CG5=0, CG4=1*/
         CASE 02, GOTO LOOP65;/*CG7=0, CG6=0, CG5=1, CG4=0*/
         CASE 03, GOTO LOOP65;/*CG7=0, CG6=0, CG5=1, CG4=1*/
                            /*Left and Right ports of SRAM#1 Busy, wait until not
                            Busy*/
         CASE 04, GOTO LOOP65;/*CG7=0, CG6=1, CG5=0, CG4=0*/
         CASE 05, GOTO LOOP65;/*CG7=0, CG6=1, CG5=0, CG4=1*/
         CASE 06, GOTO LOOP65;/*CG7=0, CG6=1, CG5=1, CG4=0*/
         CASE 07, GOTO LOOP65;/*CG7=0, CG6=1, CG5=1, CG4=1*/
                            /*Left port of SRAM#1 Busy, wait until not Busy*/
         CASE 08, GOTO LOOP65;/*CG7=1, CG6=0, CG5=0, CG4=0*/
         CASE 09, GOTO LOOP65;/*CG7=1, CG6=0, CG5=0, CG4=1*/
         CASE 10, GOTO LOOP65;/*CG7=1, CG6=0, CG5=1, CG4=0*/
         CASE 11, GOTO LOOP65;/*CG7=1, CG6=0, CG5=1, CG4=1*/
                            /*Right port of SRAM#1 Busy, wait until not Busy*/
         CASE 12, GOTO NEXT4;/*CG7=1, CG6=1, CG5=0, CG4=0*/
         CASE 13, GOTO NEXT4;/*CG7=1, CG6=1, CG5=0, CG4=1*/
         CASE 14, GOTO NEXT4;/*CG7=1, CG6=1, CG5=1, CG4=0*/
         CASE 15, GOTO NEXT4;/*CG7=1, CG6=1, CG5=1, CG4=1*/
NEXT4:  ENDSWITCH;


        MOV DOR RUN1S;      /*write running 1's into SRAM*/
        MOV IOR SRMDIS;     /*end write cycle*/

/*reading*/
        RESET HDOE, OUT SRAMNO;
                            /*Enable writing to Data Input Register*/
        SET DIREN;          /*Select Host data to be used as an input*/
        MOV IOR SRLRPRD, OUT SRAM1LR;
                            /*Enable OE for read, Enable CS to select both ports
                            of SRAM*/
        MOV GPR DIR;        /*Read dual port RAM data and write it into GPR*/
        MOV IOR SRMDIS, OUT SRAMNO;
                            /*End read cycle, Release OE and CS*/
          CMP GPR SET1;
        JMPC Z ERROR, ADD ACH INC40;
                            /*Increment ADD(15:5) h'0040' for ACH address ctr
                            for write*/
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
        SHLR RUN1S RMSB;      /*Shift  and rotate R31 left one bit*/

        LOOPNZ LOOP65, ADD ACL INC02;
                              /*Increment by 2 HAD(5:0)-ACL address counter
                              for read*/
        POPLC, ACH := ++ ACH;/*Increment ADD(4:0)-ACH(4:0)-ACH address counter
                              for read*/
        LOOPNZ LOOP64;        /*end of outer loop*/

/*TEST CASE #7*/
/*************************************************************************/
/* Dual-Port Address Arbitration Test/Busy Signal Test for Both Ports.  */
/* Test for writing to the same location of SRAM #1 and #5 at the same  */
/* time (Expect Busy signal)                                             */
/*************************************************************************/

        SET HDOE, CLR ACH, OUT SRAMNO;
                              /*clear ACH counter*/
        RESET DIREN, CLR ACL;/*clear ACL countera*/
        LDLC h'1F';
LOOP96: PLDLC h'3F';
LOOP97: MOV IOR SRLRPWR, OUT SRAM15LR;
                              /*Select right AND left SRAM port and generate WR
                              strobe to try and write in same address location
                              of both ports*/
        SWITCH CG1;           /*Check for busy signal from left or right port
                              (CG7 or CG6) AND (CG5 or CG4) of each RAM*/
         CASE 00, GOTO ERROR; /*CG7=0, CG6=0, CG5=0, CG4=0*/
         CASE 01, GOTO ERROR; /*CG7=0, CG6=0, CG5=0, CG4=1*/
         CASE 02, GOTO ERROR; /*CG7=0, CG6=0, CG5=1, CG4=0*/
         CASE 03, GOTO ERROR; /*CG7=0, CG6=0, CG5=1, CG4=1*/
         CASE 04, GOTO ERROR; /*CG7=0, CG6=1, CG5=0, CG4=0*/
         CASE 05, GOTO NEXT5; /*CG7=0, CG6=1, CG5=0, CG4=1*/
         CASE 06, GOTO NEXT5; /*CG7=0, CG6=1, CG5=1, CG4=0*/
         CASE 07, GOTO ERROR; /*CG7=0, CG6=1, CG5=1, CG4=1*/
         CASE 08, GOTO ERROR; /*CG7=1, CG6=0, CG5=0, CG4=0*/
         CASE 09, GOTO NEXT5; /*CG7=1, CG6=0, CG5=0, CG4=1*/
         CASE 10, GOTO NEXT5; /*CG7=1, CG6=0, CG5=1, CG4=0*/
         CASE 11, GOTO ERROR; /*CG7=1, CG6=0, CG5=1, CG4=1*/
         CASE 12, GOTO ERROR; /*CG7=1, CG6=1, CG5=0, CG4=0*/
         CASE 13, GOTO ERROR; /*CG7=1, CG6=1, CG5=0, CG4=1*/
         CASE 14, GOTO ERROR; /*CG7=1, CG6=1, CG5=1, CG4=0*/
         CASE 15, GOTO ERROR; /*CG7=1, CG6=1, CG5=1, CG4=1*/
NEXT5:  ENDSWITCH;

        MOV DOR DATAA3;       /*Write data in active port*/
        MOV IOR SRMDIS, OUT SRAMNO;
                              /*End write cycle*/

        ADD ACH INC20;        /*Increment both address ports to test for
                              busy at every location*/
        LOOPNZ LOOP96, ACL := ++ ACL;
        POPLC, ACH := ++ ACH;
        LOOPNZ LOOP97;
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
/*TEST CASE #8*/
/*************************************************************************/
/* Test for dual port SRAM interrupt flag function by writing to right  */
/* port memory location 7FE and checking for INTL set and then reading  */
/* from left port to clear INTL.  Then do same to check for INTR at     */
/* memory location 7FF for 2 dual-port SRAMs at a time (Example for     */
/* SRAMs 1 and 5.) Continue w/SRAMs 2 and 6.                            */
/*************************************************************************/

/*set left interrupt by writing into right port address 7FE*/
SINTL:  SET HDOE, MOV ACH INTLC, OUT SRAMNO;
                          /*set left port address to 7FE*/
        RESET DIREN, MOV ACL INTLS;
                          /*set left port address to 7FE*/
        MOV IOR SRRPWR, OUT SRAM15R;
                          /*Select right SRAM port and generate WR strobe*/
        MOV DOR DATAA;    /*write h'aa' into SRAM*/
        MOV IOR SRMDIS, OUT SRAMNO;
                          /*end write cycle*/
        SWITCH CG0;       /*check for left interrupt set (CC3 and CC1)*/
                          /*for both dual port RAMs #1 and #5*/
         CASE 00, GOTO ERROR;/*CG3=0, CG2=0, CG1=0, CG0=0*/
         CASE 01, GOTO ERROR;/*CG3=0, CG2=0, CG1=0, CG0=1*/
         CASE 02, GOTO ERROR;/*CG3=0, CG2=0, CG1=1, CG0=0*/
         CASE 03, GOTO ERROR;/*CG3=0, CG2=0, CG1=1, CG0=1*/
         CASE 04, GOTO ERROR;/*CG3=0, CG2=1, CG1=0, CG0=0*/
         CASE 05, GOTO CINTL1;/*CG3=0, CG2=1, CG1=0, CG0=1*/
                          /*Left interrupt set, so continue*/
         CASE 06, GOTO ERROR;/*CG3=0, CG2=1, CG1=1, CG0=0*/
         CASE 07, GOTO ERROR;/*CG3=0, CG2=1, CG1=1, CG0=1*/
         CASE 08, GOTO ERROR;/*CG3=1, CG2=0, CG1=0, CG0=0*/
         CASE 09, GOTO ERROR;/*CG3=1, CG2=0, CG1=0, CG0=1*/
         CASE 10, GOTO ERROR;/*CG3=1, CG2=0, CG1=1, CG0=0*/
         CASE 11, GOTO ERROR;/*CG3=1, CG2=0, CG1=1, CG0=1*/
         CASE 12, GOTO ERROR;/*CG3=1, CG2=1, CG1=0, CG0=0*/
         CASE 13, GOTO ERROR;/*CG3=1, CG2=1, CG1=0, CG0=1*/
         CASE 14, GOTO ERROR;/*CG3=1, CG2=1, CG1=1, CG0=0*/
         CASE 15, GOTO ERROR;/*CG3=1, CG2=1, CG1=1, CG0=1*/
        ENDSWITCH;

/*Clear left interrupt of SRAM #1 and #5 by reading from left port */
/*address 7FE:*/

CINTL1: RESET HDOE;       /*Enables writing to Data Input Register*/
        SET DIREN;        /*Selects host data to be used an input*/
        MOV IOR SRLPRD, OUT SRAM1L;
                          /*Enable OE for read, Enable CS to select SRAM*/
        MOV GPR DIR;      /*Read dual port RAM data*/
        MOV IOR SRMDIS, OUT SRAMNO;
                          /*End read cycle for dual port SRAM#1, Release OE*/
        JMPC CC3 CINTL5;  /*check for clear interrupt, if clear continue*/
        JMP ERROR;

CINTL5: MOV IOR SRLPRD, OUT SRAM5L;
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
                              /*Enable OE for read, Enable CS to select SRAM*/
        MOV GPR DIR;          /*Read dual port RAM data*/
        MOV IOR SRMDIS, OUT SRAMNO;
                              /*End read cycle for dual port SRAM#5, Release OE*/
        JMPC CC1 SINTR;       /*check for left clear interrupt, if clear, continue*/
        JMP ERROR;

/*Set right interrupt by writing into left port address 7FF*/

SINTR:  SET HDOE, MOV ACH INTRS, OUT SRAMNO;
                              /*set right port address to 7FF*/
        RESET DIREN, MOV ACL INTRC;
                              /*set left port address to 7FF*/
        MOV IOR SRLPWR, OUT SRAM15L;
                              /*Select left SRAM port and generate WR strobe*/
        MOV DOR DATAA2;       /*write h'aa' into SRAM*/
        MOV IOR SRMDIS, OUT SRAMNO;
                              /*end write cycle*/
        SWITCH CG0;           /*Check for right interrupt set (CC2,CC0)*/
                              /*for both dual port RAMs #1 and #5*/
         CASE 00, GOTO ERROR;/*CG3=0, CG2=0, CG1=0, CG0=0*/
         CASE 01, GOTO ERROR;/*CG3=0, CG2=0, CG1=0, CG0=1*/
         CASE 02, GOTO ERROR;/*CG3=0, CG2=0, CG1=1, CG0=0*/
         CASE 03, GOTO ERROR;/*CG3=0, CG2=0, CG1=1, CG0=1*/
         CASE 04, GOTO ERROR;/*CG3=0, CG2=1, CG1=0, CG0=0*/
         CASE 05, GOTO ERROR;/*CG3=0, CG2=1, CG1=0, CG0=1*/
         CASE 06, GOTO ERROR;/*CG3=0, CG2=1, CG1=1, CG0=0*/
         CASE 07, GOTO ERROR;/*CG3=0, CG2=1, CG1=1, CG0=1*/
         CASE 08, GOTO ERROR;/*CG3=1, CG2=0, CG1=0, CG0=0*/
         CASE 09, GOTO ERROR;/*CG3=1, CG2=0, CG1=0, CG0=1*/
         CASE 10, GOTO CINTR1;/*CG3=1, CG2=0, CG1=1, CG0=0*/
                              /*Right interrupt set, so continue*/
         CASE 11, GOTO ERROR;/*CG3=1, CG2=0, CG1=1, CG0=1*/
         CASE 12, GOTO ERROR;/*CG3=1, CG2=1, CG1=0, CG0=0*/
         CASE 13, GOTO ERROR;/*CG3=1, CG2=1, CG1=0, CG0=1*/
         CASE 14, GOTO ERROR;/*CG3=1, CG2=1, CG1=1, CG0=0*/
         CASE 15, GOTO ERROR;/*CG3=1, CG2=1, CG1=1, CG0=1*/
        ENDSWITCH;

/*Clear right interrupt of SRAM #1 and #5 by reading from right port */
/*address 7FF:*/

CINTR1: RESET HDOE;           /*Enable writing to Data Input Register*/
        SET DIREN;            /*Select host data to be used an input*/
        MOV IOR SRRPRD, OUT SRAM1R;
                              /*Enable OE for read, Enable CS to select SRAM*/
        MOV GPR DIR;          /*Read dual port RAM data*/
        MOV IOR SRMDIS, OUT SRAMNO;
                              /*End read cycle for dual port SRAM#1, Release OE*/
        JMPC CC2 CINTR5;      /*check for clear interrupt, if clear continue*/
        JMP ERROR;

CINTR5: MOV IOR SRRPRD, OUT SRAM5R;
                              /*Enable OE for read, Enable CS to select SRAM*/
        MOV GPR DIR;          /*Read dual port RAM data*/
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

```
        MOV IOR SRMDIS, OUT SRAMNO;
                              /*End read cycle for dual port SRAM#5, Release OE*/
        JMPC CC0 DONE;       /*check for left clear interrupt, if clear, continue*/
        JMP ERROR;

DONE:   JMP DONE;


ERROR: MOV IOR SRMERR, JMP ERROR;

end;
```

Appendix 1. PAC1000 Program Illustrations (Assembly Code) (Cont.)

(Alias File)

```
/*This is an alias file*/

alias DATAA R0;
alias DAT55 R1;
alias DAT66 R2;
alias DAT99 R3;
alias DATCC R4;
alias DAT33 R5;
alias DATAA2 R6;
alias DATAA3 R7;
alias DAT552 R8;
alias DATCC2 R9;
alias DAT332 R10;
alias SRMDIS R11;
alias SRMERR R12;
alias GPR R13;
alias INTLS R14;
alias INTRC R15;
alias INTRS R16;
alias INTLC R17;
alias SRLRPWR R18;
alias SRLRPRD R19;
alias SRRPWR R20;
alias SRLPWR R21;
alias SRRPRD R22;
alias SRLPRD R23;
alias RUN1S R24;
alias INC02 R25;
alias INC40 R26;
alias M70 R27;
alias M158 R28;
alias INC20 R29;
alias REG30 R30;
alias SET1 R31;
```

(Equate File)

```
/*This is an equate file*/

SRAMNO equ h'ffff';
SRAM1L equ h'fffe';
SRAM2L equ h'fffd';
SRAM3L equ h'fffb';
SRAM4L equ h'fff7';
SRAM5L equ h'ffef';
SRAM6L equ h'ffdf';
SRAM7L equ h'ffbf';
SRAM8L equ h'ff7f';
SRAM1R equ h'feff';
SRAM2R equ h'fdff';
SRAM3R equ h'fbff';
SRAM4R equ h'f7ff';
SRAM5R equ h'efff';
SRAM6R equ h'dfff';
SRAM7R equ h'bfff';
SRAM8R equ h'7fff';

SRAM1LR equ h'fefe';
SRAM2LR equ h'fdfd';
SRAM3LR equ h'fbfb';
SRAM4LR equ h'f7f7';
SRAM5LR equ h'efef';
SRAM6LR equ h'dfdf';
SRAM7LR equ h'bfbf';
SRAM8LR equ h'7f7f';

SRAM1_8L equ h'ff00';
SRAM1_8R equ h'00ff';

SRAM15LR equ h'eeee';
SRAM26LR equ h'dddd';
SRAM37LR equ h'bbbb';
SRAM48LR equ h'7777';

SRAM15R equ h'00ee';
SRAM15L equ h'ee00';
SRAM26R equ h'00dd';
SRAM26L equ h'dd00';
SRAM37R equ h'00bb';
SRAM37L equ h'bb00';
SRAM48R equ h'0077';
SRAM48L equ h'7700';
```

Appendix 2. Linker File Example

```
/*This is a linker file*/

place dual_port_memory_test;
load dpmtst;
end;
```

Appendix 3. Stimulus File Example

```
/*This is a stimulus file*/

.S RESETB  0@1 1@2;
 S RDB 0@1 1@97 ;
.S CSB 0@1 1@97 ;
.S HD15  Z@1 0@95 1@102  0@109  1@119  0@126  1@133  Z@140  0@193  1@200  0@207  1@214
.S HD14  Z@1 1@95 0@102  1@109  0@119  1@126  0@133  Z@140  1@193  0@200  1@207  0@214
.S HD13  Z@1 0@95 1@102  0@109  1@119  0@126  1@133  Z@140  1@193  0@200  1@207  0@214
.S HD12  Z@1 1@95 0@102  1@109  0@119  1@126  0@133  Z@140  0@193  1@200  0@207  1@214
.S HD11  Z@1 0@95 1@102  0@109  1@119  0@126  1@133  Z@140  0@193  1@200  0@207  1@214
.S HD10  Z@1 1@95 0@102  1@109  0@119  1@126  0@133  Z@140  1@193  0@200  1@207  0@214
.S HD9   Z@1 0@95 1@102  0@109  1@119  0@126  1@133  Z@140  1@193  0@200  1@207  0@214
.S HD8   Z@1 1@95 0@102  1@109  0@119  1@126  0@133  Z@140  0@193  1@200  0@207  1@214
.S HD7   Z@1 1@95 0@102  1@109  0@119  1@126  0@133  Z@140  1@193  0@200  1@207  0@214
.S HD6   Z@1 0@95 1@102  0@109  1@119  0@126  1@133  Z@140  0@193  1@200  0@207  1@214
.S HD5   Z@1 1@95 0@102  1@109  0@119  1@126  0@133  Z@140  0@193  1@200  0@207  1@214
.S HD4   Z@1 0@95 1@102  0@109  1@119  0@126  1@133  Z@140  1@193  0@200  1@207  0@214
.S HD3   Z@1 1@95 0@102  1@109  0@119  1@126  0@133  Z@140  1@193  0@200  1@207  0@214
.S HD2   Z@1 0@95 1@102  0@109  1@119  0@126  1@133  Z@140  0@193  1@200  0@207  1@214
.S HD1   Z@1 1@95 0@102  1@109  0@119  1@126  0@133  Z@140  0@193  1@200  0@207  1@214
.S HD0   Z@1 0@95 1@102  0@109  1@119  0@126  1@133  Z@140  1@193  0@200  1@207  0@214
.S ADD15 Z@1 ;
.S ADD14 Z@1 ;
.S ADD13 Z@1 ;
.S ADD12 Z@1 ;
.S ADD11 Z@1 ;
.S ADD10 Z@1 ;
.S ADD9  Z@1 ;
.S ADD8  Z@1 ;
.S ADD7  Z@1 ;
.S ADD6  Z@1 ;
.S ADD5  Z@1 ;
.S ADD4  Z@1 ;
.S ADD3  Z@1 ;
.S ADD2  Z@1 ;
.S ADD1  Z@1 ;
.S ADD0  Z@1 ;
.S HAD5  Z@1 ;
.S HAD4  Z@1 ;
.S HAD3  Z@1 ;
.S HAD2  Z@1 ;
.S HAD1  Z@1 ;
.S HAD0  Z@1 ;
.S IO7   Z@1 ;
.S IO6   Z@1 ;
.S IO5   Z@1 ;
.S IO4   Z@1 ;
.S IO3   Z@1 ;
.S IO2   Z@1 ;
.S IO1   Z@1 ;
.S IO0   Z@1 ;
```

Appendix 3. Command File Example

```
/*This is a command file used for batch simulation*/

open journal dpmtst
open stimulus dpmtst
set trace PC
set trace CPC
set trace LC
set trace Z
set trace IOR
set trace IO
set trace OC
set trace HDIR
set trace HDOR
set trace HD
set trace ACH
set trace ADD
set trace ACL
set trace HAD
set trace R10

open trace dpmtst
```

Appendix 4. Simulation Results

```
*******************************************************************************

                              O U T P U T   T A B L E

   P A C S I M      Ver. 3.00b                              Wed Sep 05 10:52:57 1990
*******************************************************************************

           PPP CCC LLL Z II II OOOO HHHH HHHH HHHH AAAA AAAA AA HH RRRR
           CCC PPP CCC   OO OO CCCC DDDD DDDD DDDD CCCC DDDD CC AA 1111
           173 CCC 173   RR 73 1173 IIII OOOO 1173 HHHH DDDD LL DD 0000
           1:: 173 1::   73 :: 51:: RRRR RRRR 51:: 1173 1173 53 53
           :40 1:: :40   :: 40 ::40 1173 1173 ::40 51:: 51:: :: :: 1173
           8   :40 8     40    18   51:: 51:: 18   ::40 ::40 40 40 51::
               8                 2    ::40 ::40 2    18   18        ::40
                                      18   18        2    2         18
                                      2    2                        2
     TIME
        1  000 000 000 0 00 zz 0000 0000 0000 0000 0000 zzzz 00 zz 0000
        2  000 000 000 0 00 zz 0000 0000 0000 0000 0000 zzzz 00 zz 0000
        3  010 000 000 1 00 zz ffff 0000 0000 0000 0000 zzzz 00 zz 0000
        4  011 010 000 1 00 zz ffff 0000 0000 0000 0000 0000 00 00 0000
        5  012 011 000 1 00 zz ffff 0000 0000 0000 0000 0000 00 00 0000
        6  013 012 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
        7  014 013 000 1 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
        8  015 014 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
        9  016 015 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       10  017 016 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       11  018 017 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       12  019 018 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       13  01a 019 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       14  01b 01a 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       15  01c 01b 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       16  01d 01c 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       17  01e 01d 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       18  01f 01e 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       19  020 01f 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       20  021 020 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       21  022 021 000 1 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       22  023 022 000 0 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       23  024 023 000 1 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       24  025 024 001 1 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       25  026 025 001 1 ff ff ffff 0000 0000 0000 0000 0000 00 00 0000
       26  027 026 005 1 ff ff 00ff 0000 0000 0000 0000 0000 00 00 0000
       27  060 027 005 1 ff ff 00ff 0000 0000 0000 0000 0000 00 00 0000
       28  029 060 005 0 bf bf 00ff 0000 0000 0000 0000 0000 00 00 0000
       29  02a 029 005 0 bf bf 00ff 0000 aa55 aa55 0000 0000 00 00 0000
       30  026 02a 005 0 ff ff ffff 0000 aa55 aa55 0000 0000 00 00 0000
       31  027 026 004 0 ff ff 00ff 0000 aa55 aa55 0000 0000 01 01 0000
       32  028 027 004 0 ff ff 00ff 0000 aa55 aa55 0000 0000 01 01 0000
       33  029 028 004 0 bf bf 00ff 0000 aa55 aa55 0000 0000 01 01 0000
       34  02a 029 004 0 bf bf 00ff 0000 55aa 55aa 0000 0000 01 01 0000
       35  026 02a 004 0 ff ff ffff 0000 55aa 55aa 0000 0000 01 01 0000
       36  027 026 003 0 ff ff 00ff 0000 55aa 55aa 0000 0000 02 02 0000
       37  060 027 003 1 ff ff 00ff 0000 55aa 55aa 0000 0000 02 02 0000
```

Appendix 4. Simulation Results (Cont.)

38	029	060	003	0	bf	bf	00ff	0000	55aa	55aa	0000	0000	02	02	0000
39	02a	029	003	0	bf	bf	00ff	0000	aa55	aa55	0000	0000	02	02	0000
40	026	02a	003	0	ff	ff	ffff	0000	aa55	aa55	0000	0000	02	02	0000
41	027	026	002	0	ff	ff	00ff	0000	aa55	aa55	0000	0000	03	03	0000
42	028	027	002	0	ff	ff	00ff	0000	aa55	aa55	0000	0000	03	03	0000
43	029	028	002	0	bf	bf	00ff	0000	aa55	aa55	0000	0000	03	03	0000
44	02a	029	002	0	bf	bf	00ff	0000	55aa	55aa	0000	0000	03	03	0000
45	026	02a	002	0	ff	ff	ffff	0000	55aa	55aa	0000	0000	03	03	0000
46	027	026	001	0	ff	ff	00ff	0000	55aa	55aa	0000	0000	04	04	0000
47	060	027	001	1	ff	ff	00ff	0000	55aa	55aa	0000	0000	04	04	0000
48	029	060	001	0	bf	bf	00ff	0000	55aa	55aa	0000	0000	04	04	0000
49	02a	029	001	0	bf	bf	00ff	0000	aa55	aa55	0000	0000	04	04	0000
50	026	02a	001	0	ff	ff	ffff	0000	aa55	aa55	0000	0000	04	04	0000
51	027	026	000	0	ff	ff	00ff	0000	aa55	aa55	0000	0000	05	05	0000
52	028	027	000	0	ff	ff	00ff	0000	aa55	aa55	0000	0000	05	05	0000
53	029	028	000	0	bf	bf	00ff	0000	aa55	aa55	0000	0000	05	05	0000
54	02a	029	000	0	bf	bf	00ff	0000	55aa	55aa	0000	0000	05	05	0000
55	02b	02a	000	0	ff	ff	ffff	0000	55aa	55aa	0000	0000	05	05	0000
56	02c	02b	000	0	ff	ff	ffff	0000	55aa	55aa	0000	0000	06	06	0000
57	025	02c	001	0	ff	ff	ffff	0000	55aa	55aa	0001	0001	06	06	0000
58	026	025	000	1	ff	ff	ffff	0000	55aa	55aa	0001	0001	06	06	0000
59	027	026	005	1	ff	ff	00ff	0000	55aa	55aa	0001	0001	06	06	0000
60	060	027	005	1	ff	ff	00ff	0000	55aa	55aa	0001	0001	06	06	0000
61	029	060	005	0	bf	bf	00ff	0000	55aa	55aa	0001	0001	06	06	0000
62	02a	029	005	0	bf	bf	00ff	0000	aa55	aa55	0001	0001	06	06	0000
63	026	02a	005	0	ff	ff	ffff	0000	aa55	aa55	0001	0001	06	06	0000
64	027	026	004	0	ff	ff	00ff	0000	aa55	aa55	0001	0001	07	07	0000
65	028	027	004	0	ff	ff	00ff	0000	aa55	aa55	0001	0001	07	07	0000
66	029	028	004	0	bf	bf	00ff	0000	aa55	aa55	0001	0001	07	07	0000
67	02a	029	004	0	bf	bf	00ff	0000	55aa	55aa	0001	0001	07	07	0000
68	026	02a	004	0	ff	ff	ffff	0000	55aa	55aa	0001	0001	07	07	0000
69	027	026	003	0	ff	ff	00ff	0000	55aa	55aa	0001	0001	08	08	0000
70	060	027	003	1	ff	ff	00ff	0000	55aa	55aa	0001	0001	08	08	0000
71	029	060	003	0	bf	bf	00ff	0000	55aa	55aa	0001	0001	08	08	0000
72	02a	029	003	0	bf	bf	00ff	0000	aa55	aa55	0001	0001	08	08	0000
73	026	02a	003	0	ff	ff	ffff	0000	aa55	aa55	0001	0001	08	08	0000
74	027	026	002	0	ff	ff	00ff	0000	aa55	aa55	0001	0001	09	09	0000
75	028	027	002	0	ff	ff	00ff	0000	aa55	aa55	0001	0001	09	09	0000
76	029	028	002	0	bf	bf	00ff	0000	aa55	aa55	0001	0001	09	09	0000
77	02a	029	002	0	bf	bf	00ff	0000	55aa	55aa	0001	0001	09	09	0000
78	026	02a	002	0	ff	ff	ffff	0000	55aa	55aa	0001	0001	09	09	0000
79	027	026	001	0	ff	ff	00ff	0000	55aa	55aa	0001	0001	0a	0a	0000
80	060	027	001	1	ff	ff	00ff	0000	55aa	55aa	0001	0001	0a	0a	0000
81	029	060	001	0	bf	bf	00ff	0000	55aa	55aa	0001	0001	0a	0a	0000
82	02a	029	001	0	bf	bf	00ff	0000	aa55	aa55	0001	0001	0a	0a	0000
83	026	02a	001	0	ff	ff	ffff	0000	aa55	aa55	0001	0001	0a	0a	0000
84	027	026	000	0	ff	ff	00ff	0000	aa55	aa55	0001	0001	0b	0b	0000
85	028	027	000	0	ff	ff	00ff	0000	aa55	aa55	0001	0001	0b	0b	0000
86	029	028	000	0	bf	bf	00ff	0000	aa55	aa55	0001	0001	0b	0b	0000
87	02a	029	000	0	bf	bf	00ff	0000	55aa	55aa	0001	0001	0b	0b	0000
88	02b	02a	000	0	ff	ff	ffff	0000	55aa	55aa	0001	0001	0b	0b	0000
89	02c	02b	000	0	ff	ff	ffff	0000	55aa	55aa	0001	0001	0c	0c	0000
90	02d	02c	000	0	ff	ff	ffff	0000	55aa	55aa	0002	0002	0c	0c	0000
91	02e	02d	000	1	ff	ff	ffff	0000	55aa	55aa	0002	0002	0c	0c	0000
92	02f	02e	000	1	ff	ff	ffff	0000	55aa	55aa	0002	0002	00	00	0000

Appendix 4. Simulation Results (Cont.)

93	030	02f	000	1	ff	ff	ffff	55aa	55aa	55aa	0000	0000	00	00	0000
94	031	030	001	1	ff	ff	ffff	55aa	55aa	55aa	0000	0000	00	00	0000
95	032	031	002	1	ff	ff	feff	55aa	55aa	55aa	0000	0000	00	00	0000
96	033	032	002	0	ef	ef	feff	55aa	55aa	55aa	0000	0000	00	00	0000
97	034	033	002	0	ef	ef	feff	55aa	55aa	55aa	0000	0000	00	00	55aa
98	063	034	002	1	ef	ef	ffff	55aa	55aa	55aa	0000	0000	00	00	55aa
99	036	063	002	0	ff	ff	ffff	55aa	55aa	55aa	0000	0000	00	00	55aa
100	037	036	002	0	ff	ff	ffff	55aa	55aa	55aa	0000	0000	00	00	55aa
101	031	037	002	1	ff	ff	ffff	55aa	55aa	55aa	0000	0000	00	00	55aa
102	032	031	001	0	ff	ff	feff	55aa	55aa	aa55	0000	0000	01	01	55aa
103	033	032	001	0	ef	ef	feff	aa55	55aa	aa55	0000	0000	01	01	55aa
104	034	033	001	0	ef	ef	feff	aa55	55aa	aa55	0000	0000	01	01	aa55
105	035	034	001	0	ef	ef	ffff	aa55	55aa	aa55	0000	0000	01	01	aa55
106	036	035	001	0	ff	ff	ffff	aa55	55aa	aa55	0000	0000	01	01	aa55
107	037	036	001	0	ff	ff	ffff	aa55	55aa	aa55	0000	0000	01	01	aa55
108	031	037	001	1	ff	ff	ffff	aa55	55aa	aa55	0000	0000	01	01	aa55
109	032	031	000	0	ff	ff	feff	aa55	55aa	55aa	0000	0000	02	02	aa55
110	033	032	000	0	ef	ef	feff	55aa	55aa	55aa	0000	0000	02	02	aa55
111	034	033	000	0	ef	ef	feff	55aa	55aa	55aa	0000	0000	02	02	55aa
112	063	034	000	1	ef	ef	ffff	55aa	55aa	55aa	0000	0000	02	02	55aa
113	036	063	000	0	ff	ff	ffff	55aa	55aa	55aa	0000	0000	02	02	55aa
114	037	036	000	0	ff	ff	ffff	55aa	55aa	55aa	0000	0000	02	02	55aa
115	038	037	000	1	ff	ff	ffff	55aa	55aa	55aa	0000	0000	02	02	55aa
116	039	038	000	0	ff	ff	ffff	55aa	55aa	55aa	0000	0000	03	03	55aa
117	030	039	001	0	ff	ff	ffff	55aa	55aa	55aa	0001	0001	03	03	55aa
118	031	030	000	1	ff	ff	ffff	55aa	55aa	55aa	0001	0001	03	03	55aa
119	032	031	002	1	ff	ff	feff	55aa	55aa	aa55	0001	0001	03	03	55aa
120	033	032	002	0	ef	ef	feff	aa55	55aa	aa55	0001	0001	03	03	55aa
121	034	033	002	0	ef	ef	feff	aa55	55aa	aa55	0001	0001	03	03	aa55
122	035	034	002	0	ef	ef	ffff	aa55	55aa	aa55	0001	0001	03	03	aa55
123	036	035	002	0	ff	ff	ffff	aa55	55aa	aa55	0001	0001	03	03	aa55
124	037	036	002	0	ff	ff	ffff	aa55	55aa	aa55	0001	0001	03	03	aa55
125	031	037	002	1	ff	ff	ffff	aa55	55aa	aa55	0001	0001	03	03	aa55
126	032	031	001	0	ff	ff	feff	aa55	55aa	55aa	0001	0001	04	04	aa55
127	033	032	001	0	ef	ef	feff	55aa	55aa	55aa	0001	0001	04	04	aa55
128	034	033	001	0	ef	ef	feff	55aa	55aa	55aa	0001	0001	04	04	55aa
129	063	034	001	1	ef	ef	ffff	55aa	55aa	55aa	0001	0001	04	04	55aa
130	036	063	001	0	ff	ff	ffff	55aa	55aa	55aa	0001	0001	04	04	55aa
131	037	036	001	0	ff	ff	ffff	55aa	55aa	55aa	0001	0001	04	04	55aa
132	031	037	001	1	ff	ff	ffff	55aa	55aa	55aa	0001	0001	04	04	55aa
133	032	031	000	0	ff	ff	feff	55aa	55aa	aa55	0001	0001	05	05	55aa
134	033	032	000	0	ef	ef	feff	aa55	55aa	aa55	0001	0001	05	05	55aa
135	034	033	000	0	ef	ef	feff	aa55	55aa	aa55	0001	0001	05	05	aa55
136	035	034	000	0	ef	ef	ffff	aa55	55aa	aa55	0001	0001	05	05	aa55
137	036	035	000	0	ff	ff	ffff	aa55	55aa	aa55	0001	0001	05	05	aa55
138	037	036	000	0	ff	ff	ffff	aa55	55aa	aa55	0001	0001	05	05	aa55
139	038	037	000	1	ff	ff	ffff	aa55	55aa	aa55	0001	0001	05	05	aa55
140	039	038	000	0	ff	ff	ffff	aa55	55aa	zzzz	0001	0001	06	06	aa55
141	03a	039	000	0	ff	ff	ffff	aa55	55aa	zzzz	0002	0002	06	06	aa55
142	03b	03a	000	1	ff	ff	ffff	aa55	55aa	55aa	0002	0002	06	06	aa55
143	03c	03b	000	1	ff	ff	ffff	55aa	55aa	55aa	0000	0000	06	06	aa55
144	03d	03c	008	1	ff	ff	ffff	55aa	55aa	55aa	0000	0000	06	06	aa55
145	03e	03d	008	1	ff	ff	ff00	55aa	55aa	55aa	0000	0000	00	00	aa55
146	061	03e	008	1	ff	ff	ff00	55aa	55aa	55aa	0000	0000	00	00	aa55
147	040	061	008	0	7f	7f	ff00	55aa	55aa	55aa	0000	0000	00	00	aa55

Appendix 4. Simulation Results (Cont.)

148	041	040	008	0	7f	7f	ff00	55aa	9966	9966	0000	0000	00	00	aa55
149	03d	041	008	0	ff	ff	ffff	55aa	9966	9966	0000	0000	00	00	aa55
150	03e	03d	007	0	ff	ff	ff00	55aa	9966	9966	0020	0020	00	00	aa55
151	03f	03e	007	0	ff	ff	ff00	55aa	9966	9966	0020	0020	00	00	aa55
152	040	03f	007	0	7f	7f	ff00	55aa	9966	9966	0020	0020	00	00	aa55
153	041	040	007	0	7f	7f	ff00	55aa	6699	6699	0020	0020	00	00	aa55
154	03d	041	007	0	ff	ff	ffff	55aa	6699	6699	0020	0020	00	00	aa55
155	03e	03d	006	0	ff	ff	ff00	55aa	6699	6699	0040	0040	00	00	aa55
156	061	03e	006	1	ff	ff	ff00	55aa	6699	6699	0040	0040	00	00	aa55
157	040	061	006	0	7f	7f	ff00	55aa	6699	6699	0040	0040	00	00	aa55
158	041	040	006	0	7f	7f	ff00	55aa	9966	9966	0040	0040	00	00	aa55
159	03d	041	006	0	ff	ff	ffff	55aa	9966	9966	0040	0040	00	00	aa55
160	03e	03d	005	0	ff	ff	ff00	55aa	9966	9966	0060	0060	00	00	aa55
161	03f	03e	005	0	ff	ff	ff00	55aa	9966	9966	0060	0060	00	00	aa55
162	040	03f	005	0	7f	7f	ff00	55aa	9966	9966	0060	0060	00	00	aa55
163	041	040	005	0	7f	7f	ff00	55aa	6699	6699	0060	0060	00	00	aa55
164	03d	041	005	0	ff	ff	ffff	55aa	6699	6699	0060	0060	00	00	aa55
165	03e	03d	004	0	ff	ff	ff00	55aa	6699	6699	0080	0080	00	00	aa55
166	061	03e	004	1	ff	ff	ff00	55aa	6699	6699	0080	0080	00	00	aa55
167	040	061	004	0	7f	7f	ff00	55aa	6699	6699	0080	0080	00	00	aa55
168	041	040	004	0	7f	7f	ff00	55aa	9966	9966	0080	0080	00	00	aa55
169	03d	041	004	0	ff	ff	ffff	55aa	9966	9966	0080	0080	00	00	aa55
170	03e	03d	003	0	ff	ff	ff00	55aa	9966	9966	00a0	00a0	00	00	aa55
171	03f	03e	003	0	ff	ff	ff00	55aa	9966	9966	00a0	00a0	00	00	aa55
172	040	03f	003	0	7f	7f	ff00	55aa	9966	9966	00a0	00a0	00	00	aa55
173	041	040	003	0	7f	7f	ff00	55aa	6699	6699	00a0	00a0	00	00	aa55
174	03d	041	003	0	ff	ff	ffff	55aa	6699	6699	00a0	00a0	00	00	aa55
175	03e	03d	002	0	ff	ff	ff00	55aa	6699	6699	00c0	00c0	00	00	aa55
176	061	03e	002	1	ff	ff	ff00	55aa	6699	6699	00c0	00c0	00	00	aa55
177	040	061	002	0	7f	7f	ff00	55aa	6699	6699	00c0	00c0	00	00	aa55
178	041	040	002	0	7f	7f	ff00	55aa	9966	9966	00c0	00c0	00	00	aa55
179	03d	041	002	0	ff	ff	ffff	55aa	9966	9966	00c0	00c0	00	00	aa55
180	03e	03d	001	0	ff	ff	ff00	55aa	9966	9966	00e0	00e0	00	00	aa55
181	03f	03e	001	0	ff	ff	ff00	55aa	9966	9966	00e0	00e0	00	00	aa55
182	040	03f	001	0	7f	7f	ff00	55aa	9966	9966	00e0	00e0	00	00	aa55
183	041	040	001	0	7f	7f	ff00	55aa	6699	6699	00e0	00e0	00	00	aa55
184	03d	041	001	0	ff	ff	ffff	55aa	6699	6699	00e0	00e0	00	00	aa55
185	03e	03d	000	0	ff	ff	ff00	55aa	6699	6699	0100	0100	00	00	aa55
186	061	03e	000	1	ff	ff	ff00	55aa	6699	6699	0100	0100	00	00	aa55
187	040	061	000	0	7f	7f	ff00	55aa	6699	6699	0100	0100	00	00	aa55
188	041	040	000	0	7f	7f	ff00	55aa	9966	9966	0100	0100	00	00	aa55
189	042	041	000	0	ff	ff	ffff	55aa	9966	9966	0100	0100	00	00	aa55
190	043	042	000	0	ff	ff	ffff	55aa	9966	zzzz	0120	0120	00	00	aa55
191	044	043	000	1	ff	ff	ffff	55aa	9966	zzzz	0120	0120	00	00	aa55
192	045	044	000	1	ff	ff	ffff	9966	9966	zzzz	0000	0000	00	00	aa55
193	046	045	003	1	ff	ff	fffe	9966	9966	6699	0000	0000	00	00	aa55
194	047	046	003	0	df	df	fffe	6699	9966	6699	0000	0000	00	00	aa55
195	048	047	003	0	df	df	fffe	6699	9966	6699	0000	0000	00	00	6699
196	064	048	003	1	df	df	ffff	6699	9966	6699	0000	0000	00	00	6699
197	04a	064	003	0	ff	ff	ffff	6699	9966	6699	0000	0000	00	00	6699
198	04b	04a	003	0	ff	ff	ffff	6699	9966	6699	0000	0000	00	00	6699
199	045	04b	003	1	ff	ff	ffff	6699	9966	6699	0000	0000	00	00	6699
200	046	045	002	0	ff	ff	fffe	6699	9966	9966	0020	0020	00	00	6699
201	047	046	002	0	df	df	fffe	9966	9966	9966	0020	0020	00	00	6699
202	048	047	002	0	df	df	fffe	9966	9966	9966	0020	0020	00	00	9966

Appendix 4. Simulation Results (Cont.)

203	049	048	002	0	df	df	ffff	9966	9966	9966	0020	0020	00	00	9966
204	04a	049	002	0	ff	ff	ffff	9966	9966	9966	0020	0020	00	00	9966
205	04b	04a	002	0	ff	ff	ffff	9966	9966	9966	0020	0020	00	00	9966
206	045	04b	002	1	ff	ff	ffff	9966	9966	9966	0020	0020	00	00	9966
207	046	045	001	0	ff	ff	fffe	9966	9966	6699	0040	0040	00	00	9966
208	047	046	001	0	df	df	fffe	6699	9966	6699	0040	0040	00	00	9966
209	048	047	001	0	df	df	fffe	6699	9966	6699	0040	0040	00	00	6699
210	064	048	001	1	df	df	ffff	6699	9966	6699	0040	0040	00	00	6699
211	04a	064	001	0	ff	ff	ffff	6699	9966	6699	0040	0040	00	00	6699
212	04b	04a	001	0	ff	ff	ffff	6699	9966	6699	0040	0040	00	00	6699
213	045	04b	001	1	ff	ff	ffff	6699	9966	6699	0040	0040	00	00	6699
214	046	045	000	0	ff	ff	fffe	6699	9966	9966	0060	0060	00	00	6699
215	047	046	000	0	df	df	fffe	9966	9966	9966	0060	0060	00	00	6699
216	048	047	000	0	df	df	fffe	9966	9966	9966	0060	0060	00	00	9966
217	049	048	000	0	df	df	ffff	9966	9966	9966	0060	0060	00	00	9966
218	04a	049	000	0	ff	ff	ffff	9966	9966	9966	0060	0060	00	00	9966
219	04b	04a	000	0	ff	ff	ffff	9966	9966	9966	0060	0060	00	00	9966
220	04c	04b	000	1	ff	ff	ffff	9966	9966	9966	0060	0060	00	00	9966
221	04d	04c	000	0	ff	ff	ffff	9966	9966	9966	0080	0080	00	00	9966
222	04e	04d	000	1	ff	ff	ffff	9966	9966	9966	0000	0000	00	00	9966
223	04f	04e	000	1	ff	ff	ffff	9966	9966	9966	0000	0000	00	00	9966
224	050	04f	001	1	ff	ff	ffff	9966	9966	9966	0000	0000	00	00	9966
225	051	050	002	1	ff	ff	feff	9966	9966	UUUU	0000	0000	00	00	9966
226	062	051	002	1	ff	ff	feff	9966	9966	9966	0000	0000	00	00	9966
227	053	062	002	0	bf	bf	feff	9966	9966	9966	0000	0000	00	00	9966
228	054	053	002	0	bf	bf	feff	9966	cc33	cc33	0000	0000	00	00	9966
229	055	054	002	0	ff	ff	ffff	9966	cc33	ZZZZ	0000	0000	00	00	9966
230	056	055	002	1	ff	ff	ffff	9966	cc33	ZZZZ	0000	0000	00	00	9966
231	057	056	002	1	ff	ff	fffe	cc33	cc33	33cc	0000	0000	00	00	9966
232	058	057	002	0	df	df	fffe	33cc	cc33	33cc	0000	0000	00	00	9966
233	059	058	002	0	df	df	fffe	33cc	cc33	33cc	0000	0000	00	00	33cc
234	065	059	002	1	df	df	ffff	33cc	cc33	33cc	0000	0000	00	00	33cc
235	05b	065	002	0	ff	ff	ffff	33cc	cc33	33cc	0000	0000	00	00	33cc
236	05c	05b	002	0	ff	ff	ffff	33cc	cc33	33cc	0000	0000	00	00	33cc
237	050	05c	002	0	ff	ff	ffff	33cc	cc33	33cc	0020	0020	00	00	33cc
238	051	050	001	0	ff	ff	feff	33cc	cc33	33cc	0020	0020	01	01	33cc
239	052	051	001	0	ff	ff	feff	33cc	cc33	ZZZZ	0020	0020	01	01	33cc
240	053	052	001	0	bf	bf	feff	33cc	cc33	ZZZZ	0020	0020	01	01	33cc
241	054	053	001	0	bf	bf	feff	33cc	33cc	ZZZZ	0020	0020	01	01	33cc
242	055	054	001	0	ff	ff	ffff	33cc	33cc	ZZZZ	0020	0020	01	01	33cc
243	056	055	001	1	ff	ff	ffff	33cc	33cc	ZZZZ	0020	0020	01	01	33cc
244	057	056	001	1	ff	ff	fffe	33cc	33cc	cc33	0020	0020	01	01	33cc
245	058	057	001	0	df	df	fffe	cc33	33cc	cc33	0020	0020	01	01	33cc
246	059	058	001	0	df	df	fffe	cc33	33cc	cc33	0020	0020	01	01	cc33
247	05a	059	001	0	df	df	ffff	cc33	33cc	cc33	0020	0020	01	01	cc33
248	05b	05a	001	0	ff	ff	ffff	cc33	33cc	cc33	0020	0020	01	01	cc33
249	05c	05b	001	0	ff	ff	ffff	cc33	33cc	cc33	0020	0020	01	01	cc33
250	050	05c	001	0	ff	ff	ffff	cc33	33cc	cc33	0040	0040	01	01	cc33
251	051	050	000	0	ff	ff	feff	cc33	33cc	cc33	0040	0040	02	02	cc33
252	062	051	000	1	ff	ff	feff	cc33	33cc	ZZZZ	0040	0040	02	02	cc33
253	053	062	000	0	bf	bf	feff	cc33	33cc	ZZZZ	0040	0040	02	02	cc33
254	054	053	000	0	bf	bf	feff	cc33	cc33	ZZZZ	0040	0040	02	02	cc33
255	055	054	000	0	ff	ff	ffff	cc33	cc33	ZZZZ	0040	0040	02	02	cc33
256	056	055	000	1	ff	ff	ffff	cc33	cc33	ZZZZ	0040	0040	02	02	cc33
257	057	056	000	1	ff	ff	fffe	cc33	cc33	33cc	0040	0040	02	02	cc33

Appendix 4. Simulation Results (Cont.)

258	058	057	000	0	df	df	fffe	33cc	cc33	33cc	0040	0040	02	02	cc33
259	059	058	000	0	df	df	fffe	33cc	cc33	33cc	0040	0040	02	02	33cc
260	065	059	000	1	df	df	ffff	33cc	cc33	33cc	0040	0040	02	02	33cc
261	05b	065	000	0	ff	ff	ffff	33cc	cc33	33cc	0040	0040	02	02	33cc
262	05c	05b	000	0	ff	ff	ffff	33cc	cc33	33cc	0040	0040	02	02	33cc
263	05d	05c	000	0	ff	ff	ffff	33cc	cc33	33cc	0060	0060	02	02	33cc
264	05e	05d	000	0	ff	ff	ffff	33cc	cc33	33cc	0060	0060	03	03	33cc
265	04f	05e	001	0	ff	ff	ffff	33cc	cc33	33cc	0061	0061	03	03	33cc
266	050	04f	000	1	ff	ff	ffff	33cc	cc33	33cc	0061	0061	03	03	33cc
267	051	050	002	1	ff	ff	feff	33cc	cc33	33cc	0061	0061	03	03	33cc
268	052	051	002	0	ff	ff	feff	33cc	cc33	zzzz	0061	0061	03	03	33cc
269	053	052	002	0	bf	bf	feff	33cc	cc33	zzzz	0061	0061	03	03	33cc
270	054	053	002	0	bf	bf	feff	33cc	33cc	zzzz	0061	0061	03	03	33cc
271	055	054	002	0	ff	ff	ffff	33cc	33cc	zzzz	0061	0061	03	03	33cc
272	056	055	002	1	ff	ff	ffff	33cc	33cc	zzzz	0061	0061	03	03	33cc
273	057	056	002	1	ff	ff	fffe	33cc	33cc	cc33	0061	0061	03	03	33cc
274	058	057	002	0	df	df	fffe	cc33	33cc	cc33	0061	0061	03	03	33cc
275	059	058	002	0	df	df	fffe	cc33	33cc	cc33	0061	0061	03	03	cc33
276	05a	059	002	0	df	df	ffff	cc33	33cc	cc33	0061	0061	03	03	cc33
277	05b	05a	002	0	ff	ff	ffff	cc33	33cc	cc33	0061	0061	03	03	cc33
278	05c	05b	002	0	ff	ff	ffff	cc33	33cc	cc33	0061	0061	03	03	cc33
279	050	05c	002	0	ff	ff	ffff	cc33	33cc	cc33	0081	0081	03	03	cc33
280	051	050	001	0	ff	ff	feff	cc33	33cc	cc33	0081	0081	04	04	cc33
281	062	051	001	1	ff	ff	feff	cc33	33cc	zzzz	0081	0081	04	04	cc33
282	053	062	001	0	bf	bf	feff	cc33	33cc	zzzz	0081	0081	04	04	cc33
283	054	053	001	0	bf	bf	feff	cc33	cc33	zzzz	0081	0081	04	04	cc33
284	055	054	001	0	ff	ff	ffff	cc33	cc33	zzzz	0081	0081	04	04	cc33
285	056	055	001	1	ff	ff	ffff	cc33	cc33	zzzz	0081	0081	04	04	cc33
286	057	056	001	1	ff	ff	fffe	cc33	cc33	33cd	0081	0081	04	04	cc33
287	058	057	001	0	df	df	fffe	33cd	cc33	33cd	0081	0081	04	04	cc33
288	059	058	001	0	df	df	fffe	33cd	cc33	33cd	0081	0081	04	04	33cd
289	065	059	001	1	df	df	ffff	33cd	cc33	33cd	0081	0081	04	04	33cd
290	05b	065	001	0	ff	ff	ffff	33cd	cc33	33cd	0081	0081	04	04	33cd
291	05c	05b	001	0	ff	ff	ffff	33cd	cc33	33cd	0081	0081	04	04	33cd
292	050	05c	001	0	ff	ff	ffff	33cd	cc33	33cd	00a1	00a1	04	04	33cd
293	051	050	000	0	ff	ff	feff	33cd	cc33	33cd	00a1	00a1	05	05	33cd
294	052	051	000	0	ff	ff	feff	33cd	cc33	zzzz	00a1	00a1	05	05	33cd
295	053	052	000	0	bf	bf	feff	33cd	cc33	zzzz	00a1	00a1	05	05	33cd
296	054	053	000	0	bf	bf	feff	33cd	33cc	zzzz	00a1	00a1	05	05	33cd
297	055	054	000	0	ff	ff	ffff	33cd	33cc	zzzz	00a1	00a1	05	05	33cd
298	056	055	000	1	ff	ff	ffff	33cd	33cc	zzzz	00a1	00a1	05	05	33cd
299	057	056	000	1	ff	ff	fffe	33cd	33cc	cc33	00a1	00a1	05	05	33cd
300	058	057	000	0	df	df	fffe	cc33	33cc	cc33	00a1	00a1	05	05	33cd
301	059	058	000	0	df	df	fffe	cc33	33cc	cc33	00a1	00a1	05	05	cc33
302	05a	059	000	0	df	df	ffff	cc33	33cc	cc33	00a1	00a1	05	05	cc33
303	05b	05a	000	0	ff	ff	ffff	cc33	33cc	cc33	00a1	00a1	05	05	cc33
304	05c	05b	000	0	ff	ff	ffff	cc33	33cc	cc33	00a1	00a1	05	05	cc33
305	05d	05c	000	0	ff	ff	ffff	cc33	33cc	cc33	00c1	00c1	05	05	cc33
306	05e	05d	000	0	ff	ff	ffff	cc33	33cc	cc33	00c1	00c1	06	06	cc33
307	05f	05e	000	0	ff	ff	ffff	cc33	33cc	cc33	00c2	00c2	06	06	cc33
308	05f	05f	000	1	ff	ff	ffff	cc33	33cc	cc33	00c2	00c2	06	06	cc33
309	05f	05f	000	1	ff	ff	ffff	cc33	33cc	cc33	00c2	00c2	06	06	cc33
310	05f	05f	000	1	ff	ff	ffff	cc33	33cc	cc33	00c2	00c2	06	06	cc33
311	05f	05f	000	1	ff	ff	ffff	cc33	33cc	cc33	00c2	00c2	06	06	cc33
312	05f	05f	000	1	ff	ff	ffff	cc33	33cc	cc33	00c2	00c2	06	06	cc33

Appendix 4. Simulation Results (Cont.)

```
313  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
314  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
315  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
316  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
317  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
318  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
319  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
320  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
321  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
322  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
323  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
324  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
325  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
326  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
327  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
328  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
329  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
330  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
331  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
332  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
333  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
334  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
335  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
336  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
337  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
338  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
339  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
340  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
341  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
342  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
343  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
344  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
345  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
346  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
347  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
348  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
349  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
350  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
351  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
352  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
353  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
354  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
355  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
356  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
357  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
358  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
359  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
360  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
361  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
362  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
363  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
364  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
365  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
366  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
367  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
```

Appendix 4. Simulation Results (Cont.)

```
368  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
369  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
370  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
371  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
372  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
373  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
374  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
375  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
376  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
377  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
378  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
379  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
380  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
381  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
382  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
383  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
384  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
385  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
386  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
387  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
388  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
389  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
390  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
391  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
392  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
393  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
394  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
395  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
396  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
397  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
398  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
399  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
400  05f 05f 000 1 ff ff ffff cc33 33cc cc33 00c2 00c2 06 06 cc33
```

Development Systems

Section Index

Development Systems

For additional information,
call 800-TEAM-WSI (800-832-6974).
In California, Call 800-562-6363.

Programmable Peripheral
Electronic Bulletin Board

Bulletin Board

WSI provides a 24-hour electronic bulletin board system that provides the user with the latest information on software updates, enhancements, and applications relating to WSI products. In addition, users developing applications software for WSI products can send portions of their code to WSI for application's consultation if desired.

The following hardware is required to use the WSI bulletin board:

- ❑ Computer Terminal
- ❑ 300, 1200, 2400 Baud Modem
- ❑ 8 Data Bits
- ❑ No Parity
- ❑ 1 Stop Bit

Access Line

To access the bulletin board, dial

(510) 498-1002

and wait for the modem tone. When your modem establishes a connection, enter <return> <return> to signal the bulletin board software. The board should respond:

WSI Customer Engineering Support
Electronic Bulletin Board Service

followed by some other messages, after which you will be asked for your name, and a password. Upon initial use, follow the on-screen prompts for establishing your password.

Now that you have entered the bulletin board service, you will be given a choice of "MAIN" commands:

Main Commands

M)sg-Section
Choose this option to leave messages.

F)ile-Section
Choose this option to download or upload data files and/or utility programs

B)ulletins
Choose this option to see the latest important news such as software versions and programming tips for WSI Memory and PSD products.

S)tatistics
This option describes the current bulletin board statistics

C)hange
Choose this option to change operational settings that the bulletin board maintains for your user name.

P)age-Operator
Choose this to page the operator for assistance. It is not likely that the operator will be available during West Coast U.S. non-business hours.

L)ist-Callers
Choose this option to see who else is using the board at this moment.

A)ns-Questionnaire
Choose this option to answer a user profile questionnaire.

V)ersion
Describes the board software version.

G)oodbye
Choose this to leave the bulletin board.

See the individual software manuals for more detailed explanation and usage of the bulletin board.

Programmable Peripheral
PAC1000-Gold/Silver
Development System

Description

PAC1000-GOLD/PAC1000-SILVER is a complete set of IBM-PC-based development tools. They provide the integrated easy-to-use environment to support the PAC1000 program development and device programming.

The tools run on an IBM-PC XT, AT or compatible computer running MS-DOS version 3.1 or later.

PACSEL

PACSEL is the PAC1000 system entry language. It has the following features:

- ❑ Enables specification of up to three parallel operations:
  - — Program control operation
  - — CPU operation
  - — Out Control operation

  General Syntax:
  Label: Program Control, CPU, Out Control;
- ❑ Enables mixing of three source language types in one instruction:
  - — High Level Language
  - — Assembler
  - — Microcode
- ❑ Specific instructions support unique PAC1000 architecture features available in all three source language types.
- ❑ Links unlimited amounts of modules.

PACSIM

PACSIM is a functional simulator and software debugger. It has the following features:

- ❑ Clock driven functional simulator.
- ❑ Provides trace capabilities on internal states (Registers, Flags, Pins and more).
- ❑ Provides breakpoint capabilities on any internal state of the PAC1000.
- ❑ Supports batch mode simulation.
- ❑ Provides waveform analysis.
- ❑ On-line HELP available at any level.

5

PACPRO

PACPRO is the interface software that enables the user to program a PAC1000 microcontroller on the WS6000 MagicPro™ programmer. The PACPRO enables the user to load the program into the programmer and to execute the following operations:

- ❑ Help
- ❑ Upload RAM from PAC
- ❑ Load RAM from disk
- ❑ Write RAM to FILE
- ❑ Display PAC data
- ❑ Blank test PAC
- ❑ Verify PAC
- ❑ Program PAC
- ❑ Configuration
- ❑ Quit

IMPACT

IMPACT is the interface manager to the PAC1000 tools. IMPACT enables the user to access PACSEL, PACSIM, PACPRO, DOS and an editor with a menu driven interface. File specification can be done without extension enabling the user to use the same name throughout the design. A HELP window is available on-line giving information on all the needed steps at each level.

WS6000 MagicPro™ Programmer

MagicPro is an engineering development tool designed to program all WSI programmable products (EPROMs, RPROMs, PAC1000, MAP168, PSD3XX Family and SAM448). It is used within the IBM-PC and compatible environment. The MagicPro consists of a short plug-in board and a Remote Socket Adaptor (RSA). It occupies a short expansion slot in the PC. The RSA has two ZIF-DIP sockets that will support WSI's 24, 28, 32 and 40 pin standard 600 mil or slim 300 mil DIP packages without adaptors. Other packages are supported using adaptors.

WS6010 Socket Adaptor

The WS6010 is a socket adaptor that mounts on the MagicPro RSA and adapts the PAC1000 in an 88-pin CPGA package to the programmer.

WS6013 Socket Adaptor

The WS6013 is a socket adaptor that mounts on the MagicPro RSA and adapts the PAC1000 in a 100-pin QFP package to the programmer.

WSI-Support

WSI provides on-going support for users of PAC1000-GOLD/PAC1000-SILVER. For the first year, software and programmer updates are included at no charge. After that, the user may purchase the WSI-Support agreement to continue to receive the latest software releases.

Ordering Information

Product	Description
PAC1000-GOLD	Contains PAC1000-Silver, WS6000 MagicPro Programmer, Two Product Samples and Matching Package Adaptor Socket, WSI-Support
PAC1000-SILVER	Contains PAC1000 Software (PACSEL, PACSIM, PACPRO, and IMPACT), Software User's Manual, WSI-Support.
WSI-Support	12-Month Software Update Service, Access to WSI's 24-Hour Electronic Bulletin Board, and Hotline to WSI System Application Experts.

PAC1000-GOLD

Contents

- ❑ PACSEL
  System design entry language and program linker.
- ❑ PACSIM
  Functional simulator and software debugger.
- ❑ PACPRO
  Interface software to PAC1000 device programmer (MagicPro™).
- ❑ IMPACT
  Interface manager for PAC1000 embedded controller development tools.
- ❑ Software user's manual.
- ❑ WSI-SUPPORT agreement.
- ❑ WS6000 MagicPro Programmer.
- ❑ Two product samples and matching package adaptor socket.

5

PAC1000-SILVER

Contents

- ❑ PACSEL
  System design entry language and program linker.
- ❑ PACSIM
  Functional simulator and software debugger.
- ❑ PACPRO
  Interface software to PAC1000 device programmer (MagicPro™).
- ❑ IMPACT
  Interface manager for PAC1000 embedded controller development tools.
- ❑ Software user's manual.
- ❑ WSI-SUPPORT agreement.

WS6000

MagicPro™ Memory and Programmable Peripheral Programmer

Key Features

- ❑ Programs All WSI CMOS Memory and and Programmable Peripheral Products and All Future Programmable Products
- ❑ Programs 24, 28, 32 and 40 Pin Standard 600 Mil or Slim 300 Mil Dip Packages without Adaptors
- ❑ Programs LCC, PGA and QFP Packaged Product by Using Adaptors
- ❑ Easy-to-Use Menu-Driven Software
- ❑ Compatible with IBM PC/XT/AT Family of Computers (and True Plug-Compatible

General Description

MagicPro is an engineering development tool designed to program existing WSI EPROMs, RPROMs, Programmable Peripherals, and future WSI programmable products. It is used within the IBM-PC® and compatible computers. The MagicPro is meant to bridge the gap betweeen the introduction of a new WSI programmable product and the availability of programming support from programmer manufacturers (e.g., Data I/O, etc.). The MagicPro programmer and accompanying software enable quick programming of newly released WSI programmable products, thus accelerating the system design process.

The MagicPro plug-in board is integrated easily into the IBM-PC. It occupies a short expansion slot and its software requires only 256K bytes of computer memory. The two external ZIF-Dip sockets in the Remote Socket Adaptor (RSA) support 24, 28, 32 and 40 pin standard 600 mil or slim 300 mil Dip packages without adaptors. LCC, PGA and QFP packages are supported using adaptors.

Many features of the MagicPro Programmer show its capabilities in supporting WSI's future products. Some of these are:

- ❑ 24 to 40 pin JEDEC Dip Pinouts
- ❑ 1 Meg Address Space (20 address lines)
- ❑ 16 Data I/O Lines

5

General Description (Cont.)

The MagicPro menu driven software makes using different features of the MagicPro an easy task. Software updates are done via floppy disk which eliminates the need for adding a new memory device for system upgrading.

Please call 800-TEAM-WSI for information regarding programming WSI products not listed herein. The MagicPro reads Intel Hex format for use with assemblers and compilers.

MagicPro Commands

- ❑ Help
- ❑ Upload RAM from Device
- ❑ Load RAM from Disk
- ❑ Write RAM to Disk
- ❑ Display RAM Data
- ❑ Edit RAM
- ❑ Move/Copy RAM
- ❑ Fill RAM
- ❑ Blank Test Device
- ❑ Verify Device
- ❑ Program Device
- ❑ Select Device
- ❑ Configuration
- ❑ Quit MagicPro

Technical Information

- ❑ ***Size:***
  IBM-PC Short Length Card
- ❑ ***Port Address Location:***
  100H to 1FFH – default 140H (if a conflict exists with this address space, the address location can be changed in software and with the switches on the plug-in board.)
- ❑ ***System Memory Requirements:***
  256K Bytes of RAM
- ❑ ***Power:***
  + 5 Volts, 0.03 Amp; +12 Volts, 0.04 Amp
- ❑ ***Remote Socket Adaptor (RSA):***
  The RSA contains two ZIF-Dip sockets that are used to program and read WSI programmable products. The 32 pin ZIF-Dip socket supports 24, 28 and 32 pin standard 600 mil or slim 300 mil Dip packaged product. The 40 pin ZIF-Dip socket supports all 40 pin Dip packages. Adaptor sockets are available for LCC, PGA and QFP packages.

Ordering Information

The WS6000 MagicPro Systems Contains:

- ❑ MagicPro IBM-PC Plug-in Programmer Board
- ❑ MagicPro Remote Socket Adaptor and Cable
- ❑ MagicPro Operating System Floppy Disk and Operating Manual

The WS6000 MagicPro Adaptors Include:

- ❑ WS6001 28-Pin CLLCC Package Adaptor for Memory.
- ❑ WS6008 28-Pin 0.3" Wide Dip Adaptor for SAM448
- ❑ WS6009 28-Pin PLDCC/CLDCC/ CLLCC Package Adaptor for SAM448
- ❑ WS6010 88-Pin PGA Package Adaptor for PAC1000
- ❑ WS6012 32-Pin CLDCC Package Adaptor for Memory
- ❑ WS6013 100-Pin QFP Package Adaptor for PAC1000
- ❑ WS6014 44-Pin CLDCC/PLDCC Package Adaptor for MAP168
- ❑ WS6015 44-Pin PGA Package Adaptor for MAP168 and PSD3XX
- ❑ WS6016 44-Pin CLDCC/PLDCC Package Adaptor for Memory
- ❑ WS6020 52-Pin PQFP Package Adaptor for PSD3XX
- ❑ WS6021 44-Pin CLDCC/PLDCC Package Adaptor for PSD3XX

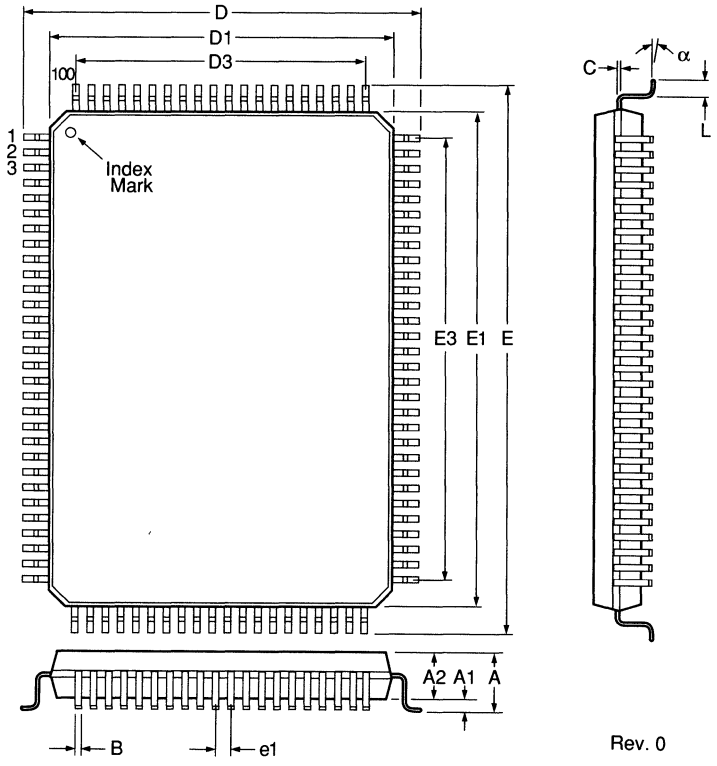
Package Information

6

Section Index

For additional information,
call 800-TEAM-WSI (800-832-6974).
In California, Call 800-562-6363.

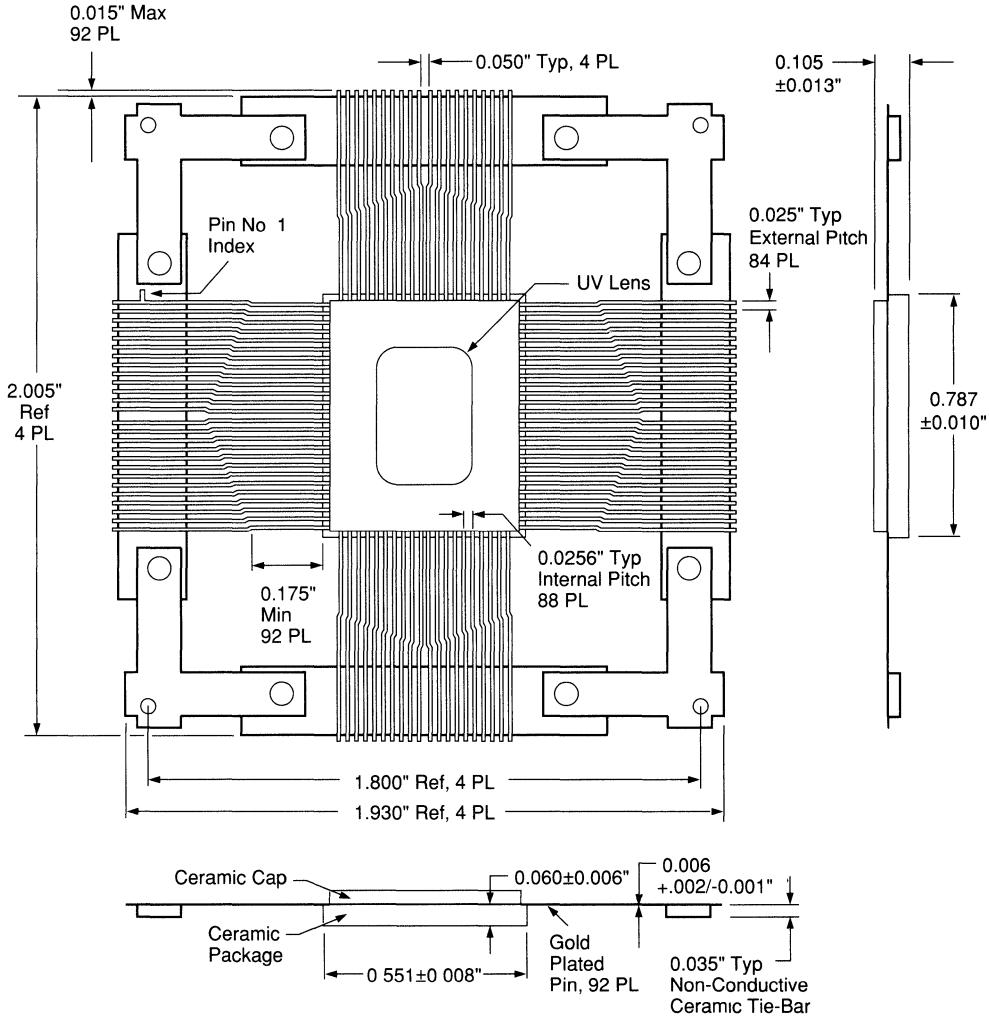
Drawing Q1
100 Pin Plastic
Quad Flatpack,
Gull Wing,
Fine Pitch (PQFP)



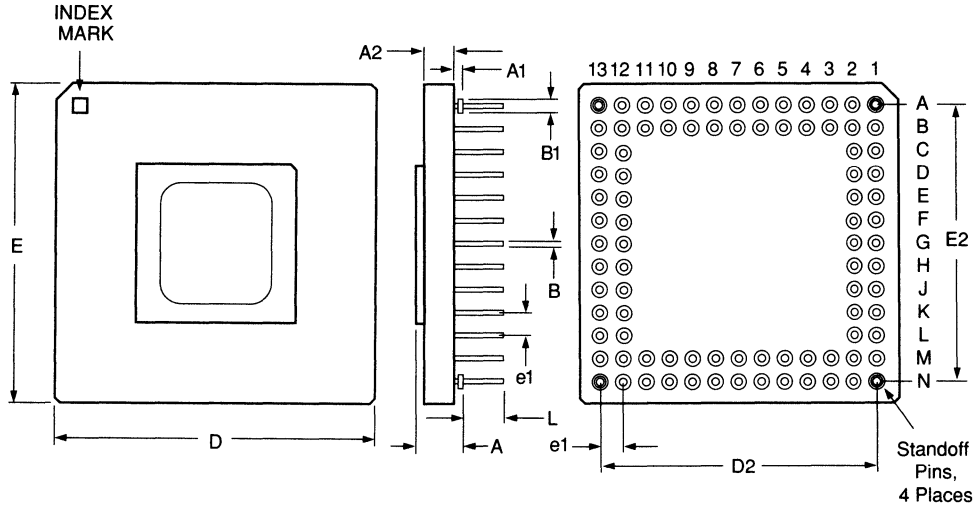
Rev. 0

Family: Plastic Quad Flatpack						
	Millimeters			Inches		
Symbol	Min	Max	Notes	Min	Max	Notes
a	0°	8°		0°	8°	
A	-	3.40		-	0.134	
A1	0.00	0.25		0.010	-	
A2	2.57	2.87		0.101	0.113	
B	0.22	0.38		0.009	0.015	
C	0.13	0.23		0.005	0.009	
D	16.95	17.45		0.667	0.687	
D1	13.90	14.10		0.547	0.555	
D3	12.35		Reference	0.486		Reference
E	22.95	23.45		0.904	0.923	
E1	19.90	20.10		0.783	0.791	
E3	18.85		Reference	0.472		Reference
e1	0.65		Reference	0.026		Reference
L	0.60	0.95		0.024	0.037	
N	100			100		

Drawing V1 92 Pin Ceramic Quad Flatpack (CQFP)



Drawing X1 88 Pin Ceramic PGA



X1b Rev 2

Family: Ceramic Pin Grid Array Package						
	Millimeters			Inches		
Symbol	Min	Max	Notes	Min	Max	Notes
A	3.30	4.83		0.130	0.190	
A1	1.02		Typical	0.040		Typical
A2	2.41	3.43		0.095	0.135	
B	0.41	0.51	Diameter	0.016	0.020	Diameter
B1	1.02		Typical Dia.	0.040		Typical Dia.
D	32.51	33.91		1.280	1.335	
D2	30.48		Reference	1.200		Reference
E	32.51	33.91		1.280	1.335	
E2	30.48		Reference	1.200		Reference
e1	2.54		Reference	0.100		Reference
L	3.30	4.32		0.130	0.170	
N	88			88		

X1b

6

Sales Representatives and Distributors

Section Index

For additional information,
Call 800-TEAM-WSI (800-832-6974).
In California, Call 800-562-6363

Sales Representatives and Distributors

Domestic Representatives

ALABAMA

Rep Inc
Huntsville
Tel (205) 881-9270
Fax (205) 882-6692

ARIZONA

Summit Sales
Scottsdale
Tel (602) 998-4850
Fax (602) 998-5274

CALIFORNIA

Bager Electronics, Inc
Fountain Valley
Tel (714) 957-3367
Fax (714) 546-2654

Bager Electronics, Inc
Woodland Hills
Tel (818) 712-0011
Fax (818) 712-0160

Earle Assoc , Inc
San Diego
Tel· (619) 278-5441
Fax (619) 278-5443

I Squared, Inc
Santa Clara
Tel (408) 988-3400
Fax (408) 988-2079

CANADA

Intelatech, Inc
Mississauga
Tel. (416) 629-0082
Fax. (416) 629-1795

COLORADO

Waugaman Associates, Inc
Wheat Ridge
Tel (303) 423-1020
Fax (303) 467-3095

CONNECTICUT

Advanced Tech Sales
Wallingford
Tel (203) 284-0838
Fax (203) 284-8232

FLORIDA

QXI of Florida, Inc
Fort Lauderdale
Tel (305) 978-0120
Fax. (305) 972-1408

QXI of Florida, Inc
Orlando
Tel. (407) 872-2321
Fax (407) 321-2098

QXI of Florida, Inc
St Petersburg
Tel. (813) 894-4556
Fax· (813) 894-3989

GEORGIA

Rep Inc
Tucker
Tel (404) 938-4358
Tax (404) 938-0194

ILLINOIS

Victory Sales
Hoffman Estates
Tel (708) 490-0300
Telex 206248
Fax (708) 490-1499

INDIANA

Giesting & Associates
Carmel
Tel (317) 844-5222
Fax (317) 844-5861

IOWA

Gassner & Clark Co
Cedar Rapids
Tel (319) 393-5763
Twx 62950087
Fax (319) 393-5799

KANSAS/NEBRASKA

C Logsdon & Assoc
Prairie Village
Tel (913) 381-3833
Fax· (913) 381-9774

KENTUCKY

Giesting & Associates
Versailles
Tel (606) 873-2330
Fax (606) 873-6233

MARYLAND/VIRGINIA

New Era Sales, Inc
Severna Park
Tel (410) 544-4100
Fax (410) 544-6092

MASSACHUSETTS

Advanced Tech Sales, Inc
North Reading
Tel (508) 664-0888
Fax (508) 664-5503

MICHIGAN

Giesting & Associates
Comstock Park
Tel (616) 784-9437
Fax (616) 784-9438

Giesting & Associates
Livonia
Tel (313) 478-8106
Fax (313) 477-6908

MINNESOTA

OHMS Technology, Inc
Edina
Tel (612) 932-2920
Fax (612) 932-2918

MISSOURI

John G Macke Company
St Louis
Tel (314) 432-2830
Fax (314) 432-1456

NEW JERSEY

Metro Logic Corp
(AT&T only)
Fairfield
Tel (201) 575-5585
Fax (201) 575-8023

Strategic Sales, Inc
Teaneck
Tel (201) 833-0099
Fax (201) 833-0061

S J Associates, Inc
Mt Laurel, NJ 08084
Tel (609) 866-1234
Fax (609) 866-8627

NEW MEXICO

S & S Technologies
Albuquerque
Tel (505) 298-7177
Fax (505) 298-2004

NEW YORK

Strategic Sales, Inc
New York City
Tel (201) 833-0099
Fax (201) 833-0061

Tri-Tech Electronics, Inc
East Rochester
Tel (716) 385-6500
Twx 62934993
Fax (716) 385-7655

Tri-Tech Electronics, Inc
Fayetteville
Tel (315) 446-2881
Twx 7105410604
Fax (315) 446-3047

Tri-Tech Electronics, Inc
Fishkill
Tel (914) 897-5611
Twx 62906505
Fax (914) 897-5611

NORTH CAROLINA

Rep, Inc
Morrisville
Tel (919) 469-9997
Fax (919) 481-3879

OHIO

Giesting & Associates
Cincinnati
Tel: (513) 385-1105
Fax (513) 385-5069

Giesting & Associates
Cleveland
Tel. (216) 261-9705
Fax (216) 261-5624

Giesting & Associates
Columbus
Tel (614) 459-4800
Fax (614) 459-4801

OKLAHOMA

West Associates
Tulsa
Tel (918) 665-3465
Fax (918) 663-1762

OREGON

Thorson Company
Northwest
Portland
Tel (503) 293-9001
Fax (503) 293-9007

PENNSYLVANIA

Giesting & Associates
Pittsburgh
Tel (412) 828-3553
Fax (412) 828-6160

Metro Logic Corp
(AT&T only)
Fairfield, NJ
Tel (201) 575-5585
Fax (201) 575-8023

S J Associates, Inc
Mt Laurel, NJ 08084
Tel (609) 866-1234
Fax (609) 866-8627

PUERTO RICO

QXI of Florida, Inc
Fort Lauderdale
Tel (305) 978-0120
Fax (305) 972-1408

TENNESSEE

Rep Inc
Jefferson City
Tel (615) 475-9012
Fax (615) 475-6340

TEXAS

West Associates
Austin
Tel (512) 343-1199
Fax (512) 343-1922

West Associates
Houston
Tel (713) 621-5983
Fax (713) 621-5895

West Associates
Richardson
Tel (214) 680-2800
Fax (214) 699-0330

Domestic Representatives (Cont.)

UTAH

Utah Component
Sales Inc.
Midvale
Tel· (801) 561-5099
Fax. (801) 561-6016

WASHINGTON

Thorson Company
Northwest
Bellevue
Tel: (206) 455-9180
Twx. 9104432300
Fax: (206) 455-9185

WISCONSIN

Victory Sales
Milwaukee
Tel· (414) 789-5770
Fax: (414) 789-5760

OHMS Technology, Inc.
Edina, MN
Tel: (612) 932-2920
Fax: (612) 932-2918

Domestic Distributors

ALABAMA

Arrow/Schweber
Huntsville
Tel: (205) 837-6955
Fax: (205) 721-1581

Time Electronics
Huntsville
Tel· (205) 721-1133

ARIZONA

Arrow/Schweber
Tempe
Tel: (602) 431-0030
Fax: (602) 431-9555

Insight
Tempe
Tel: (602) 829-1800

Insight
Tucson
Tel: (602) 792-1800

Time Electronics
Tempe
Tel: (602) 967-2000

Wyle Laboratories
Phoenix
Tel: (602) 437-2088

CALIFORNIA

Arrow/Schweber
Calabasas
Tel. (818) 880-9686

Arrow/Schweber
San Diego
Tel: (619) 565-4800

Arrow/Schweber
San Jose
Tel. (408) 441-9700

Arrow/Schweber
San Jose
Tel: (408) 432-7171

Arrow/Schweber
Tustin
Tel: (714) 838-5422

F/X Electronics
Calabasas
Tel: (818) 591-9220

Insight
San Diego
Tel· (619) 587-1100

Insight
Westlake Village
Tel: (818) 707-2101

Insight
Irvine
Tel: (714) 727-3291

Insight
Sunnyvale
Tel. (408) 720-9222

Time Electronics
Anaheim
Tel: (714) 669-0100

Time Electronics
Chatsworth
Tel: (818) 998-7200

Time Electronics
San Diego
Tel· (619) 578-2500

Time Electronics
Sunnyvale
Tel: (408) 734-9888

Time Electronics
Torrance
Tel: (213) 320-0880

Wyle Laboratories
Santa Clara
Tel: (408) 727-2500

Wyle Laboratories
Rancho Cordova
Tel: (916) 638-5282

Wyle Laboratories
Irvine
Tel· (714) 863-9953

Wyle Laboratories
Irvine (Military Div.)
Tel: (714) 851-9953

Wyle Laboratories
Calabasas
Tel: (818) 880-9000

Wyle Laboratories
San Diego
Tel: (619) 565-9171

CANADA

Arrow/Schweber
Burnaby, B. C.
Tel: (604) 421-2333

Arrow/Schweber
Dorval, Quebec
Tel (514) 421-7411

Arrow/Schweber
Mississauga, Ontario
Tel: (416) 670-7769

Arrow/Schweber
Nepean, Ontario
Tel: (613) 226-6903

COLORADO

Arrow/Schweber
Englewood
Tel (303) 799-0258
Fax: (303) 799-0730

Insight
Aurora
Tel: (303) 693-4256

Time Electronics
Englewood
Tel· (303) 799-8851

Wyle Laboratories
Thornton
Tel: (303) 457-9953

CONNECTICUT

Arrow/Schweber
Wallingford
Tel: (203) 265-7741
Fax: (203) 265-7988

Time Electronics
Tel: (203) 271-3200

FLORIDA

Arrow/Schweber
Deerfield Beach
Tel: (305) 429-8200
Fax: (305) 428-3991

Arrow/Schweber
Lake Mary
Tel· (407) 333-9300

Time Electronics
Tel: (305) 484-7778

Time Electronics
Orlando
Tel: (407) 841-6565

Vantage Components
Altamonte Springs
Tel: (407) 682-1199

Vantage Components
Deerfield Beach
Tel: (305) 429-1001

GEORGIA

Arrow/Schweber
Duluth
Tel: (404) 497-1300

Time Electronics
Tel: (404) 448-4448

ILLINOIS

Arrow/Schweber
Itasca
Tel: (708) 250-0500

Arrow/Schweber
AT&T DOES Center
Tel: (908) 949-7621
Fax· (201) 984-8908

Marsh Electronics
Schaumburg
Tel: (708) 240-9290

Time Electronics
Schaumburg
Tel: (708) 303-3000

INDIANA

Arrow/Schweber
Indianapolis
Tel: (317) 299-2071
Fax: (317) 299-2379

Time Electronics
Tel: (800) 331-5114

IOWA

Arrow/Schweber
Cedar Rapids
Tel: (319) 395-7230
Fax: (319) 395-0185

Time Electronics
Tel. (800) 325-9085

KANSAS

Arrow/Schweber
Lenexa
Tel· (913) 541-9542
Fax: (913) 541-0328

Time Electronics
Tel· (800) 325-9085

KENTUCKY

Time Electronics
Tel· (800) 331-5114

MARYLAND

Arrow/Schweber
Columbia
Tel: (301) 596-7800
Fax: (301) 596-7821

Time Electronics
Baltimore
Tel: (301) 964-3090

Vantage Components
Columbia
Tel: (301) 720-5100
or. (301) 621-8555

Domestic Distributors (Cont.)

MASSACHUSETTS

Arrow/Schweber
Wilmington
Tel· (508) 658-0900

Port Electronics
Tyngsboro
Tel (508) 649-4880

Time Electronics
Peabody
Tel: (508) 532-9900

Wyle Laboratories
Burlington
Tel (617) 272-7300

MICHIGAN

Arrow/Schweber
Livonia
Tel (313) 462-2290
Fax. (313) 462-2686

Time Electronics
Tel (800) 331-5114

MINNESOTA

Arrow/Schweber
Eden Prairie
Tel. (612) 941-5280
Fax (612) 941-9405

Arrow/Schweber
Eden Prairie
Tel: (612) 941-1506
Fax: (612) 943-2086

MISSOURI

Arrow/Schweber
St Louis
Tel (314) 567-6888
Fax (314) 567-1164

Time Electronics
Manchester
Tel: (314) 391-6444

NEBRASKA

Time Electronics
Tel· (800) 325-9085

NEW JERSEY

Arrow/Schweber
AT&T DOES Center
Tel (908) 949-7627
Fax (201) 984-8708

Arrow/Schweber
Holmdel
Tel· (908) 949-4700
Fax (908) 949-4035

Arrow/Schweber
Marlton
Tel (609) 596-8000
Fax (609) 596-9632

Arrow/Schweber
Pine Brook
Tel (201) 227-7880
Fax (201) 227-2064

Time Electronics
Marlton
Tel (609) 596-6700

Time Electronics
N. New Jersey
Tel: (201) 882-4611

Vantage Components
Clifton
Tel: (201) 777-4100

NEW MEXICO

Insight
Tel (505) 823-1800

NEW YORK

Arrow/Schweber
Melville (Headquarters)
Tel (516) 391-1300

Arrow/Schweber
Hauppauge
Tel· (516) 231-1000
Fax (516) 231-1072

Arrow/Schweber
Rochester
Tel· (716) 427-0300
Fax (716) 427-0735

Time Electronics
Hauppauge (NYC)
Tel (516) 273-0100

Time Electronics
East Syracuse
Tel (315) 432-0355

Time Electronics
Rochester
Tel (716) 383-8853

Vantage Components
Smithtown
Tel (516) 543-2000

NORTH CAROLINA

Arrow/Schweber
Raleigh
Tel (919) 876-3132
Fax. (919) 878-9517

Time Electronics
Tel (800) 833-8235

NORTH DAKOTA

Time Electronics
Tel (800) 331-5114

OHIO

Arrow/Schweber
Solon
Tel (216) 248-3990
Fax. (216) 248-1106

Arrow/Schweber
Centerville
Tel: (513) 435-5563
Fax· (513) 435-2049

Time Electronics
Columbus
Tel. (614) 761-1100

OKLAHOMA

Arrow/Schweber
Tulsa
Tel· (918) 252-7537
Fax. (918) 254-0917

OREGON

Almac/Arrow Electronics
Beaverton
Tel (503) 629-8090
Fax: (503) 645-0611

Insight
Portland
Tel (503) 644-3300

Time Electronics
Portland
Tel (503) 684-3780

Wyle Laboratories
Beaverton
Tel (503) 643-7900

PENNSYLVANIA

Arrow/Schweber
Pittsburgh (Sales Office)
Tel (412) 963-6807
Fax (412) 963-1573

Time Electronics
Philadelphia
Tel: (215) 337-0900

Time Electronics
Pittsburgh
Tel. (800) 331-5114

Time Electronics
Marlton, NJ
Tel· (609) 596-6700

SOUTH DAKOTA

Time Electronics
Tel: (800) 331-5114

TEXAS

Arrow/Schweber
Austin
Tel· (512) 835-4180
Fax: (512) 832-9875

Arrow/Schweber
Carrollton
Tel (214) 380-6464
Fax (214) 248-7208

Arrow/Schweber
Houston
Tel (713) 530-4700
Fax (713) 568-8518

Insight
Austin
Tel (512) 467-0800

Insight
Ft. Worth
Tel (817) 338-0800

Insight
Houston
Tel (713) 448-0800

Insight
Richardson
Tel. (214) 783-0800

Time Electronics
Austin
Tel: (512) 339-3051

Time Electronics
Houston
Tel (713) 530-0800

Time Electronics
Richardson
Tel (214) 241-7441

Wyle Laboratories
Austin
Tel· (512) 345-8853

Wyle Laboratories
Houston
Tel· (713) 879-9953

Wyle Laboratories
Richardson
Tel· (214) 235-9953

UTAH

Arrow/Schweber
Salt Lake City
Tel: (801) 973-6913
Fax· (801) 972-0200

Time Electronics
West Valley
Tel (801) 973-8181

Wyle Laboratories
West Valley
Tel (801) 974-9953

WASHINGTON

Almac/Arrow Electronics
Bellevue
Tel. (206) 643-9992
Fax· (206) 643-9709

Almac/Arrow Electronics
Spokane
Tel. (509) 924-9500
Fax· (509) 928-6096

Insight
Kirkland
Tel· (206) 820-8100

Time Electronics
Redmond
Tel. (206) 882-1600

Wyle Laboratories
Redmond
Tel (206) 881-1150

WISCONSIN

Arrow/Schweber
Brookfield
Tel: (414) 792-0150
Fax (414) 792-0156

Marsh Electronics
Milwaukee
Tel: (414) 475-6000

Time Electronics
Tel: (800) 331-5114

International Distributors

AUSTRALIA
GEC/George Brown
Rydalmare, N.S.W.
Tel: 61-2-638-1888
Fax: 61-2-638-1798

AUSTRIA
Eljapex
Eitnergasse 6
A-1232 Wein
Tel: (43) 222-86-15-31
Fax: (43) 222-86-15-31-200

BELGIUM, LUX
D&D Electronics bvba
Antwerp
Tel: 32-38277934
Fax: 32-38287254

DENMARK
C-88 A/S
101 Kokkedal Industripark
DK-2980 Kokkedal
Tel: 45-42-24-48-88
Fax: 45-42-24-48-89

UNITED KINGDOM
Micro Call, Ltd.
Thame, Oxon 0X9 3XD
Tel: 44-84-426-1939
Fax· 44-84-426-1678

FINLAND
Nortec Electronics OY
SF-00210 Helsinki
Tel· 358-067-02-77
Tlx: 857125876
Fax· 358-06922326

FRANCE
A2M
B.P. 89
78152 LE CHESNAY
CEDEX
Tel: 33 (1) 39-54-91-13
Tlx: 698376F
Fax. 33 (1) 39-54-30-61

Microel
BP3
91941 Les Ulis
CEDEX
Tel: 33 (1) 69-07-08-24
Tlx. 692493F
Fax: 33 (1) 69-07-17-23

GERMANY
Jermyn GmbH
6250 Limburg
Tel: (06) 431-5080
Fax: (06) 431-508289

Scantec GmbH
D-33 Planegg
Tel: (089) 859-8021
Tlx: 5213219
Fax: (089) 857-6574

Topas Electronic GmbH
3000 Hannover 1
Tel: (0511) 13-12 -17
Tlx: 9218176
Fax. (0511) 13-12-16

HOLLAND
Arcobel bv
Griekenweg 25
5342 Px OSS
Tel: 31-4120-42322
Fax. 31-4120-30635

HONG KONG
CET, Ltd.
Tel: 852-520-0922
Fax: 852-865-0639

INDIA
Pamir Electronics Corp.
400 West Lancaster
Devon, PA 19333 USA
Tel· 215-688-5299
Fax: 215-688-5382
Tlx: 210656 Pamir UR

ISRAEL
Vectronics
60 Medinat Hayehudim St.
P O Box 2024
Herzlia B 46120, Israel
Tel: 972-52-556070
Tlx: 922342579
Fax: 972-52-556508

ITALY
Comprel s.p.a.
20092 Cinisello B.
Milano
Tel: (02) 6120641/5
Tlx. 332484 COMPRL
Fax: (02) 6128158

Silverstar
20126 Milano
Tel: 39 2661251
Fax: 39 266101922

JAPAN
Internix, Inc.
Shinjuku Hamada
Bldg. 7-4-7
Nishi-Shinjuku, Shinjuku-Ku
Tokyo 160
Tel: 813-3-369-1105
Fax: 813-3-363-8486

Kyocera Corporation
Setagaya-ku, Tokyo
Tel: 813-3-708-3111
Tlx· 7812466091
Fax 813-3-708-3864

Nippon Imex Corporation
Setagaya-ku, Tokyo
Tel: 813-3-321-8000
Tlx: 78123444
Fax: 813-3-325-0021

KOREA
Eastern Electronics, Inc.
Kangnam-Gu, Seoul
Tel: 82-2-553-2997
Tlx: 78727381
Fax: 82-2-553-2998

NORWAY
Nortec Electronics A/S
Postboks 123
N-1364 Hvalstad
Tel. 2-84-62-10
Fax: 2-84-65-45

PORTUGAL
ATD Electronica, Lda
Rua Faria de
Vasconcelos, 3-A
1900 Lisboa
Tel: 3511-847-2200
Fax: 3511-847-2197

SINGAPORE
Westech Electronics
Singapore 1334
Tel. 65-743-6355
Tlx: RS 55070
WESTEC
Fax: 65-746-1396

SPAIN
Sagitron
Corazon de Maria 80
28002 Madrid
Tel: 416-92-61
Tlx: 43819
Fax: 415-86-52

SWEDEN
Nortec Electronics A/B
Box 1830
S-171 27 Solna
Tel: 8-7051800
Fax: 8-836918

SWITZERLAND
Eljapex
Hardstr. 72
CH - 5430 Wettingen
Tel: (41) 56-27-57-77
Fax: (41) 56-26-14-86

Laser & Electronic
Equipment
8053 Zurich
Tel: 41 (1) 55-33-30
Fax: 41 (1) 55-34-58

TAIWAN
Ally, Inc.
Taipei
Tel: 886-2-788-6270
Fax: 886-2-786-3550

WSI Direct Sales Offices

REGIONAL SALES

Northeast
Stow, MA
Tel. (508) 685-6101
Fax: (508) 685-6105

Midwest
Hoffman Estates, IL
Tel: (708) 882-1893
Fax: (708) 882-1881

Southwest
Irvine, CA
Tel: (714) 753-1180
Fax: (714) 753-1179

Mid-Atlantic
Trevose, PA
Tel. (215) 638-9617
Fax: (215) 638-7326

Southeast
Dallas, TX
Tel: (214) 680-0077
Fax: (214) 680-0280

Northwest
Fremont, CA
Tel: (510) 656-5400
Telex: 289255
Fax: (510) 657-5916

EUROPE SALES

WSI – France
2 voie LA CARDON
91126 PALAISEAU
CEDEX, France
Tel: 33 (1) 69-32-01-20
Fax: 33 (1) 69-32-02-19

WSI – Germany
c/o B&RS
Rosenstrasse 7
8000 Munich 2, Germany
Tel: (49) 89.23 11 38.49
Fax: (49) 89.23.11.38.11

ASIA SALES

WSI – Asia, Ltd.
1006 C.C. Wu Bldg.
302-308 Hennessy Road
Wan Chai, Hong Kong
Tel: 852-575-0112
Fax. 852-893-0678

Corporate Headquarters

47280 Kato Road
Fremont, CA 94538
Tel: (510) 656-5400
Fax: (510) 657-5916

LIFE SUPPORT POLICY:
WaferScale Integration, Inc. (WSI) products are not authorized for use as critical components in life support systems or devices without the express written approval of the President of WSI As used herein:

A) Life support devices or systems are devices or systems which 1) are intended for surgical implant into the body, or 2) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury or death to the user,

B) A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Products in this book may be covered by one or more of the following patents Additional patents are pending

U.S.A. 4,328,565; 4,361,847, 4,409,723, 4,639,893, 4,649,520, 4,795,719; 4,763,184, 4,758,869, 5,006,974, 5,016,216; 5,014,097, 5,021,847, 5,034,786

West Germany· 3,103,160
Japan: 1,279,100
England 2,073,484, 2,073,487

MagicPro™ is a trademark of WaferScale Integration, Inc
IBM and IBM Personal Computer are registered trademarks of International Business Machines Corporation

Patents Pending

Rev. 1 4

47280 Kato Road
Fremont, California 94538-7333
Phone: 510/656-5400
Fax: 510/657-5916
TELEX: 289255
800/ TEAM-WSI (800/832-6974)
In California 800/562-6363

Printed in U.S.A. 2/92