VOLUME 10, NUMBER 5
APRIL 15, 1996

MICROPROCESSOR REPORT

THE INSIDERS' GUIDE TO MICROPROCESSOR HARDWARE

New Embedded CPU Goes ShBoom

Patriot Scientific's Unusual 32-Bit Stack Machine Has 8-Bit Instruction Word

by Jim Turley

The world may not applaud the appearance of yet another 32-bit microprocessor architecture, but Patriot Scientific thinks it has found the right design to address a number of growing markets. The small Southern California startup recently unveiled its first microprocessor. The chip's stack-based architecture lends itself to efficient Java and PostScript execution, and the company has already landed its first design win: an Internet terminal.

The chip, which has a 32-bit internal architecture but uses tiny 8-bit instructions, stands to deliver some of the tightest, most compact code of any 32-bit microprocessor so far. Its name is as peculiar as its architecture: ShBoom.

Patriot received silicon of its first ShBoom implementation, the PSC1000, in October of last year and expects samples of a revised design to arrive this week. General sampling is slated to begin midyear, with 1,000-piece pricing set at $20. Patriot expects the chip to run at 50–75 MHz, delivering 17–21 Dhrystone MIPS performance.

The PSC1000 has no cache or on-chip RAM; it relies on a programmable memory controller adaptable for DRAM, VRAM, SRAM, and ROM. Future implementations may include RAM or ROM and a cache to improve performance.

The first ShBoom chip's modest power consumption and compact object code should help it in its struggle to gain market acceptance. The company is targeting industrial and automotive applications, real-time systems, and Internet terminals. Patriot has not licensed the design or teamed with any fabrication partners, although the company is considering both moves. For now, the 10-person firm hopes to sell the PSC1000 on its own, buying manufacturing capacity from a number of semiconductor vendors.

The ShBoom architecture has a long and checkered history. From an original design by Forth advocate Chuck Moore, the project was acquired by Nanotronics in 1991 and then by Patriot in 1994. After some initial fits and starts and a few generations of silicon, the first working parts are almost ready for general sampling.

Weird Can Be Good

The ShBoom architecture is fundamentally different from most other microprocessors because it is stack-based. The chip's register set, illustrated in Figure 1, is a combination of last-in first-out queues, or stacks, and individually addressable registers. The primary operand stack is addressable only as a stack; the local register file is addressable both as a stack and as individual registers; the global registers are only directly addressable.

Operands are pushed onto and popped off the operand stack, and instructions operate mainly on the top one or two items. An ADD instruction, for example, always replaces the top two items on the stack with their sum.

Forth programmers and those familiar with Hewlett-Packard calculators will recognize this system as postfix notation, also called reverse-Polish notation (RPN). Although not as common as the familiar algebraic notation most people are taught in school, postfix notation is efficient

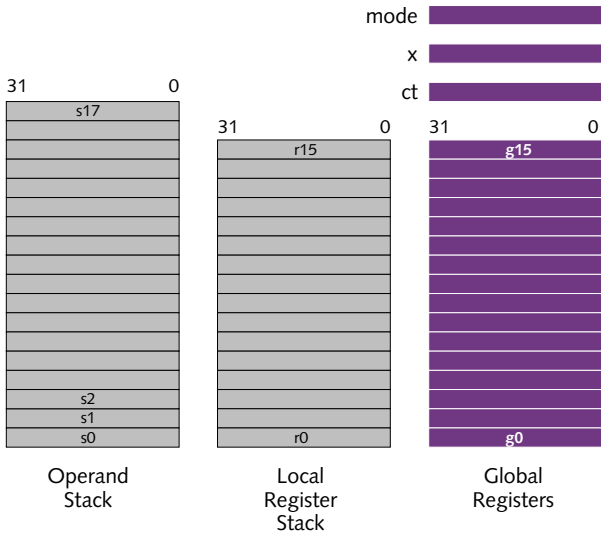

Figure 1. The ShBoom register set includes a primary operand stack and two additional resources: a local register file/stack and a global register set.

in terms of its data storage (i.e., code density). To use postfix notation effectively, though, requires a different programming model than most engineers are used to.

Primary Register Set Is Managed As a Stack

All of the PSC1000's registers are 32 bits wide, and most instructions operate on 32-bit data. In our taxonomy, these features classify the chip as a 32-bit microprocessor, despite its diminutive instruction word.

The major programming resource is the operand stack, s0–s17. To the programmer, this is managed as a true stack, with only the top three items accessible. With very few exceptions, all ShBoom instructions operate on only the top of the operand stack, register s0. Data is pushed onto the stack before processing; most operations consume one or two of these items and push the result onto the stack.

Besides the operand stack, the programmer has access to two additional resources: the local register stack, r0–r15, and the global registers, g0–g15. As the name implies, the local register stack is managed as a second stack, but it is also directly addressable as registers. The global registers are addressable only as individual registers.

Stack-based architectures have their good and bad sides. Instructions are very compact, because no bits are needed to specify operands. Stack-oriented programming can theoretically be very efficient, but in practice, some fudging is always required. Ideally, only those operands that are actually needed are pushed onto the stack, with intermediate results remaining on the stack only as long as they are required. Unneeded values should be removed from the stack, allowing more important results to surface. In reality, having an additional resource for storing temporary variables and intermediate values results in more efficient (and less awkward) code. Hence, the local and global registers.

Programs can push values onto the local register stack by popping them off the top of the operand stack. Like the operand stack, values can be pushed onto the local register stack and popped back off. The local register stack grows and shrinks as it is used. Transferring operands between these two stacks serves to reverse their order, a useful side effect.

The global registers, in contrast, act like the general-purpose registers in a more conventional microprocessor

Push Value	
PUSH	Duplicate top of stack
PUSH CT	Push from ct register
PUSH X	Push from x register
PUSH Rn	Push from local register n
PUSH Gn	Push from global register n
PUSH Sn	Push from operand register n
PUSH LSTACK	Remove from local stack
PUSH MODE	Push from mode register
PUSH.N #n	Push nibble constant
PUSH.B #n	Push byte constant
PUSH.L #n	Push word constant
PUSH LA	Push local stack pointer
PUSH SA	Push operand stack pointer
Pop Value	
POP	Discard top of stack
POP CT	Load ct register
POP X	Load x register
POp Rn	Load local register n
POP Gn	Load global register n
POP LSTACK	Push onto local stack
POP MODE	Load mode register
POP LA	Load local stack pointer
POP SA	Load operand stack pointer
Arithmetic	
ADD	Add
ADDC	Add with carry
ADDA	Add address
SUB	Subtract
SUBB	Subtract with borrow
MULS	Multiply signed
MULU	Multiply unsigned
MULFS	Multiply fast signed
DIVU	Unsigned divide
INC #1	Increment by 1
INC #4	Increment by 4
DEC #1	Decrement by 1
DEC #4	Decrement by 4
DEC CT	Decrement ct register

Load/Store to/from Memory	
LD [R0]	Load indirect through r0
LD [X]	Load indirect through x
LD [ ]	Load indirect through s0
LD.B [ ]	Load byte indirect thru s0
LD [--R0]	Load, predecrement r0
LD [--X]	Load, predecrement x
LD [R0++]	Load, postincrement r0
LD [X++]	Load, postincrement x
LDO [ ]	Load from on-chip resource
LDO.L [ ]	Load bit from on-chip
ST [R0]	Store indirect through r0
ST [X]	Store indirect through x
ST [ ]	Store indirect through s0
ST [--R0]	Store, predecrement r0
ST [--X]	Store, predecrement x
ST [R0++]	Store, postincrement r0
ST [X++]	Store, postincrement x
STO [ ]	Store to on-chip resource
STO.L [ ]	Store bit to on-chip
Flow Control	
BR	Branch unconditionally
BR [ ]	Branch indirect
BZ	Branch if zero
DBR	Decrement and branch
CALLI	Call subroutine
CALL [ ]	Call subroutine indirect
RET	Return from subroutine
RETI	Return from interrupt
MLOOPcc	Microloop on condition
SKIPcc	Skip on condition
BKPT	Call breakpoint trap
Stack-Cache Control	
LCACHE	Confirm local stack size
SCACHE	Confirm operand stack size
LDEPTH	Return local stack size
SDEPTH	Return operand stack size
LFRAME	Allocate local stack space
SFRAME	Allocate operand stack space

Shift and Logical	
AND	Logical AND
IAND	Logical invert-AND
OR	Logical OR
XOR	Logical exclusive-OR
NEG	Two's complement negate
NOTC	Invert carry flag
SHIFT	Shift by signed count
SHIFTD	Shift double by signed count
SHR #1	Shift right by 1
SHR #8	Shift right by 8
SHL #1	Shift left by 1
SHL #8	Shift left by 8
SHLD #1	Shift left double by 1
SHRD #1	Shift right double by 1
CMP	Compare
MXM	Select maximum
TESTB	Test byte for zero
EQZ	Set if equal zero
Floating-Point Support	
EXTEXP	Extract exponent
EXTSIG	Extract significand
DENORM	Denormalize FP number
NORMR/L	Normalize FP right/left
ADDEXP	Add exponents
SUBEXP	Subtract exponents
REPLEXP	Replace exponent
TESTEXP	Test exponent
EXPDIF	Exponent difference
RND	Round FP number
Miscellaneous	
REPLB	Replace byte into word
COPYB	Copy byte into word
SPLIT	Split word
REV	Revolve operand stack
XCG	Exchange operands
EI/DI	Enable/disable interrupts
STEP	Single-step CPU
NOP	No operation

Table 1. The ShBoom instruction set includes a collection of data-movement instructions to transfer data to and from the primary operand stack, integer arithmetic functions, memory loads and stores, and a set of floating-point conversion instructions for handling FP values.

architecture. Values can be moved from the operand stack to any of the global registers or copied from a global register to the top of the operand stack. This, and accessing registers in the local register stack, are the only cases wherein ShBoom instructions can designate a specific register to use.

Physically, the operand stack is implemented as on-chip RAM, with a queue pointer identifying the current top of stack. The stack is 18 words deep; pushing additional data onto the stack causes the word in s17 to overflow to external memory. Likewise, when operands are consumed off the top of the stack and the stack shrinks, previously overflowed values are reloaded from external RAM. This protects the operand stack from overflow or underflow and is invisible to the programmer, although the memory accesses degrade performance. It also allows future versions of ShBoom to implement more or fewer physical registers. The local register stack works the same way as the operand stack.

The PSC1000 has only one condition-code flag: a carry bit in the mode register. Conditional instructions reference the carry flag or test the top of the operand stack for sign and zero. The ct, or count, register controls repetitive operations. Finally, the x register is used as an address pointer.

Instruction Set Includes Conventional Functions

One of the PSC1000's most significant features is its compact instruction set. All instructions are encoded in exactly eight bits. This remarkably compact encoding is possible only because (with very few exceptions) none of the instructions specify a source register, destination register, or memory address. Nearly all ShBoom instructions operate only on the data at the top of the operand stack. Therefore, all eight bits of each instruction are dedicated to specifying the operation, not the operands. For ShBoom, the definitions of an opcode and an instruction are virtually identical.

Despite the severe limitations on instruction size, the PSC1000 has a reasonably rich instruction set. As Table 1 shows, ShBoom includes load, store, push, and pop instructions to transfer data between memory and the stack or between the stack and the other registers. The chip can add and subtract with carry/borrow, multiply, divide, shift, invert, and compare operands. The usual logical instructions are also included: OR, AND, XOR, and NOT AND.

ShBoom does not have a rotate instruction. To perform a logical rotate, programs duplicate the data on the top of the stack and execute a 64-bit shift instruction, shifting bits into the destination from the copy of the source.

Memory-resident data is referenced through one of three address pointers: the top of the stack (s0), x, or r0. Both r0 and x support predecrement and postincrement addressing for loads and stores, greatly simplifying memory-scan and -fill operations. No special addressing modes are supported; programs must calculate memory addresses via the usual instructions, leaving the result in s0. The address can then be moved to one of the other two pointer registers.

Most instructions operate on an entire 32-bit register. To support common C data types such as unsigned chars, ShBoom can load a single byte from memory, sign extend bytes, or merge a byte into a register.

Flow-control instructions are handled as in any other microprocessor. ShBoom has unconditional branch and call instructions, one conditional branch (if s0 equals zero), and a decrement-and-branch instruction. The processor also includes conditional and unconditional forms of the SKIP and MLOOP (micro-loop) instructions, which are useful for either skipping or repeating, respectively, very small three-instruction constructs.

Instruction Grouping Plays a Significant Role

Because ShBoom instructions are only a single byte, the chip is able to fetch four at a time over its 32-bit bus. Each group of four instructions is held in a simple fetch buffer. One instruction is removed from the buffer and executed to completion on every clock cycle. ShBoom has no pipeline to speak of; only one instruction is in process at any given time. When the group of four instructions is finished, the next group of four is fetched.

This grouping is not arbitrary and is of significant interest to the programmer. Because the PSC1000 holds four instructions at once, it can branch backward by up to three instructions without fetching the target instruction again. This is the purpose of the MLOOP instruction. By loading an unsigned 32-bit loop count into the ct register and executing an MLOOP instruction, programs can repeat a short loop up to four billion times without accessing external memory. For a chip with no instruction cache, this is a significant feature.

The MLOOP instruction does not accept a branch-target address; this is where the instruction grouping becomes an issue. The destination of an MLOOP is always the first instruction of the four-instruction group, regardless of where in that group the MLOOP instruction itself appears. Thus, it is important that the programmer (or compiler) structure ShBoom code so that the intended target of the microloop falls on a 32-bit boundary. This is one of the unusual features that contributes to the PSC1000's code density.

The SKIP instruction works in a similar manner. Like MLOOP, SKIP has no target address but simply skips over the remaining instructions in the group. The number of instructions skipped depends on the alignment of the SKIP instruction itself. A SKIP at the end of a group is treated as a NOP.

Operand Specifications Depend On Alignment

Specifying immediate operands is even more peculiar. With little room in an 8-bit instruction for immediate data, ShBoom can place only nibble-sized operands (-7 to +8) in line. For larger values, the chip relies on instruction grouping to identify data constants or literals. As a rule, when any instruction specifies a byte constant, the data is taken from the least-significant byte of the four-byte instruction group—regardless of where the instruction is in the group.

For example, if the first instruction of a four-instruc-

tion group specifies an 8-bit constant (such as a PUSH.B #n instruction, which pushes a byte onto the stack), ShBoom interprets the last byte of the group as the data value in question. This is true whether the PUSH.B appears as the first, second, or third instruction of the group. Placing such an instruction at the end of a group would push the encoding of the instruction itself onto the stack.

Figure 2 illustrates three examples. The placement of the actual instruction is irrelevant; the data is always taken from the last byte of the group. If the constant instruction appears first in the group, the two intervening instructions do not affect its outcome. The PSC1000 will execute the three instructions in order, skipping over the data value at the end before executing another group of four instructions.

Writing code with more than one such instruction in a single group is not generally effective; each instruction will simply reference the same byte constant. This construct could, for instance, be used to push the same value onto the stack three times in a row.

To encode 32-bit values, such as address pointers, data is read from the next four-byte group. Again, it doesn't matter which of the four byte locations the instruction occupies. Unlike the byte-sized example, programs can load multiple 32-bit operands in series, chaining them together in consecutive words of memory. As many as four such instructions may be grouped, each specifying a 32-bit constant. Their respective values are fetched in the order they appear. Thus, four 32-bit load instructions occupy five words of memory.

In contrast to data constants, PC-relative branches are highly dependent on their position when encoding their branch targets. The branch and call instructions are each encoded such that their least-significant three bits are unused. These three bits, plus all the remaining bits in the instruction group, are used to specify the offset. The magnitude of the offset is thus utterly dependent on the alignment of the instruction within a word. Figure 3 demonstrates how branch offsets of 3, 11, 19, and 27 bits are encoded, depending on the relative location of the branch within a word.

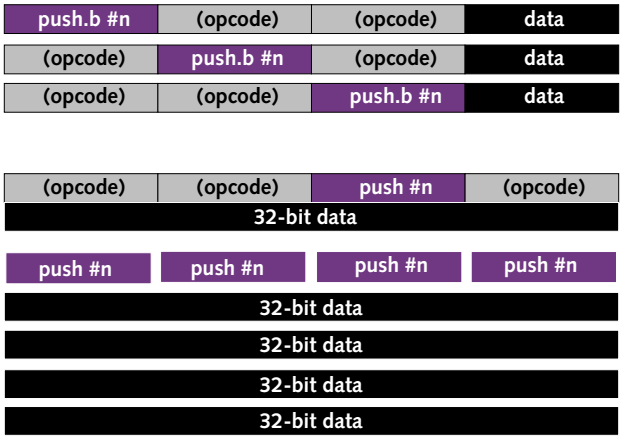

Figure 2. ShBoom processors implicitly encode literal byte values by placing them at the end of a group of instructions.

Why the bizarre addressing scheme? The simple answer is simplicity. Patriot's designers wanted to avoid the complexity of byte-steering logic, so the chip was designed for right-justified values. Compilers, and even the assembler, automatically justify these values, so the peculiar alignment characteristics of the PSC1000 are largely invisible.

Floating-Point Support Included

Although the PSC1000 does not have an FPU or traditional floating-point instructions, Patriot realized that many industrial and motion-control applications perform limited floating-point calculations. The processor, therefore, includes some floating-point "support" instructions that ease the task of converting FP values to integers and back.

The chip includes 11 instructions to normalize, denormalize, round, and check boundary conditions for single- and double-precision IEEE-754 numbers. By conditioning FP numbers with these functions, programs can more efficiently implement FP libraries. The chip's mode register includes 13 bits to control FP rounding, precision, and exceptions. All in all, the chip provides considerably better FP support than most integer processors.

External Bus Includes DRAM Controller

The PSC1000's external interface includes a multiplexed 32-bit address/data bus and a programmable memory controller. The chip drives RAS, CAS, and other signals to four groups of memory devices according to user-selected timing parameters. The memory-control signals can be programmed to accommodate page-mode DRAM, VRAM, SRAM, or ROM devices; device types can be mixed among the four groups.

Because the processor has no on-chip cache or memory, its performance is at the mercy of the system memory. ShBoom's one-byte instructions work in its favor here, allowing the chip to fetch four instructions in a single bus cycle. But consuming one of those instructions on every clock forces the processor to fetch another word every 80 ns (at 50 MHz) or stall the processor. Disregarding the effects of microloops and data accesses, the PSC1000 needs to be fed at a constant 50 Mbytes/s to reach optimum performance.

The overhead in the chip's memory controller prevents most systems from maintaining this rate. Even fast page-mode DRAMs with 30-ns column-access times will decrease

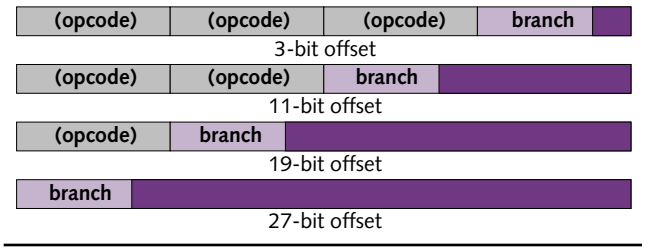

Figure 3. Like data constants, offset addresses are right-justified in a four-byte group of ShBoom instructions.

performance to about 82% of optimal, according to the chip's designers. Faster (and far more expensive) 12-ns SRAMs allow a 50-MHz chip to run at full speed. Keeping up with a proposed 75-MHz implementation would be problematic without reworking the memory interface.

Before the end of the year, Patriot hopes to deliver a derivative of the PSC1000 with an enhanced memory controller that reduces the overhead of DRAM accesses. Another version with on-chip cache is also being considered. Because the part is so memory dependent, speeding up the controller could significantly improve the chip's overall performance. Customers could instead choose less expensive memories and maintain current performance levels.

Separate I/O Processor Runs Concurrently

The PSC1000 has still more tricks up its sleeve. In keeping with Patriot's emphasis on industrial and control-oriented applications, the chip includes an independent I/O processor (IOP), shown in Figure 4, that executes its own code in parallel with the main CPU. The IOP is fairly straightforward and has a simple instruction set with 12 opcodes geared for timing, control, and pulse-width-modulation (PWM) functions.

The IOP is not stack-based, nor does it have its own register set. Instead, it uses g1–g15 from the PSC1000's global registers. The IOP uses these registers to communicate with the CPU and to pass values back and forth between them.

Like the main CPU, the IOP executes one instruction per clock cycle, has a four-instruction buffer, and shares the chip's memory controller and address map. The two processors execute in lockstep from separate instruction streams. Although both cores fetch instructions from the same external memory, their instruction sets are unrelated and incompatible. In concept, Patriot's IOP is similar to Motorola's timing processor unit (TPU) and other autonomous peripherals found on several of that company's 68300-family devices.

The IOP includes only the most basic flow-control, loop, load, and input/output instructions, plus a REFRESH instruction for external DRAM. The two output instructions can toggle any of the chip's eight output pins. At 50 MHz, each pin can be toggled every 20 ns, a considerably finer resolution than most programmable timers allow. Fine motor control and other PWM applications are also practical.

Deterministic Performance Guaranteed

With the IOP and the main CPU sharing memory and some registers, some contention is inevitable. Patriot's goal was to guarantee completely deterministic performance for the IOP, so all arbitration is resolved in favor of the IOP. Technically, no arbitration ever occurs: enough information is available to determine when memory accesses will impact performance and schedule them accordingly.

For example, before every CPU or DMA access to external memory, internal logic first checks the IOP. If it is not about to make a memory request, the CPU or DMA access is allowed to proceed. Otherwise, the CPU or DMA will be stalled until the IOP completes its bus transaction. Because the PSC1000 controls all memory timing, the chip can accurately calculate exactly how long each DRAM access will take and correctly estimate whether a CPU or DMA access to memory will interfere with the IOP.

The IOP's four-instruction buffer prevents it from hogging the PSC1000's bus. Normally, the IOP and the CPU each use 25% of the available bus bandwidth to fetch code; the remainder is free for data transfers. When either the IOP or the CPU is looping, bus utilization is lower.

Design Is Portable Across Fab Processes

Patriot's four-person design team used portable 1.0-micron design rules for the PSC1000. The first generation was built by Oki in 1990; subsequent iterations have been fabricated by National Semiconductor on its 0.8-micron double-metal process. For National's fab, the transistor size was shrunk to 0.8 micron, leaving the metal pitches alone. The shrink resulted in a slightly faster chip, but not a smaller one.

The die, shown in Figure 5, measures 8.7 × 7.7 mm, or 68 mm^2. While not a particularly small die compared with other 32-bit chips, the part's dimensions aren't helped by National's doddering process technology. The MDR Cost Model yields an estimated manufacturing cost of $11, on par with a 68020 or a 960JA.

Initial indications from sample silicon are that the chip will run at 50 MHz, with 75-MHz operation a possibility. At 50 MHz, the part should draw about 250 mW from a 5-V supply. A 75-MHz version would burn roughly 350 mW, or

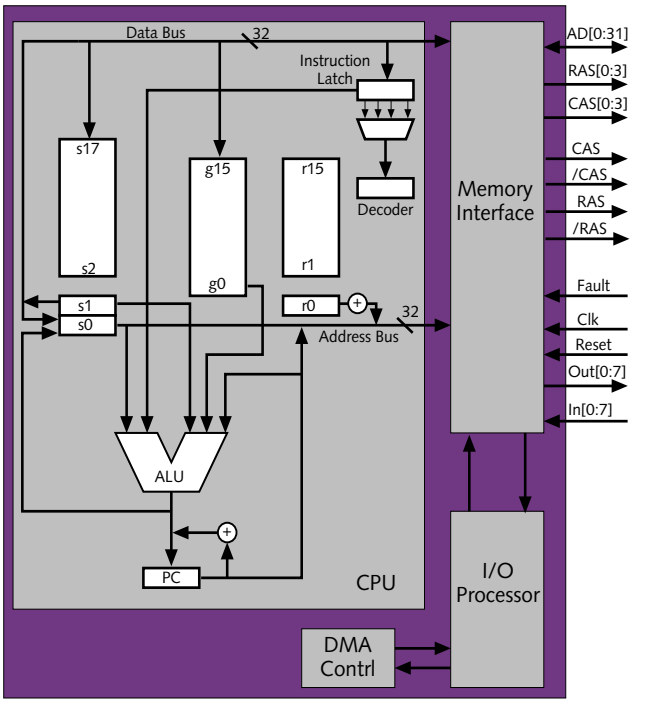

Figure 4. Block diagram of the PSC1000 processor illustrates the chip's division among CPU, I/O processor, and DMA.

Price and Availability

Initial samples of the PSC1000 are available now for beta customers at 50 MHz; general sampling begins in June. In 1,000-unit quantities, the 50-MHz PSC1000 is priced at $20. For more information, contact Patriot Scientific (Poway, Calif.) at 619.679.4428; fax 619.679.4429 or access the Web at *www.ptsc.com.*

about 4.5–5.0 mW/MHz. These numbers make the PSC1000 far cooler than any other 5-V part, with about half the power of an ARM610 and just a bit more than that of a 3.3-V PowerPC 403GA, for example.

Chip Complements Stack-Based Languages

For assembly-language programmers, the PSC1000 will most likely demand some adjustment period. C programmers, however, should not be exposed to any underlying architectural differences. Patriot has developed both an assembler and a C compiler; a debugger is expected by midyear.

The assembler takes some steps toward code rationalization. It inserts unconditional SKIP instructions, for example, when required to force alignment of certain instructions. The assembler also accepts conventional register-to-register nomenclature, converting the instruction MOV s0, g5 to the sequence PUSH s0, POP g5.

Patriot initially had high hopes the PSC1000 would make an attractive PostScript processor; the chip's architecture complements the language's stack-oriented design. The design was not ready soon enough, however, to attract any customers, and the company has since set its immediate sights on another stack-oriented language: Java.

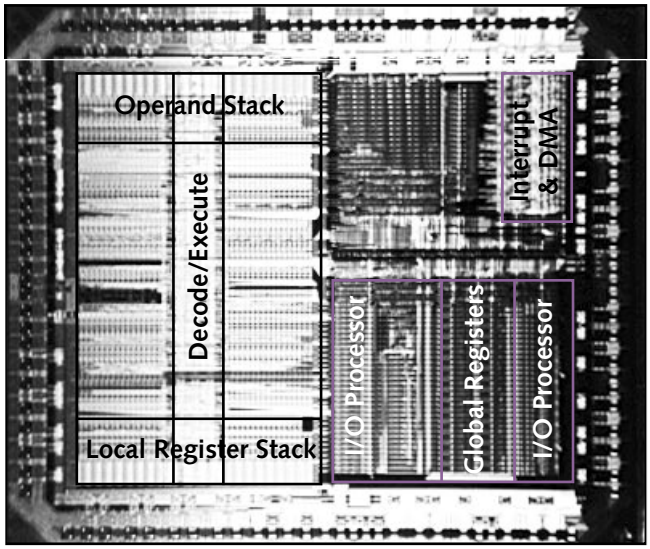

Figure 5. Using 1.0-micron design rules, PSC1000 measures about 68 mm^2 in National's 0.8-micron two-layer-metal process.

Patriot's only publicly announced customer so far is WebBook (Birmingham, Mich.), a company with designs on the nascent Internet-terminal market for its eponymous product. The WebBook is similar to a notebook PC, but with no applications of its own. Instead, it downloads and runs Java applets exclusively. WebBook designers felt that the PSC1000 was ideal for such a system. The company is writing its own Java interpreter, which it will license to Patriot.

Unwilling to put all its eggs in the Internet basket, Patriot is pursuing automotive, robotic, and other motion-control customers. The company will also be marketing an ISDN interface based on a combination of PSC1000 and a 68302. The PSC1000's I/O processor provides considerably finer granularity for timing functions than do most programmable controllers or embedded CPUs. The IOP's guaranteed deterministic performance and simple instruction set make the chip more straightforward to program than Motorola's complex TPU, another possible bonus.

On the other hand, industry support for ShBoom is nonexistent, and a company with Patriot's resources cannot afford to subsidize third-party tool development. The company will have to depend on adventurous, resourceful customers who are attracted either by the chip's programming model or by its I/O capabilities. The design is portable, so a licensing agreement with a fab partner is a possibility, as is selling the design outright to a semiconductor vendor.

Fresh Approaches Are Sometimes Valuable

Patriot has bucked design convention by embracing a long-established, but little-used, stack-based architecture. It's an unusual approach that most companies would not have taken. Apart from AT&T's Hobbit *(see* **061403.PDF** *)* and SGS-Thomson's ST20 Transputer *(see* **091003.PDF** *)*, very few stack-based microprocessors have been developed for commercial use. Patriot believes the unique benefits of post-fix notation, tight object code, and deterministic I/O performance will help its ShBoom chips eke out a niche in the crowded embedded market.

The company may be right, but the hurdles are formidable. Granted, the PSC1000 isn't like most other microprocessors; neither is Hitachi's successful SuperH. But Hitachi and other vendors have the considerable advantage of huge manufacturing capabilities, worldwide marketing forces, multimillion-dollar development budgets, and loyal customers—none of which Patriot can claim. In a crowded market, it will be very tough indeed for an upstart with a unique product and no track record to make any headway.

New devices like Internet terminals are just taking shape, however, and the needs of that market (if, indeed, any ever develop) have not been established. With new applications come new criteria for success, and ShBoom might be just the fresh approach some of these applications need. M