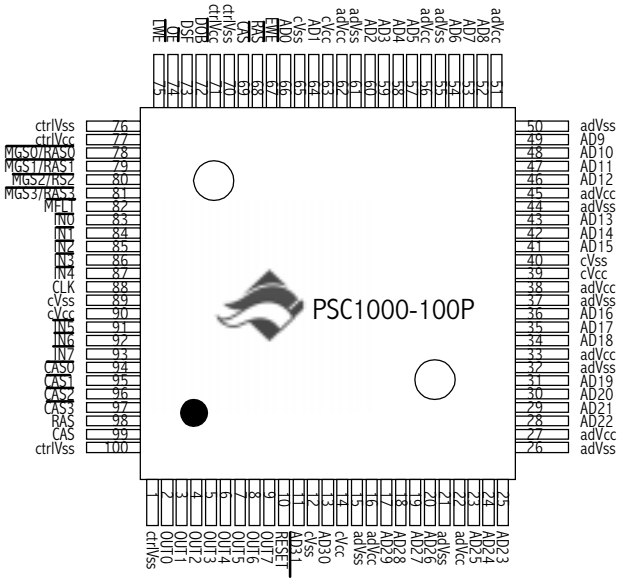
PSC1000 Microprocessor

32 BIT RISC Processor

FEATURES

- ♦ Low-System-Cost 32-Bit RISC Microprocessor
- ♦ 100-MHz (10ns/cycle) Operating Frequency
- ♦ Native execution of Java™
- ♦ Dual-Processor Architecture
- Microprocessing Unit (MPU)
  - ShBoom™ Architecture
  - High-performance zero-operand dual-stack architecture
- Input-Output Processor (IOP)
  - Used to perform timing, time-synchronous data transfers, bit outputs, DRAM refresh
- ♦ 4-Gigabyte Physical Address Space
- ♦ 50-MHz External Clock
- 2x Internal clock, 4x Bus timing
- On-chip PLL clock circuitry allows use of inexpensive oscillators
- ♦ 4-Group Memory/Bus Interface (MIF)
- Supports any combination of EPROM, SRAM, DRAM, VRAM
- Programmable memory group and I/O-channel timing
- ♦ 8-Level Interrupt Controller (INTC)
- ♦ 8-Level Direct Memory Access Controller (DMAC)
- ♦ 52 General-Purpose 32-Bit Registers
- ♦ "Glueless" System Interface
- ♦ Small, low-cost, 100-Pin PQFP package
- ♦ 5V & 3.3V version available
- ♦ Power consumption:
- 5V part: 350 mW @ 100 MHz
- 3.3V part: 165 mW @ 100 MHz



GENERAL DESCRIPTION

The PSC1000 Microprocessor is a highly integrated 32-bit RISC processor that offers high performance at low system cost for a wide range of embedded applications. At 100 MHz internally, the processor executes at 100 native MIPS peak performance, and 28 Dhrystone 2.1 MIPS. The processor is based upon Patriot's patented ShBoom™ dual stack architecture, which enables the native execution of Java code. The 32-bit registers and data paths fully support 32-bit addresses and data types. The processor addresses up to four gigabytes of physical memory, and supports virtual memory with the use of external mapping logic.

Conventional high-performance microprocessors are register-based with large register sets, and are pipelined or superscalar. These complex architectures consume costly silicon with multiple-operand instructions, multiple execution units, or lengthy execution pipelines. All these features diminish the fastest possible execution of individual instructions and increase silicon size, thus increasing chip cost.

The PSC1000 CPU architectural philosophy is that of simplification and efficiency of use. A zero-operand design eliminates most operand bits and the decoding time and instruction space they require. Instructions are shrunk to eight bits, significantly increasing instruction bandwidth and reducing program size. By not using pipeline or superscalar execution, the resulting control simplicity increases execution speed to issue *and* complete an instruction in a single clock cycle—as often as every clock cycle—without a conventional instruction cache. To ensure a low-cost chip, a data cache and its cost are also eliminated in favor of efficient register caches.

The PSC1000 CPU operates up to four groups of programmable bus configurations from as fast as 30 ns to as slow as 820 ns, allowing any desired mix of high-speed and low-speed memory. Minimum system cost is reduced, thus allowing the system designer to trade system cost for performance as needed.

By incorporating many on-chip system functions and a "glueless" bus interface, support chips are eliminated, further lowering system cost. The CPU includes an MPU, an I/O processor, a DMA controller, an interrupt controller, bit inputs, bit outputs, and a programmable memory interface. It can operate with 32-bit-wide and 8-bit-wide memory and devices, and includes hardware debugging support. A minimum system consists of a PSC1000 CPU, an 8-bit-wide EPROM, an oscillator, and optionally one x8 or two x16 memories—a total of 4 or 5 active components. The small die, which contains only 137,500 transistors, produces a high-performance, low-cost CPU, and a high level of integration produces a high-performance, low-cost system.

PSC1000 Microprocessor

32 BIT RISC Processor

PURPOSE

This information booklet describes the PSC1000 Microprocessor. The processor is targeted for embedded applications that require high MPU performance and low system cost. In addition, the processor's ability to execute Java natively makes it an ideal choice for Java-based devices. These include laser printers, cell-phones, ignition controllers, network routers, personal digital assistants, TV set-top boxes, embedded web servers, and many other applications. A full reference manual is available that provides the information required to design products that will use the PSC1000 CPU. It includes functional capability, electrical characteristics and ratings, and package definitions, as well as programming information.

OVERVIEW

The PSC1000 Microprocessor is a highly integrated 32-bit RISC processor that executes at 100 native MIPS peak performance, and 28 Dhrystone 2.1 MIPS, with a 100-MHz internal clock frequency. The CPU is designed specifically for use in those embedded applications for which MPU performance and system cost are deciding selection factors.

The PSC1000 CPU instruction sets are hardwired, allowing most instructions to execute in a single cycle, without the use of pipelines or superscalar architecture. A "flow-through" design allows the next instruction to start before the prior instruction completes, thus increasing performance.

The PSC1000 MPU contains 52 general-purpose registers, including 16 global data registers, an index register, a count register, a 16-deep addressable register/return stack, and an 18-deep operand stack. Both stacks contain an index register in the top element, are cached on chip, and, when required, automatically spill to and refill from external memory. The stacks minimize the data movement typical of register-based architectures, and also minimize memory accesses during procedure calls, parameter passing, and variable assignments. Additionally, the MPU contains a mode/status register, stack pointers, and 41 locally addressed registers for I/O, control, configuration, and status.

KEY FEATURES

Dual-Processor Architecture: The CPU contains both a high-performance, zero-operand, dual-stack architecture microprocessing unit (MPU), and an input-output processor (IOP) that executes instructions to transfer data, count events, measure time, and perform other timing-dependent functions.

Zero-Operand Architecture: Many RISC architectures waste valuable instruction space—often 15 bits or more per instruction—by specifying three possible operands for every instruction. Zero-operand (stack) architectures eliminate these operand bits, thus allowing much shorter instructions—typically one-fourth the size—and thus a higher instruction-execution bandwidth and smaller program size. Stacks also minimize register saves and loads within and across procedures, thus allowing shorter instruction sequences and faster-running code.

Fast, Simple Instructions: Instructions are simpler to decode and execute than those of conventional RISC processors, allowing the PSC1000 MPU and IOP to issue *and* complete instructions in a single clock cycle, as often as every clock cycle—each at 100 native MIPS peak execution.

Four-Instruction Buffer: Using 8-bit opcodes, the CPU obtains up to four instructions from memory each time an instruction fetch or prefetch is performed. These instructions can be repeated without rereading them from memory. This maintains high performance when connected directly to DRAM, without the expense of a cache.

Local and Global Registers: Local and global registers minimize the number of accesses to data memory. The local-register stack automatically caches up to sixteen registers and the operand stack up to eighteen registers. As stacks, the data space allocated efficiently nests and unnests across procedure calls. The sixteen global registers provide storage for shared data.

Posted Write: Decouples the processor from data writes to memory, allowing the processor to continue executing after a write is posted.

Programmable Memory/Bus Interface: Allows the use of lower-cost memory and system components in price-sensitive systems. The interface supports many types of EPROM/SRAM/DRAM/VRAM directly, including fast-page mode on up to four groups of DRAM devices. On-chip support of RAS cycle /OE and /WE, CAS-before-RAS, and the DSF signal allow use of VRAM without additional external hardware. Programmable bus timing and driver power allow the designer a range of solutions to system design challenges to match the time, performance and budget requirements for each project.

Clock Multiplier: Internally doubles and quadruples the external clock. An on-chip PLL circuit eliminates typical stringent oscillator specifications, thus allowing the use of lower-cost oscillators.

Fully Static Design: A fully static design allows running the clock from DC up to rated speed. Lower clock speeds can be used to drastically cut power consumption.

Hardware Debugging Support: Both breakpoint and single-step capability aid in debugging programs.

Virtual Memory: Supported through the use of external mapping SRAMs and support logic.

Floating-Point Support: Special instructions implement efficient single- and double-precision IEEE floating-point arithmetic.

Direct Memory Access Controller: Supports up to eight prioritized levels at data rates of up to 100 MB/second.

Interrupt Controller: Supports up to eight prioritized levels with interrupt responses as fast as eight 2X-clock cycles.

Eight Bit Inputs and Eight Bit Outputs: I/O bits are available for MPU and IOP application use, reducing the need for external hardware.

CENTRAL PROCESSING UNIT

The PSC1000 CPU architectural philosophy is that of simplification and efficiency of use: implement the simplest solution that adequately solves the problem and provides the best utilization of existing resources. In hardware, this typically equates to using fewer transistors, and fewer transistors means a lower-cost CPU.

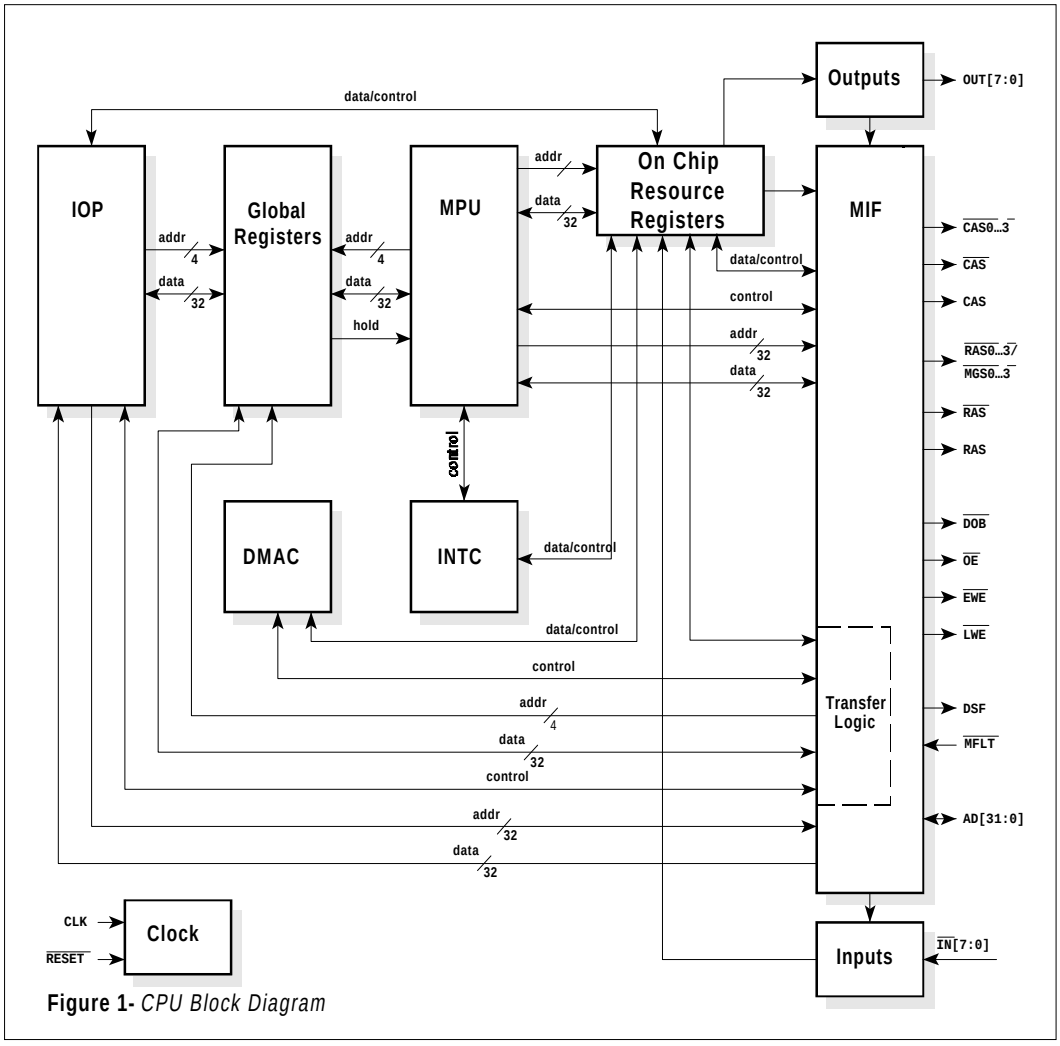


Figure 1- *CPU Block Diagram*

• The global registers are shared by the MPU, the IOP, and the transfer logic within the MIF. They are used by the MPU for data storage and control communication with the DMAC and the IOP; by the IOP for transfer information, loop counts, and delay counts; and by the DMAC for transfer information. Further, the transfer information is used by the transfer logic in the MIF that is shared by the IOP and DMAC.

• The MIF is shared by the MPU, the IOP, the DMAC, the bit outputs and the bit inputs for access to the system bus. Bus transaction requests are arbitrated and prioritized by the MIF to ensure temporally deterministic execution of the IOP.

• The bit inputs are made available to the system through the On-Chip Resource Registers. They are shared by the INTC and the DMAC for service requests, are available to the MPU and the IOP for programmed input, and are bit-addressable.

• The DMAC transfer-termination logic is significantly reduced by using specific termination conditions and close coupling with the MPU for intelligent termination action.

• The INTC is shared by the bit inputs, the IOP, and the DMAC (through the MIF transfer logic) for interrupt requests to the MPU.

• The bit outputs are made available to the system through the On-Chip Resource Registers. They are shared by the MPU and the IOP for programmed output, and are bit-addressable.

Early RISC processors reduced transistor counts compared to CISC processors, and gained their cost and performance improvements therein. Today, interconnections between transistors dominate the silicon of many CPUs. The PSC1000 MPU architectural philosophy results in, along with fewer transistors, the minimization of interconnections compared to register-based MPUs.

Resources

The PSC1000 CPU contains ten major functional areas: microprocessing unit (MPU), input-output processor (IOP), global registers, direct memory access controller (DMAC), interrupt controller (INTC), on-chip resources, bit inputs, bit outputs, programmable memory interface (MIF), and clock. In part, the PSC1000 CPU gains its small silicon size and capability from the resource sharing within and among these areas. **See Figure 1**. For example.

Although the maximum usage case requiring a complex IOP program, many interrupt sources, many input bits, many output bits, all available DMA channels, and maximum MPU computational ability might leave a shortage of resources, such applications are not typical. The sharing of resources among functional units significantly reduces transistor count, package pin count, and thus silicon size and cost, and increases CPU capability and flexibility. The ability to select among available resources, compared to the fixed resource set of other CPUs, allows the PSC1000 CPU to be used for a wider range of applications.

Clock Speed

The clock speed of a CPU is not a predictor of its performance. For instance, the PowerPC 604, running at about half the speed of the DEC Alpha 21064A, achieves about the same SPECint95 benchmark performance. In this respect, the PSC1000 CPU is more like the DEC Alpha than the PowerPC. However, the PSC1000 CPU is based on a significantly different design philosophy than either of these CPUs.

Most processors historically have forced the system designer to maintain a balanced triangle among CPU execution speed, memory bandwidth, and I/O bandwidth. However, as system clock rate increases so typically does bus speed, cache memory speed and system interface costs. Typically, too, so does CPU cost, as often thousands of transistors are added to maintain this balance.

The PSC1000 CPU lets the system designer select the performance level desired, while maintaining low system cost. This may tilt the triangle slightly, but cost is not part of the triangle-balancing equation. The PSC1000 CPU's programmable memory interface permits a wide range of memory speeds to be used, allowing systems to use slow or fast memory as needed. Slow memory will clearly degrade system performance, but the fast internal clock speed of the PSC1000 CPU causes internal operations to be completed quickly. Thus the multi-cycle multiply and divide instructions always execute quickly, without the silicon expense of a single-cycle multiply unit. At up to eight times the clock rate of competing parts with single-cycle multipliers, the difference in multiply/divide performance diminishes while the remainder of the application executes correspondingly faster. Although higher performance can sometimes be gained by dedicating large numbers of transistors to functions such as these, silicon cost also increases.

MICROPROCESSING UNIT

The PSC1000 MPU supports the PSC1000 CPU architectural philosophy of simplification and efficiency of use through its basic design in several interrelated ways.

Whereas most RISC processors use pipelines and superscalar execution to execute at high clock rates, the PSC1000 MPU uses neither. By having a simpler architecture, the PSC1000 MPU issues *and* completes most instructions in a single clock cycle. There are no pipelines to fill and none to flush during changes in program flow. Though more instructions are sometimes required to perform the same procedure in PSC1000 MPU code, the MPU operates at a higher clock frequency than other processors of similar silicon size and technology, thus giving comparable performance at significantly reduced cost.

A microprocessor's performance is often limited by how quickly it can be fed instructions from memory. The MPU reduces this bottleneck by using 8-bit instructions so that up to four instructions (an *instruction group*) can be obtained during each memory access. Each instruction typically takes one 2x-clock cycle to execute, thus requiring four 2x-clock cycles to execute the instruction group. Because a memory access can take four or fewer 2x-clock cycles, the next instruction group can be available when execution of the previous group completes. This makes it possible to feed instructions to the processor at maximum instruction-execution bandwidth without the cost and complexity of an instruction cache.

The zero-operand (stack) architecture makes 8-bit instructions possible. The stack architecture eliminates the need to specify source and destination operands in every instruction. By not using opcode bits on every instruction for operand specification, a much greater bandwidth of functional operations—up to four times as high—is possible. **Table 1** depicts an example PSC1000 MPU instruction sequence that demonstrates twice the typical RISC CPU instruction bandwidth. The instruction sequence on the PSC1000 MPU requires one-half the instruction bits, and the uncached performance benefits from the resulting increase in instruction bandwidth.

Stack MPUs are thus simpler than register-based MPUs, and the PSC1000 MPU has two hardware stacks to take advantage of this: the operand stack and the local-register stack. The simplicity is widespread and is reflected in the efficient ways stacks are used during execution.

The ALU processes data from primarily one source of inputs—the top of the operand stack. The ALU is also used for branch address calculations. Data bussing is thus greatly reduced and simplified. Intermediate results typically "stack up" to unlimited depth and are used directly when needed, rather than requiring specific register allocations and management. The stacks are individually cached and spill and refill automatically, eliminating software overhead for stack manipulation typical in other RISC processors. Function parameters are passed on, and consumed directly off of the operand stack, eliminating the need for most stack frame management. When additional local storage is needed, the local-register stack supplies registers that efficiently nest and unnest across functions. As stacks, the stack register spaces are only allocated for data actually stored, maximizing storage utilization and bus bandwidth when registers are spilled or refilled—unlike architectures using fixed-size register windows. Stacks speed context switches, such as interrupt servicing, because registers do not need to be explicitly saved before use—additional stack space is allocated as needed. The stacks thus reduce the number of explicitly addressable registers otherwise required, and speed execution by reducing data location specification and movement. Stack storage is inherently local,

g5 = g1 - (g2 + 1) + g3 - (g4 * 2)

Typical RISC MPU		PSC1000 MPU	
		push	g1
		push	g2
add	g2,#1,g5	inc	#1
sub	g5,g1,g5	sub	
		push	g3
add	g5,g3,g5	add	
		push	g4
shl	g4,#1,temp	shl	#1
		sub	
sub	temp,g5,g5	pop	g5
20 bytes		10 bytes	

Example of twice the instruction bandwidth available on the PSC1000 MPU

Table 1

so the global registers supply non-local register resources when required.

Eight-bit opcodes are too small to contain much associated data. Additional bytes are necessary for immediate values and branch offsets. However, variable-length instructions usually complicate decoding and complicate and lengthen the associated data access paths. To simplify the problem, byte literal data is taken only from the rightmost byte of the instruction group, regardless of the location of the byte literal opcode within the group. Similarly, branch offsets are taken as all bits to the right of the branch opcode, regardless of the opcode position. For 32-bit literal data, the data is taken from a subsequent memory cell. These design choices ensure that the required data is always right-justified for placement on the internal data busses, reducing interconnections and simplifying and speeding execution.

Since most instructions decode and execute in a single clock cycle, the same ALU that is used for data operations is also available, and is used, for branch address calculations. This eliminates an entire ALU often required for branch offset calculations.

Rather than consume the chip area for a single-cycle multiply-accumulate unit, the higher clock speed of the MPU reduces the execution time of conventional multi-cycle multiply and divide instructions. For efficiently multiplying by constants, a fast multiply instruction will multiply only by the specified number of bits.

Rather than consume the chip area for a barrel shifter, the counted bit-shift operation is “smart” to first shift by bytes, and then by bits, to minimize the cycles required. The shift operations can also shift double cells (64 bits), allowing bit-rotate instructions to be easily synthesized.

Although floating-point math is useful, and sometimes required, it is not heavily used in embedded applications. Rather than consume the chip area for a floating-point unit, MPU instructions to efficiently perform the most time-consuming aspects of basic IEEE floating-point math operations, in both single and double precision, are supplied. The operations use the “smart” shifter to reduce the cycles required.

Byte read and write operations are available, but cycling through individual bytes is slow when scanning for byte values. These types of operations are made more efficient by instructions that operate on all of the bytes within a cell at once.

Registers and Stacks

The register set contains 52 general-purpose registers, a mode/status register, two stack pointers, and 41 local address-mapped on-chip resource registers used for I/O, configuration, and status. **See Figure 2**.

The operand stack contains eighteen registers and operates as a push-down stack, with direct access to the top three registers (`s0–s2`). These registers and the remaining registers (`s3–s17`) operate together as a stack cache. Arithmetic, logical, and data-movement operations, as well as intermediate result processing, are performed on the operand stack. Parameters are passed to procedures and results are returned from procedures on the stack, without the requirement of building a stack frame or necessarily moving data between other registers and the frame. As a true stack, registers are allocated only as needed for efficient use of available storage. External operand stack memory is addressed by register `sa`.

The local-register stack contains sixteen registers and operates as a push-down stack with direct access to the first fifteen registers (`r0–r14`). Theses registers and the remaining register (`r15`) operate together as a stack cache. As a stack, they are used to hold subroutine return addresses and automatically nest local-register data. External local-register stack memory is addressed by register `la`.

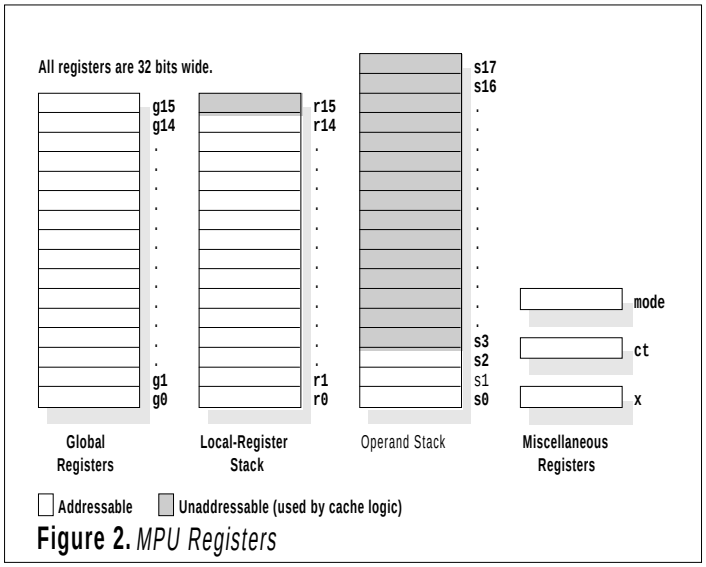


Figure 2. *MPU Registers*

Both cached stacks automatically spill to memory and refill from memory, and can be arbitrarily deep. Additionally, `s0` and `r0` can be used for memory access.

The use of stack-cached operand and local registers improves performance by eliminating the overhead required to save and restore context (when compared to processors with only global registers available). This allows for very efficient interrupt and subroutine processing.

In addition to the stacks are sixteen global registers and three other registers. The global registers (`g0–g15`) are used for data storage, as operand storage for the MPU multiply and divide instructions (`g0`), and for the IOP. Since these registers are shared, the MPU and the IOP can also communicate through them. Remaining are `mode`, which contains mode and status bits; `x`, which is an index register (in addition to `s0` and `r0`); and `ct`, which is a loop counter and also participates in floating-point operations.

Programming Model

For those familiar with American National Standard Forth (ANS Forth), or Hewlett-Packard calculators that use postfix notation, commonly known as Reverse Polish Notation (RPN), programming the PSC1000 MPU will in many ways be very familiar.

An MPU architecture can be classified as to the number of operands specified within its instruction format. Typical 16-bit and 32-bit CISC and RISC MPUs are usually two- or three-operand architectures, whereas smaller microcontrollers are often one-operand architectures. In each instruction, two- and three-operand architectures specify a source and destination, or two sources and a destination, whereas one-operand architectures specify only one source and have an implicit destination, typically the accumulator. Architectures are also usually not pure. For example, one-operand architectures often have two-operand instructions to specify both a source and destination for data movement between registers.

The PSC1000 MPU is a zero-operand architecture, known as a *stack computer*. Operand sources and destinations are assumed to be on the top of the operand stack, which is also the accumulator. An operation such as `add` uses both source operands from the top of the operand stack, adds them, and returns the result to the top of the operand stack, thus causing a net reduction of one in the operand stack depth. **See Figure 3**.

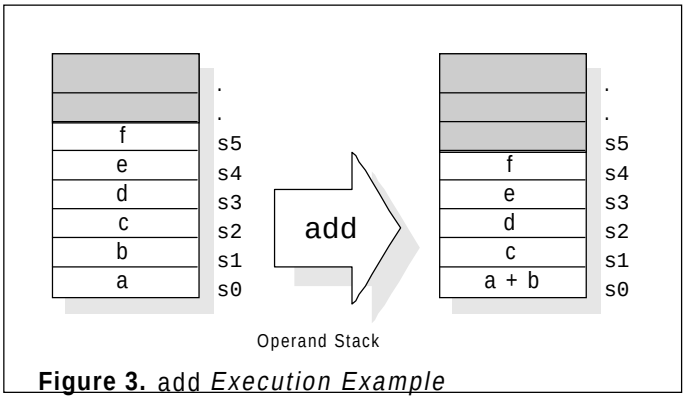


Figure 3. `add` *Execution Example*

Most ALU operations behave similarly, using two source operands and returning one result operand to the operand stack. A few ALU operations use one source operand and return one result operand to the operand stack. Some ALU and other operations also require a non-stack register, and a very few do not use the operand stack at all.

Non-ALU operations are also similar. Loads (memory reads) either use an address on the operand stack or in a specified register, and place the retrieved data on the operand stack. Stores (memory writes) use either an address on the operand stack or in a register, and use data from the operand stack. Data movement operations push data from a register onto the operand stack, or pop data from the stack into a register.

Once data is on the operand stack it can be used for any instruction that expects data there. The result of an `add`, for instance, can be left on the stack indefinitely, until needed by a subsequent instruction. Instructions are also available to reorder the data in the top few cells of the operand stack so that prior results can be accessed when required. Data can also be removed from the operand stack and placed in local or global registers to minimize or eliminate later reordering of stack elements. Data can even be popped from the operand stack and restacked by pushing it onto the local-register stack.

Computations are usually most efficiently performed by executing the most deeply nested computations first, leaving the intermediate results on the operand stack, and then combining the intermediate results as the computation unnests. If the nesting of the computation is complex, or if the intermediate results need to be used some time later after other data will have been added to the operand stack, the intermediate results can be removed from the operand stack and stored in global or local registers. Global registers are used directly and maintain their data indefinitely. Local registers are registers within the local-register stack cache and, as a stack, must first be allocated. Allocation can be performed by popping data from the operand stack and pushing it onto the local-register stack one cell at a time. It can also be preformed by allocating a block of uninitialized stack registers at one time; the uninitialized registers are then initialized by popping data, one cell at a time, into the registers in any order. The allocated local registers can be deallocated by pushing data onto the operand stack and popping it off of the local register stack one cell at a time, and then discarding from the operand stack the data that is unneeded. Alternatively, the allocated local registers can be deallocated by first saving any data needed from the registers, and then deallocating a block of registers at one time. The method selected will depend on the number of registers required and whether the data on the operand stack is in the required order.

Registers on both stacks are referenced relative to the tops of the stacks and are thus local in scope. What was accessible in `r0`, for example, after one cell has been push onto the local-register stack, is accessible as `r1`; the newly pushed value is accessible as `r0`.

Parameters are passed to and returned from subroutines on the operand stack. An unlimited number of parameters can be passed and returned in this manner. An unlimited number of local-register allocations can also be made. Parameters and allocated local registers thus conveniently nest and unnest across subroutines and program basic blocks.

Subroutine return addresses are pushed onto the local-register stack and thus appear as `r0` on entry to the subroutine, with the previous `r0` accessible as `r1`, and so on. As data is pushed onto the stacks and the available register space fills, registers are spilled to memory when required. Similarly, as data is removed from the stacks and the register space empties, the registers are refilled from memory as required. Thus from the program's perspective, the stack registers are always available.

All MPU instructions consist of eight bits, except for those that require immediate data. This allows up to four instructions (an instruction group) to be obtained on each instruction fetch, thus reducing memory-bandwidth requirements compared to typical RISC machines with 32-bit instructions. This characteristic also allows looping on an instruction group (a micro-loop) without additional instruction fetches from memory, further increasing efficiency.

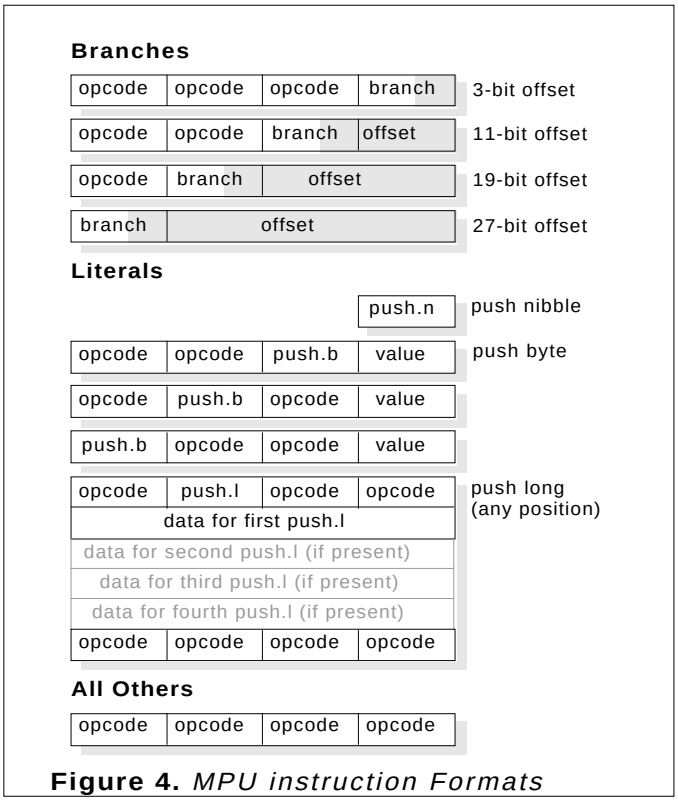


Figure 4. *MPU instruction Formats*

Table 2. *MPU Instructions*

Mnemonic	Description	Mnemonic	Description	Mnemonic	Description
add	add	ld [x++]	load indirect and postincrement x	rnd	round
add pc	add program counter	ld [x]	load indirect using x	scache	fill/empty operand stack cache
adda	add address	ld []	load indirect	sdepth	operand stack depth
addc	add with carry	ld.b []	load byte indirect	sexb	sign-extend byte
addexp	add exponents	ldo []	load on-chip indirect	sframe	allocate operand stack frame
and	bitwise AND	ldo.i []	load bit on-chip indirect	shift	counted shift
bkpt	breakpoint	ldepth	depth of local-register stack	shiftd	counted shift double
br offset	branch unconditionally	lframe	allocate local-register stack frame	shl #1	shift left one bit
br []	branch indirect	mloop	micro-loop unconditionally	shl #8	shift left eight bits
bz offset	branch if zero	mloop_	micro-loop on condition (c,n,nc,nz,p,z)	shld #1	shift double left one bit
call offset	call subroutine	mulfs	multiply fast signed	shr #1	shift right one bit
call []	call subroutine indirect	muls	multiply signed	shr #8	shift right eight bits
cmp	compare	mulu	multiply unsigned	shrd #1	shift double right one bit
copyb	copy byte	mxm	maximum	skip	skip unconditionally
dbr offset	decrement ct and branch	neg	negate	skip_	skip on condition (c,n,nc,nz,p,z)
dec #1	decrement by one	nop	no operation	split	split cell
dec #4	decrement by four	norml	normalize left	st [--r0]	store indirect and predecremented r0
dec ct	decrement ct by one	normr	normalize right	st [--x]	store indirect and predecrement x
denorm	denormalize	notc	complement carry	st [r0++]	store indirect and postincrement r0
di	disable interrupts	or	bitwise OR	st [r0]	store indirect using r0
divu	divide unsigned	pop	pop stack	st [x++]	store indirect and postincrement x
ei	enable interrupts	pop reg	pop into register	st [x]	store indirect using x
eqz	equal zero	pop lstack	pop into local-register stack	st []	store indirect
expdif	exponent difference	push	push stack	sto []	store on-chip indirect
extexp	extract exponent	push reg	push from register	sto.i []	store bit on-chip indirect
extsig	extract significand	push lstack	push from local-register stack	sub	subtract
iand	bitwise invert then AND	push.b #	push byte	subb	subtract with borrow
inc #1	increment by one	push.l #	push long	subexp	subtract exponents
inc #4	increment by four	push.n #	push nibble	testb	test cell for zero bytes
lcache	fill/empty local-register stack cache	replb	replace byte	testexp	test exponent
ld [--r0]	load indirect and predecremented r0	replexp	replace exponent	xcg	exchange stack
ld [--x]	load indirect and predecrement x	ret	return from subroutine	xor	bitwise exclusive OR
ld [r0++]	load indirect and postincrement r0	reti	return from interrupt		
ld [r0]	load indirect using r0	rev	revolve stack		

INPUT-OUTPUT PROCESSOR

The Input-Output Processor (IOP) is a special-purpose processing unit that executes instructions to transfer data between devices and memory,

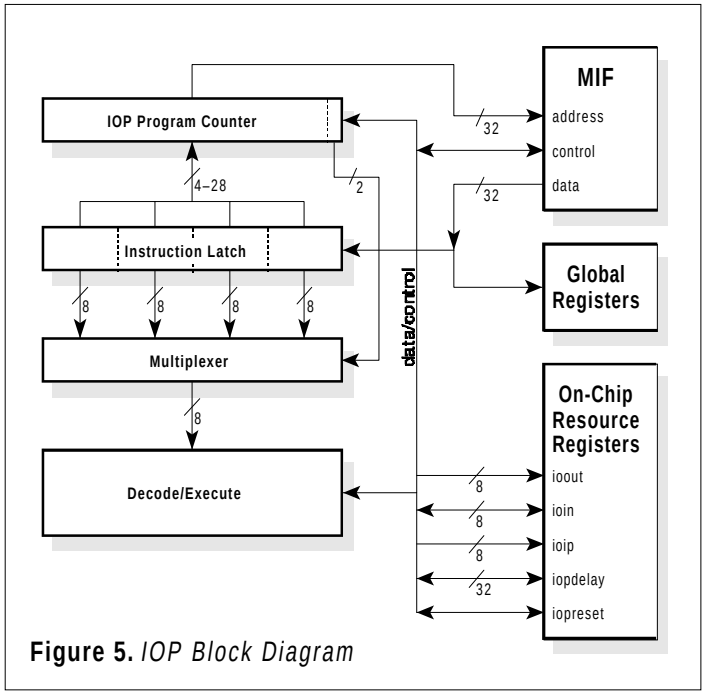


Figure 5. *IOP Block Diagram*

refresh dynamic memory, measure time, manipulate bit inputs and bit outputs, and perform system timing functions. IOP programs are usually written to be temporally deterministic. Because it can be difficult or impossible to write programs that contain conditional execution paths that execute in an efficient temporally deterministic manner, the IOP contains no computational and minimal decision-making ability. IOP programs are intended to be relatively simple, using interrupts to the MPU to perform computation or decision making.

To ensure temporally deterministic execution, the IOP exercises absolute priority over bus access. Bus timing must *always* be deterministic; wait states are programmed in the MIF. Temporal determinism is achieved by counting IOP-execution and bus clock cycles between the timed IOP events. Bus access is granted to the IOP unless it is executing `delay`, which allows MPU and DMA requests access to the bus during a specified time. Thus, when a memory access is needed, the IOP simply seizes the bus and performs the required operation at precisely the programmed instant.

The MIF ensures that the bus will be available when the IOP requires it. The MPU and the DMAC request the bus from the MIF, which prioritizes the requests and grants the bus while the IOP is executing `delay`. The MIF ensures that any transactions will be completed before the delay time expires and the IOP then siezes the bus. When transferring data, the IOP does not modify any data that is transferred; it only causes the bus transaction to occur at the programmed time. It performs time-synchronous I/O-channel transfers, as opposed to the DMAC, which prioritizes and performs asynchronous I/O-channel transfers. Other than how they are initiated, the two types of transfers are identical.

Usage

An IOP program can be used to eliminate an extensive amount of external logic and simplify system designs. Further, by using the IOP for timing-dependent system and application operations, timing constraints on the MPU program can often be eliminated or greatly relaxed.

For example, an IOP program of about 150 bytes supplies the data transfers and timing for a video display. The program produces vertical and horizontal sync, and transfers data from DRAM to a video shift register or palette. Additionally, the IOP supplies flexibility. Video data from various areas of memory could be displayed, without requiring that the data be moved to create a contiguous frame buffer. As new data areas are specified, the IOP instructions are rewritten by the MPU to change the program the IOP will execute for the next video frame. While this is executing, the MPU still has access to the bus to execute instructions and process data, and the DMAC still has access to the bus to transfer data.

Many other applications are possible. The IOP is best used for applications that require data to be moved, or some other event to occur, at specific times. For example:

- sending digitized 16-bit data values to a pair of DACs to play CD-quality stereo sound,
- sampling data from input devices at specified time intervals for the MPU to later process,
- sending data and control signals to display images on an LCD display,
- transferring data packets for an intelligent network interface,
- transferring synchronous data blocks for an intelligent SCSI controller,
- sending multiple channels of data to DACs for a wave-table synthesizer,
- controlling video and I/O for serial and X-Windows video terminals or PC video accelerators,
- controlling timed events in process-control environments,
- controlling ignition and fuel for automotive engines, or
- combining several of the above applications to create a PC multimedia board.

The IOP is designed to dictate access to the bus (to ensure temporally deterministic execution), but to be a slave to the MPU. The IOP can communicate status to the MPU by:

- the status changing on a device the IOP has accessed,
- loading a value in a global register,
- setting a bit output, or
- consuming a bit input.

The MPU can control the IOP by:

- rewriting IOP instructions in memory,
- modifying the global registers the IOP is using,
- clearing a bit input, or
- resetting the IOP.

The events controlled do not need to occur at a persistent, constant rate. The IOP is appropriate for applications whose event rates must be consistently controlled, whether once or many times. As an example of the former, the IOP can take audio data from memory and send it to

a DAC to play the sound at a continuous rate, for as long as the audio clip lasts. As an example of the latter, the IOP can be synchronized to the rotation of an automotive engine by the MPU in order for the IOP to time fuel injection and ignition, with the synchronization constantly changed by the MPU (by changing global registers or rewriting the IOP program) as the MPU monitors engine performance.

DELAY	NO OPERATION
DECREMENT AND SKIP	OUTPUT TRUE
INTERRUPT MPU	OUTPUT FALSE
JUMP	REFRESH
LOAD REGISTER	TEST INPUT AND SKIP
MICRO-LOOP	TRANSFER

Table 3. *Instruction Set*

Instruction Set

All IOP instructions consist of eight-bit opcodes except for `ld`, which requires 32-bit immediate data following, and `jump`, which requires a page-relative destination address. The use of eight-bit opcodes allows up to four instructions (referred to as an *instruction group*) to be obtained on each instruction fetch, thus reducing instruction memory-bandwidth requirements compared to typical RISC machines with 32-bit instructions. This characteristic also allows looping on the instruction group (a micro-loop) without additional instruction fetches from memory, further increasing efficiency. Each instruction requires one 2X-clock cycle to execute plus any delay or explicit or implicit bus cycle specified. Instruction formats are depicted in **Figure 6**.

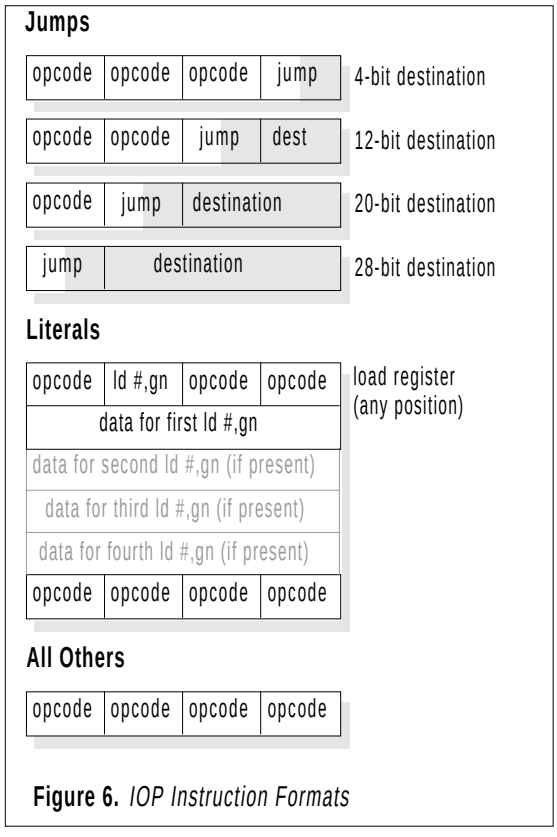


Figure 6. *IOP Instruction Formats*

- `delay`: load `iopdelay` and wait the specified number of 2X-clock cycles, allowing bus access for DMA and the MPU. `iopdelay` counts down once each 2X-clock cycle. DMA and MPU bus transactions are granted only when `iopdelay` indicates that sufficient time remains for the complete bus transaction to occur. When `iopdelay` reaches zero, the IOP instruction after `delay` executes.
- `dskipz`: decrement the specified global register and skip the remainder of the instruction group if the register is zero. Primarily used to create program loops by following `dskipz` with `jump`. Loops can be nested by using a different global register for each level of loop counter.
- `int`: request the specified interrupt from the MPU. Used to notify the MPU that an event has occurred.
- `jump`: branch to the specified page-relative address.
- `ld`: load the specified global register with the specified data. Used to load values for `xfer`, `mloop`, `dskipz` and `delay`, or to communicate with the MPU.
- `mloop`: loop on the instructions within the current instruction group. Used to loop on sequences of up to three other instructions without requiring the re-fetching of the instructions from memory.
- `nop`: no operation. Used as a placeholder for an instruction to be later placed or to waste time.
- `outt`: Set the specified bit output high.
- `outf`: Set the specified bit output low.
- `refresh`: performs a RAS-only refresh cycle on the memory groups that have refresh enabled. IOP program code must be written to include `refresh` at intervals adequate for any DRAM used.
- `tskipz`: test and consume the specified bit input, and skip the remainder of the instruction group if the bit is zero. Used to cause the IOP code to operate conditionally on bit inputs.
- `xfer`: cause an I/O-channel transfer to occur immediately using the specified global register. The global register contains the de vice address, memory address, and control information. **See Figure 7**. The type of bus transaction performed depends on whether the memory group involved is cell-wide or byte-wide and on the device transfer type. `xfer` bus transactions are identical to DMA bus transactions except for how they are initiated.

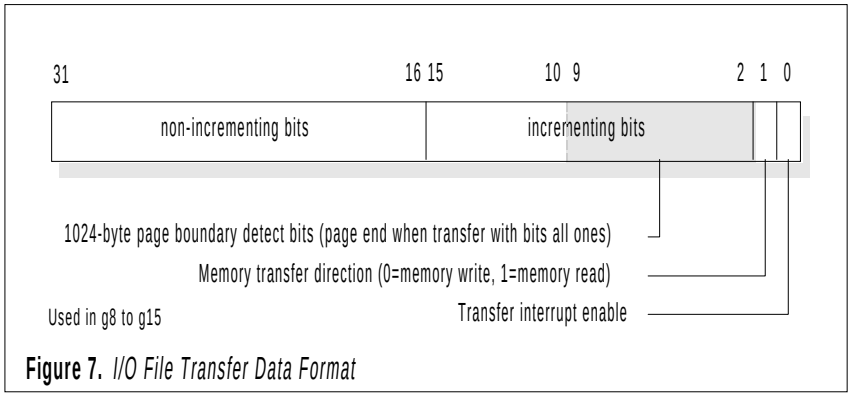


Figure 7. *I/O File Transfer Data Format*

DIRECT MEMORY ACCESS CONTROLLER

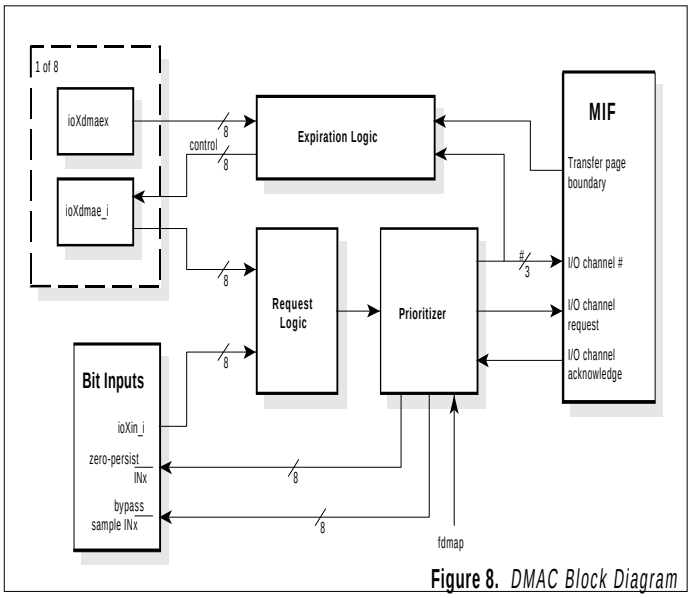


Figure 8. *DMAC Block Diagram*

The Direct Memory Access Controller (DMAC) allows I/O devices to transfer data to and from system memory without the intervention of the MPU. It supports eight I/O channels prioritized from eight separate sources. Direct memory access (DMA) requests are received from the bit inputs through `ioin`. DMA and MPU bus request priorities are either fixed, which allows higher-priority requests to block lower-priority requests, or revolving, which prevents higher-priority requests that cannot be satisfied from blocking lower-priority requests.

Device Width	Device Transfer Type[1]	Memory Width	Flyby2/ Buffered3	Bus Cycles4	Bits Moved
byte	0	byte	F	4	32
byte	0	cell	B	5	32
byte	1	byte	F	1	8
byte	1	cell	F	1	8
cell	2	byte	B	5	32
cell	2	cell	F	1	32

1.Refers to device type specified in `iodtta` or `iodttb`.
2.Data is transferred directly between device and memory.
3.Data is stored in the MIF during part of the transfer.
4.The entire sequence of cycles is an atomic bus transaction.

Table 4. *I/O Channel Transfer Characteristics*

DMA is supported for both cell-wide and byte-wide devices in both cell-wide and byte-wide memory. Each I/O channel can be individually configured as to the type of device and bus timing requirements. Byte-wide devices can be configured as either one-byte byte-transfer or four-byte byte-transfer devices. Transfers are flybys or are buffered, as required for the I/O-channel bus transaction. and IOP `xfer` transfers are identical except for how they are initiated. DMAC transfers occur from asynchronous requests whereas `xfer` transfers occur at their programmed time.

Prioritization

A DMA request is prioritized with other pending DMA requests, and, if the request has the highest priority or is the next request in revolving-priority sequence, its corresponding I/O channel will be the next to request the bus. DMA request prioritization requires one 2X-clock cycle to complete. When the I/O channel bus request is made, the MIF waits until the current bus transaction, if any, is almost complete. It then checks `iopdelay` to determine if the available bus slot is large enough for the required I/O channel bus transaction. If the bus slot is large enough, the bus is granted to the I/O channel, and the bus transaction begins.

The IOP always seizes the bus when `iopdelay` decrements to zero. Otherwise, a DMA I/O channel bus request and an MPU bus request contend for the bus, with the DMA I/O channel bus request having higher priority.

Memory and Device Addressing

Addresses used for I/O channel transfers contain both the I/O device address and the memory address. By convention, the uppermost address bits (when A31 is set) select I/O device addresses, while the lower address bits select the memory source/destination for the transfer. Multi-cycle transfer operations (e.g., transferring between a byte device and cell memory) assume A31 is part of the external I/O-device address decode and pass/clear A31 to select/deselect the I/O device as needed during the bus transaction.

1024-byte memory page boundaries have special significance to I/O channel transfers. When each I/O-channel bus transaction completes, bits 15–2 of the memory address in the global register are incremented. The new address is evaluated to determine if the last location in a 1024-byte memory page was just transferred (by detecting that bits 9–2 are now zero). When the last location in a 1024-byte memory page was just transferred, an MPU interrupt can be requested or DMA can be disabled.

Interrupts

An MPU interrupt can be requested after an I/O channel transfer accesses the last location in a 1024-byte memory page. The interrupt requested is the same as the I/O-channel number, and occurs if interrupts are enabled on that channel (i.e., if bit zero of the corresponding global register is set). This allows, for example, the MPU to be notified that a transfer has completed (by aligning the end of a transfer memory area with the end of a 1024-byte memory page), or to inform the MPU of progress during long transfers.

INTERRUPT CONTROLLER

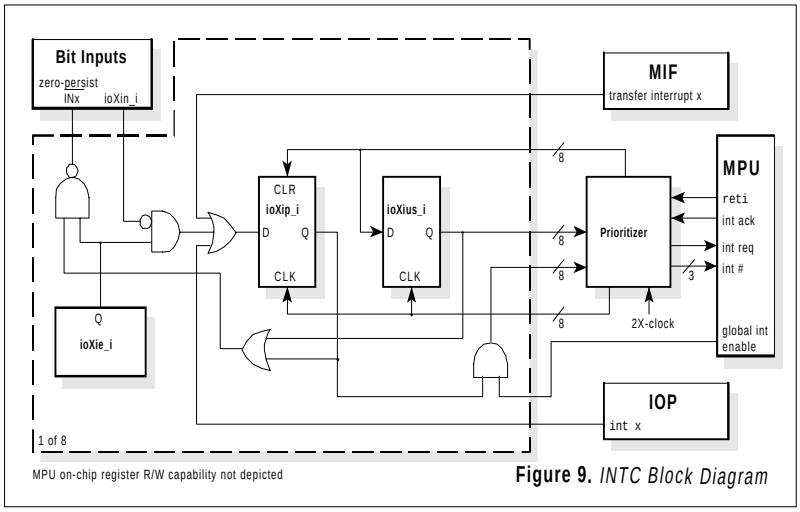


Figure 9. *INTC Block Diagram*

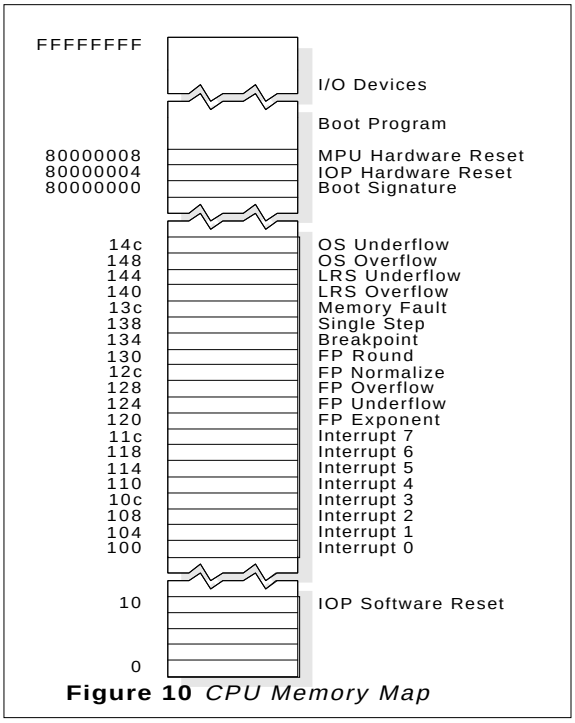


Figure 10 *CPU Memory Map*

The interrupt controller (INTC) allows multiple external or internal requests to gain, in an orderly and prioritized manner, the attention of the MPU. It supports up to eight prioritized interrupt requests from twenty-four sources. Interrupts are received from the bit inputs through `ioin`, from I/O-channel transfers, or from the IOP interrupt instruction `int`.

Each interrupt request is shared by three sources. A request can arrive from a zero bit in `ioin` (typically from an external input low), from an I/O-channel transfer interrupt, or from the IOP instruction `int`. Interrupt request zero comes from `ioin` bit zero, I/O channel zero (using `g8`), or `int 0`; interrupt request one comes from `ioin` bit one, I/O channel one (using `g9`), or `int 1`; the other interrupt requests are similarly assigned. Application usage typically designates only one source for an interrupt request, though this is not required.

Associated with each of the eight interrupt requests is an interrupt service routine (ISR) executable-code vector located in external memory. **See Figure 10**. A single ISR executable-code vector for a given interrupt request is used for all requests on that interrupt. It is programmed to contain executable code, typically a branch to the ISR. When more than one source is possible, the current source might be determined by examining associated bits in `ioin`, `ioie`, `iodmae` and the global registers.

BIT INPUTS

Eight external bit inputs are available in bit input register `ioin`. They are shared for use as interrupt requests, as DMA requests, as input to the IOP instruction `tskipz`, and as bit inputs for general use by the MPU. They are sampled externally from one of two sources determined by the state of `pkgio`.

All asynchronously sampled signals are susceptible to metastable conditions. To reduce the possibility of metastable conditions resulting from the sampling of the bit inputs, they are held for four 2X-clock cycles to resolve to a valid logic level before being made available to `ioin` and thus for use within the CPU. The worst-case sampling delay for bit inputs from `/IN[7:0]` to reach `ioin` is eight 2X-clock cycles.

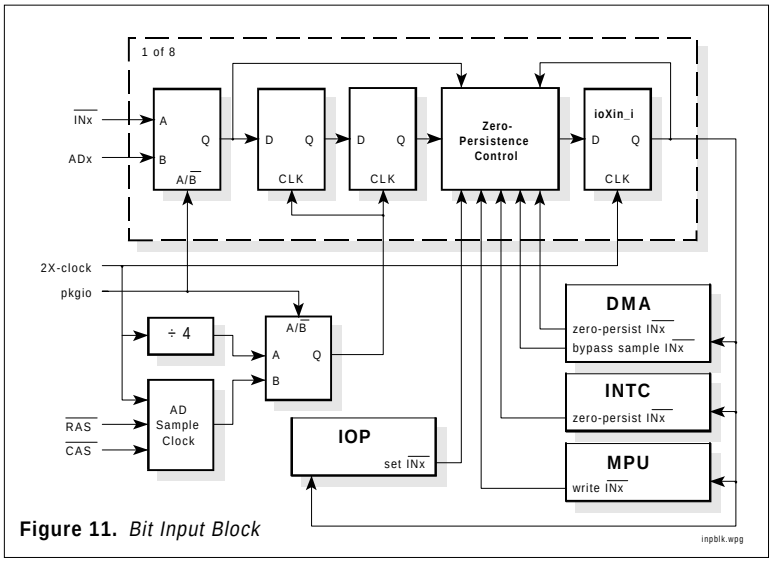


Figure 11. *Bit Input Block*

BIT OUTPUTS

On-Chip Resource Registers

Eight general-purpose bit outputs can be set high or low by either the MPU or the IOP. The bits are available in the bit output register, `ioout`.

The bits are read and written by the MPU as a group with `ldo [ioout]` and `sto [ioout]`, or are read and written individually with `ldo.i [ioXout_i]` and `sto.i [ioXout_i]`.

The bit outputs are written individually by the IOP with `outt` and `outf`. The bit outputs cannot be read by the IOP.

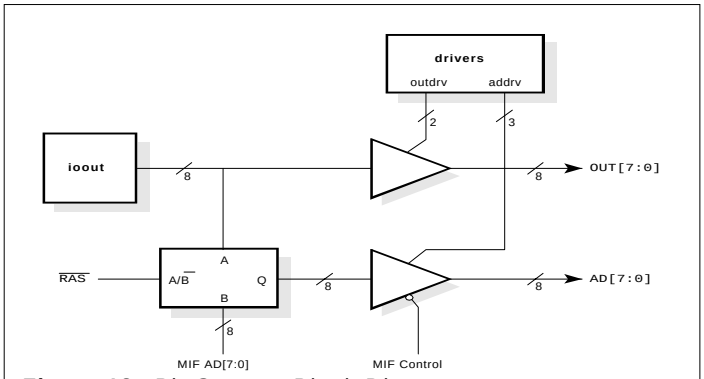
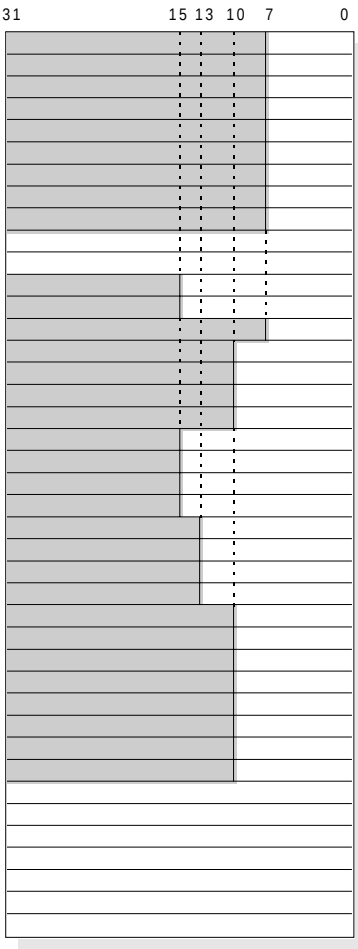


Figure 12. *Bit Outputs Block Diagram*

ON-CHIP RESOURCE REGISTERS



Addr	Mnemonic	Description
000	ioin	Bit Input Register
020	ioip	Interrupt Pending Register
040	ioius	Interrupt Under Service Register
060	ioout	Bit Output Register
080	ioie	Interrupt Enable Register
0a0	iodmae	DMA Enable Register
0c0	vram	VRAM Control Bit Register
0e0	misca	Miscellaneous A Register
100	miscb	Miscellaneous B Register
120	mfltaddr	Memory Fault Address Register
140	mfltdata	Memory Fault Data Register
160	msgsm	Memory System Group Select Mask Register
180	mgds	Memory Group Device Size Register
1a0	miscc	Miscellaneous C Register
1c0	mg0ebt	Memory Group 0 Extended Bus Timing Register
1e0	mg1ebt	Memory Group 1 Extended Bus Timing Register
200	mg2ebt	Memory Group 2 Extended Bus Timing Register
220	mg3ebt	Memory Group 3 Extended Bus Timing Register
240	mg0casbt	Memory Group 0 CAS Bus Timing Register
260	mg1casbt	Memory Group 1 CAS Bus Timing Register
280	mg2casbt	Memory Group 2 CAS Bus Timing Register
2a0	mg3casbt	Memory Group 3 CAS Bus Timing Register
2c0	mg0rasbt	Memory Group 0 RAS Bus Timing Register
2e0	mg1rasbt	Memory Group 1 RAS Bus Timing Register
300	mg2rasbt	Memory Group 2 RAS Bus Timing Register
320	mg3rasbt	Memory Group 3 RAS Bus Timing Register
340	**io0ebt**	**I/O Channel 0 Extended Bus Timing Register**
360	io1ebt	I/O Channel 1 Extended Bus Timing Register
380	io2ebt	I/O Channel 2 Extended Bus Timing Register
3a0	io3ebt	I/O Channel 3 Extended Bus Timing Register
3c0	io4ebt	I/O Channel 4 Extended Bus Timing Register
3e0	io5ebt	I/O Channel 5 Extended Bus Timing Register
400	io6ebt	I/O Channel 6 Extended Bus Timing Register
420	io7ebt	I/O Channel 7 Extended Bus Timing Register
440	msra	Memory System Refresh Address Register (WO)
440	iopdelay	IOP Delay Register (RO)
460	iodtta	I/O Device Transfer Types A Register
480	iodttb	I/O Device Transfer Types B Register
7a0	iodmaex	I/O DMA Enable Expiration Register
7c0	drivers	Driver Current Register
7e0	iopreset	IOP Reset Register

Figure 13. *On-Chip Resource Registers*

The on-chip resource registers comprise portions of various functional units on the CPU. The registers are addressed from the MPU in their own address space at the register level or at the bit level (for those registers that have bit addresses). On other processors, resources of this type are often either memory-mapped or opcode-mapped. By using a separate address space for these resources, the normal address space remains uncluttered, and opcodes are preserved.

The first six registers are bit addressable in addition to being register addressable. This allows the MPU to modify individual bits without corrupting other bits that might be changed concurrently by the IOP, DMAC, or INTC logic.

PROGRAMMABLE MEMORY INTERFACE

The Programmable Memory Interface (MIF) allows the timing and behavior of the CPU bus interface to be adapted to the needs of peripheral devices with minimal external logic. A variety of memory devices are supported, including EPROM, SRAM, DRAM and VRAM, as well as a variety of I/O devices. Most aspects of the bus interface are programmable, including address setup and hold times, data setup and hold times, output buffer enable and disable times, write enable activation times, memory cycle times, DRAM-type device address multiplexing, and when DRAM-type RAS cycles occur. Additional specifications are available for I/O devices, including data setup and hold times, output buffer enable and disable times, and device transfer type (one-byte, four-byte or one-cell).

The MIF supports direct connection to a variety of memory and peripheral devices. The primary requirement is that the device access time be deterministic; wait states are not available because they would create non-deterministic timing for the IOP. The MIF directly supports a wide range of sizes for multiplexed-address devices (DRAM, VRAM, etc.) up to 128 MB, as well as sizes for demultiplexed- address devices (SRAM, EPROM, etc.) up to 1 MB. Fast- page mode access and RAS-only refresh to DRAM-type devices are supported. SRAM-type devices appear to the MIF as DRAM with no RAS address bits and a large number of CAS address bits.

MPU, I/O-channel transfers require addressing an I/O device and a memory location simultaneously. This is achieved by an application-dependent splitting of the available 32 address bits into two areas: the lower address bits, which address memory, and the higher address bits, which address I/O devices. The areas can overlap, if required, with the side effect that an I/O device can only transfer data with a corresponding area of memory.

The MIF must always grant the bus to the IOP immediately when requested in order to guarantee temporally deterministic IOP execution.

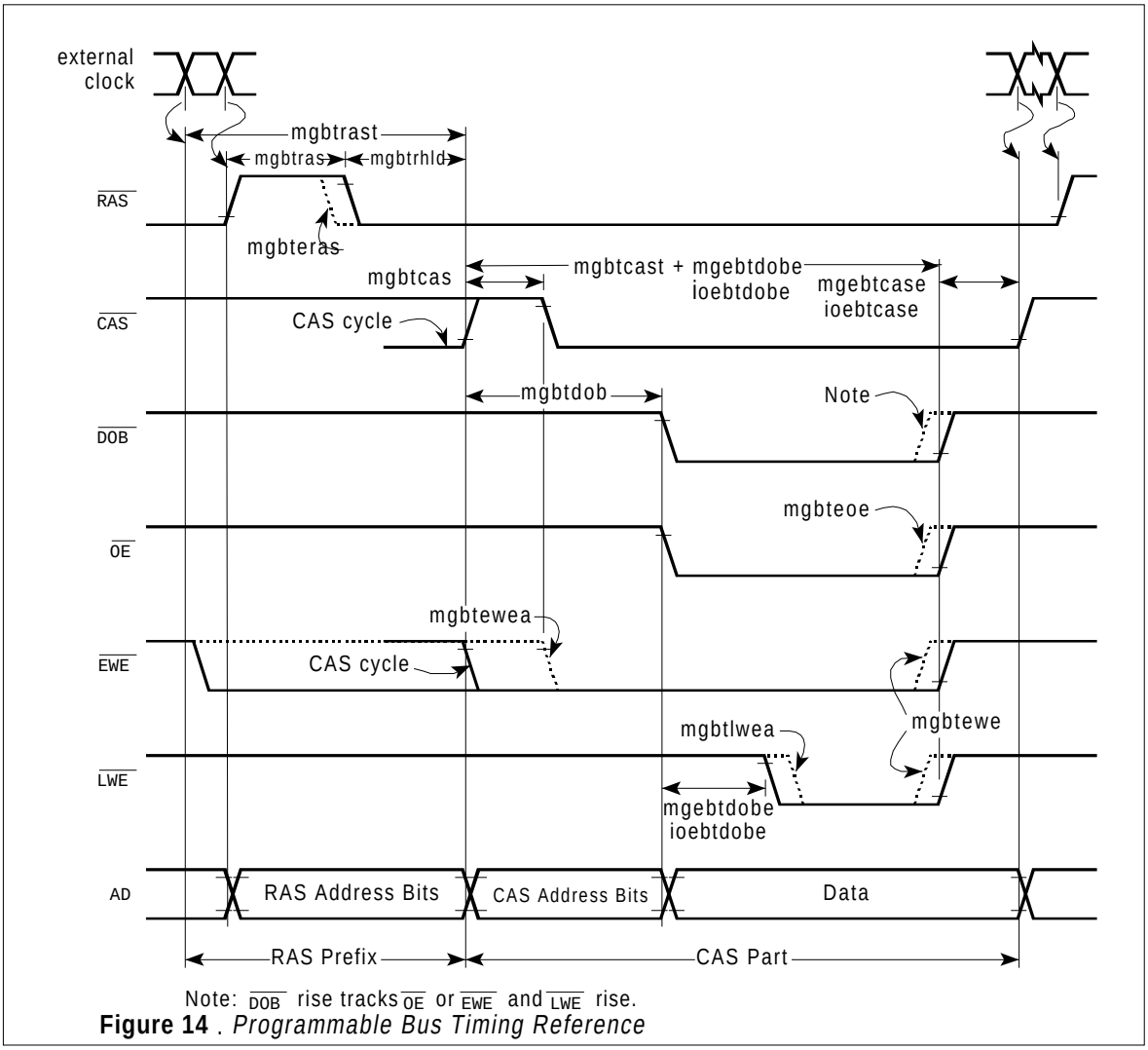


Figure 14 . *Programmable Bus Timing Reference*

Address bits are multiplexed out of the CPU on AD[31:9] to reduce package pin count. DRAM-type devices collect the entire memory address in two pieces, referred to as the *row address* and *column address*. Their associated bus cycles are referred to as *Row Address Strobe* (RAS) cycles and *Column Address Strobe* (CAS) cycles. With the exception of memory faults, refresh, and CAS-before-RAS VRAM cycles, a RAS cycle contains, enclosed within the /RAS active period, a CAS cycle.

Though I/O devices can be addressed like memory for access by the

To allow this, the IOP has exclusive access to the bus except when it is executing delay. When a DMA or MPU bus request is made, the MIF determines the type of bus transaction, computes the estimated time required, and compares this to iopdelay—the amount of time before the IOP seizes the bus. This available bus time is called the *slot*. If iopdelay is zero, the IOP currently has the bus. If iopdelay is larger than the value computed for the bus transaction, the bus is granted to the requestor. Once a bus request has passed the slot check, the bus transaction begins on the next 2X-clock cycle.

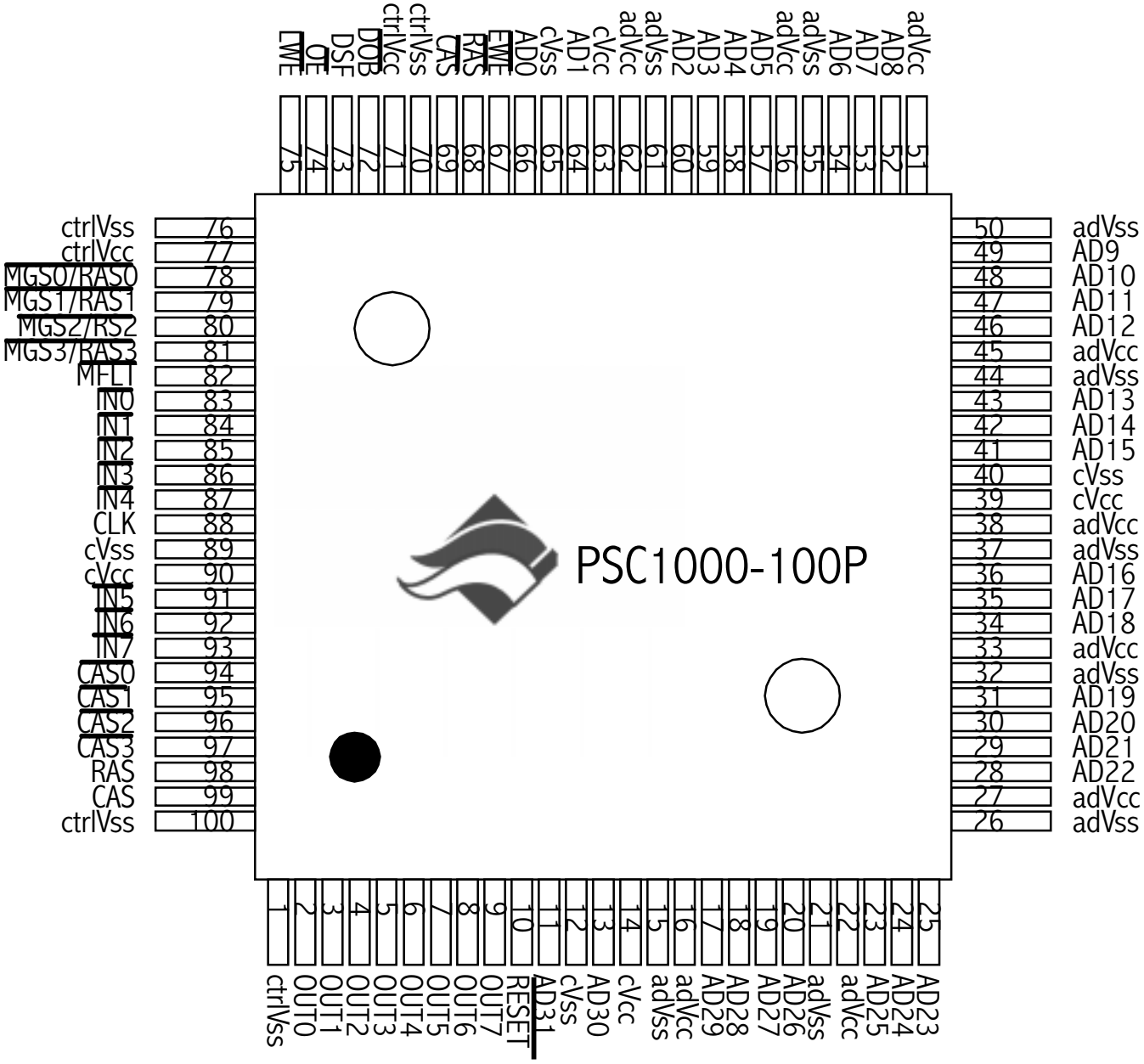
Table 5

SYMBOL	TYPE	DESCRIPTION
cV_{SS}	PWR	Ground for core logic and all output driver pre-drivers.
cV_{CC}	PWR	Power for core logic and all output driver pre-drivers.
$ctrlV_{SS}$	PWR	Ground for control signal output drivers (DSF, OUT[7:0], all RASes, all CASes, /DOB, /OE, /xWE).
$ctrlV_{CC}$	PWR	Power for control signal output drivers (DSF, OUT[7:0], all RASes, all CASes, /DOB, /OE, /xWE).
adV_{SS}	PWR	Ground for AD[31:0] output drivers.
adV_{CC}	PWR	Power for AD[31:0] output drivers.
CLK	I	**EXTERNAL OSCILLATOR:** The processor operating frequency is twice the external oscillator frequency.
/RESET	I A( )	**RESET:** Asserting /RESET causes the entire CPU to be initialized and the MPU and IOP to begin execution at their hardware reset locations. If /RESET is not held low during power-up, the signal alternatively is input on AD8 during /RAS active and /CAS inactive, and /RESET is ignored.
DSF	O I(L)	**DEVICE SPECIAL FUNCTION:** Set on VRAM memory cycles during /RAS and /CAS accesses by the MPU to control VRAM function.
/MFLT	I S(/RAS)	**MEMORY FAULT:** Asserted by external memory-management hardware before /RAS active to invalidate the current MPU bus cycle and cause the MPU to trap if the configuration bit pkgmflt is set. The signal alternatively is input on AD8 at /RAS fall during /CAS inactive, if the bit pkgmflt is clear.
/IN[7:0]	I A( )	**INPUTS:** Asserted by external hardware to request an interrupt or DMA, or to input a bit, when the configuration bit pkgio is set. The bits alternatively are input on AD[7:0] during /RAS active and /CAS inactive, if the bit pkgio is clear.
OUT[7:0]	O I(H)	**OUTPUTS:** Bit outputs writable from the IOP or MPU. These bits are also available on AD[7:0] during /RAS inactive.
/RAS	O I(L)	**ROW ADDRESS STROBE:** A control signal asserted to define row address valid and deasserted only when another row address cycle is required.
RAS	O, I(H)	Inverted /RAS
/CAS	O I(H)	**COLUMN ADDRESS STROBE:** A control signal asserted to define column address valid and deasserted at the end of the current bus cycle.
CAS	O, I(L)	Inverted /CAS
/MGS0…3/ /RAS0…3	O I(L)	**MEMORY GROUP SELECTS/ROW ADDRESS STROBES:** In multiple memory bank (MMB) mode (configuration bit mmb is set), the strobes are active during all bus cycles for the entire bus cycle. In single memory bank (SMB) mode, they are similar to /RAS.
/CAS0-3	O I(H)	**COLUMN ADDRESS STROBES:** Similar to /CAS, to assert a column address cycle on the specified memory bank within the current memory group.
/DOB	O, I(H)	**DATA ON BUS:** Active during data portion, inactive during the address portion of cycle.
/OE	O I(H)	**OUTPUT ENABLE:** Active when the current bus transaction is a read from memory. The configuration bit oed is set or cleared during the CPU reset startup process.
EWE	O I(H)	**EARLY WRITE ENABLE:** Active when the current bus transaction is a write to memory. Active time at either start of cycle or /CAS fall is programmable for each memory group.
/LWE	O I(H)	**LATE WRITE ENABLE:** Active when the current bus transaction is a write to memory and for VRAM control. Active time either at or after /DOB active is programmable for each memory group.
AD[31:0]	I/O S(/DOB) S(/RAS) A( ) I(Z)	**ADDRESS DATA BUS:** Multiplexed address, data, I/O and control bus. For data. For alternate memory fault on AD8. For alternate reset on AD8. See /RESET.

Notes:

I	=	Input-Only Pins	A( )	=	Asynchronous inputs	I(H)	= high value on reset
O	=	Output-Only Pins	S(sym)	=	Synchronous inputs must meet setup and hold requirements relative to symbol.	I(L)	= low value on reset
I/O	=	Bidirectional Pins				I(Z)	= high impedance on reset

PSC1000 Pin Out Diagram



PSC1000 Microprocessor

32 BIT RISC Processor

FEATURES

MICROPROCESSING UNIT (MPU)

Zero-operand dual-stack architecture
10-ns instruction cycle
52 General-purpose 32-bit registers
16 global data registers (g0–g15)
16 local registers (r0–r15) double as return stack cache
r0 is an index register with predecrement and postincrement
Automatic local-register stack spill and refill
18 operand stack cache registers (s0–s17)
s0 is an address register
Automatic operand stack spill and refill
Index register (x) with predecrement and postincrement
Count register (ct)
Stack paging traps
Cache-management instructions
MPU communicates with DMA and IOP via global registers
Hardware single- and double-precision IEEE floating-point support
Fast multiply
Fast bit-shifter
Hardware single-step and breakpoint
Virtual-memory support
Posted write
Power-on-reset flag
Instruction-space-saving 8-bit opcodes

DIRECT MEMORY ACCESS CONTROLLER (DMAC)

Eight prioritized DMA channels
Fixed or revolving DMA priorities
Byte, four-byte or cell DMA devices
Single or back-to-back DMA requests
Transfer rates to 100 MB/second
Programmable timing for each channel
Interrupt MPU on transfer boundary/count reached
Terminate DMA on transfer boundary/count reached
Channels can be configured as event counters
DMA communicates with MPU and IOP via global registers

INPUT-OUTPUT PROCESSOR (IOP)

Executes instruction stream independent of MPU
Deterministic execution
Performs timing, time-synchronous data transfers, bit-output operations, DRAM refresh
Eight transfer channels
Byte, four-byte or cell device transfers
Programmable timing for each channel
Interrupt MPU on transfer boundary/count reached
Set/reset output bits
Set MPU interrupt
Test and branch on input bit
Looping instructions
Load transfer address, direction, interrupt on boundary
IOP communicates with DMA and MPU via global registers or memory
Channels can be configured as timers
Instruction-space-saving 8-bit opcodes

INPUT-OUTPUT/INTERRUPTS

Eight bit inputs; Bits can be configured as zero-persistent
Eight bit outputs
I/O Bits are Register- and bit-addressable
I/O bits available on pins or multiplexed on bus
Eight prioritized and vectored interrupts

PROGRAMMABLE MEMORY INTERFACE (MIF)

Programmable bus interface timing to 1/4 external clock
Four independently configurable memory groups:
Any combination of 32-bit and 8-bit devices
Any combination of EPROM, SRAM, DRAM, VRAM
Almost any DRAM size/configuration
Fast-page mode access for each DRAM group
Glueless support for one memory bank per group
1.25 gates per memory bank for decoding up to 16 memory banks (four per memory group)
Virtual-memory support
DRAM refresh support (via IOP)
VRAM support includes DSF,/OE, WE,/CAS before /RAS control

For company and product information,
Patriot Scientific Corporation
10989 Via Frontera
San Diego, CA 92127
Phone : 619-674-5000
Fax: 619-674-5005
e-mail: info@ptsc.com
web site: http://www.ptsc.com