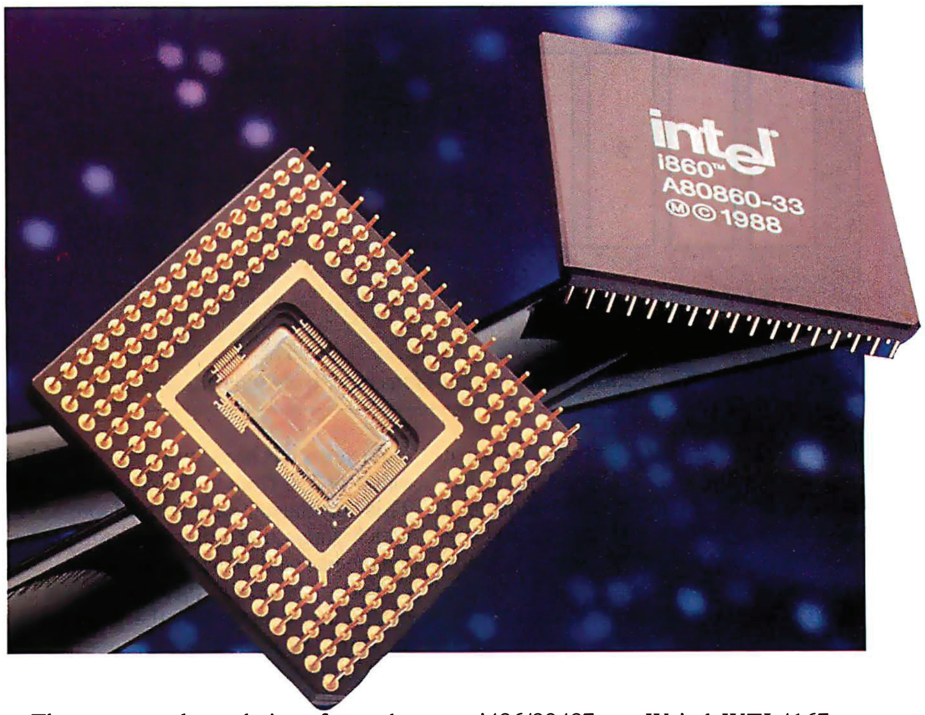
UNDER THE HOOD ■ Stephen S. Fried

Personal Supercomputing with the Intel i860

A close look at Intel's RISC-based "Cray on a chip"



You've heard the old adage: Most people use only 15 percent of their brainpower. Geniuses, however, tap into the 85 percent reserve. Well, I don't know if that's really true for humans, but it certainly applies to computers.

The key to great performance is a CPU architecture that keeps transistors in constant and productive use. The i860 has that ability. Master its memory systems and integer and floating-point processors, then make all these pipelined units run in parallel, and you're well on the way to PC supercomputing.

Who needs personal MFLOPS (millions of floating-point operations per second)? The scientists and engineers who (like me) grew up on the IBM 7094s of the sixties and 370s of the seventies, and who today use VAXes, Crays, and IBM 3090s—that's who. In particular, anyone who studies three-dimensional physical systems develops an insatiable lust for numeric horsepower. Of course, all sorts of nontechnical professionals—bankers, architects, economists, filmmakers, brokerage firms, meteorologists—also benefit from supercomputers more than most people suspect.

Before the arrival of supercomputers, you had to use simplifying assumptions to reduce 3-D problems to two dimensions. But when it comes to analyzing something like the precise flow of air over an airplane in flight, there's no substitute for true 3-D analysis. And the payoff can be dramatic. Shaving even a fraction of a point off the drag coefficient of a new airliner will result in immense fuel savings over the lifetime of a fleet of 500 planes.

The most popular technique for studying real-world objects is called finite-element analysis. In FEA, you "mesh" an object with a polygon grid to create an armature of elements. Then you solve for some property while applying constraints to each element.

Whether the property under investigation is an electrical, thermal, fluid-flow, or stress field, the problem always boils down to the same thing: solving a linear equation that contains a matrix of coefficients. These matrices often have dimensions in excess of 10,000 by 10,000 elements and can consume hundreds of megabytes of disk storage. Of course, matrix math also plays a key role in 3-D graphics. Zooming, rotating, translating, and clipping all rely on matrix operations.

These are just the operations at which the i860 can excel. But it doesn't happen automatically. In scalar mode, as you'll see, the i860 doesn't do much better than an i486/80487 or a Weitek WTL4167.

To attain peak performance, you have to exploit the chip's pipelining and parallel-processing capabilities. Figure 1 shows the i860 with its recommended memory subsystem. The architecture is of the Harvard type, with separate instruction and data caches. Instructions feed out of a 4K-byte, 64-bit-wide cache that can drive both the CPU ("RISC core") and FPU simultaneously through independent 32-bit instruction buses—that's one flavor of i860 parallelism.

Data feeds out of an 8K-byte, 128-bit-wide cache that can drive two long real arguments at a time at the adder, multiplier, or graphics unit. (The processor also has 32 integer registers and 32 floating-point registers, each 32 bits wide.) The adder is a three-stage pipeline, as is the multiplier, and these two units can hook together in a variety of ways. That's another form of parallelism.

To make the i860 hit full stride on a

PHOTOGRAPHY COURTESY OF INTEL CORP.

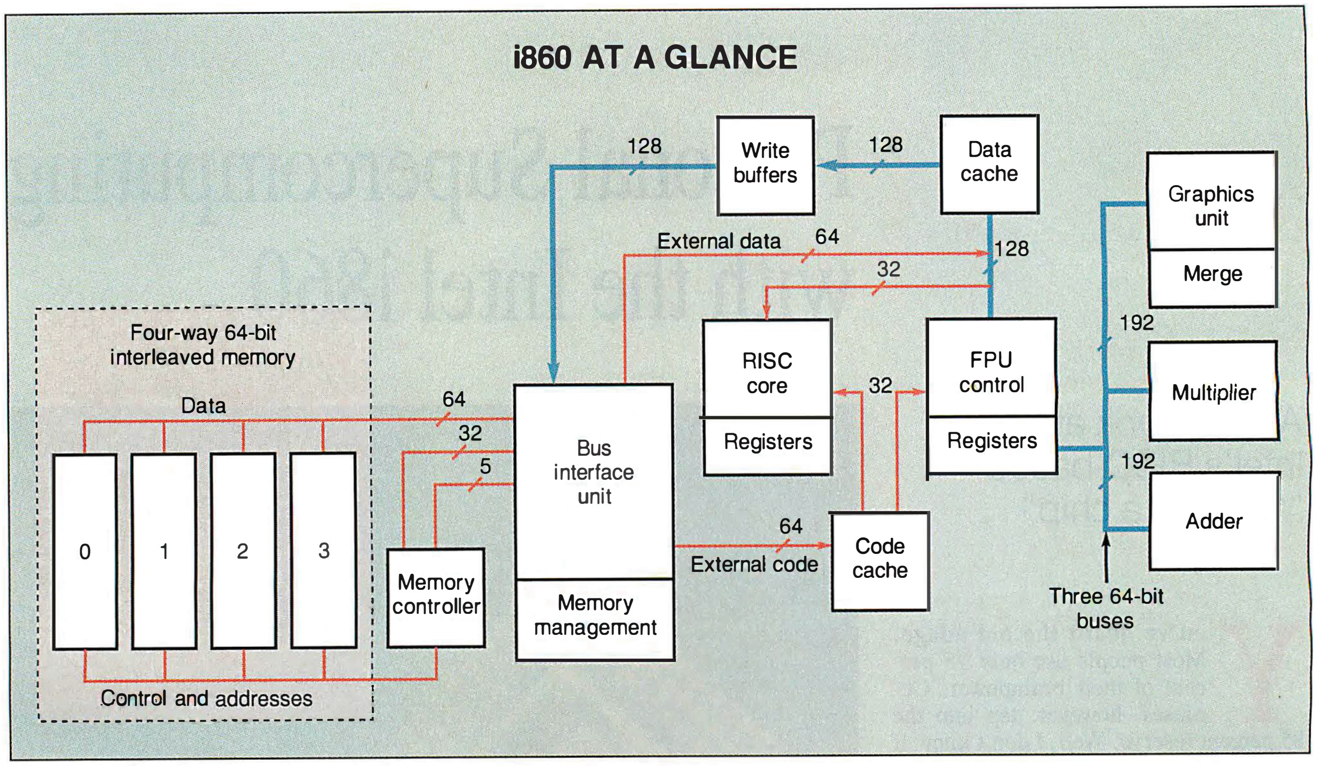


Figure 1: *The i860 features a Harvard architecture, wide internal data paths, pipelined arithmetic units, and a graphics unit.*

numeric problem, as you'll see, you have to get the CPU and FPU working at the same time, feed and flush the adder and multiplier pipelines efficiently, and exploit the adder/multiplier synergy. Once you see how that's done, you'll understand better why the i860 has the architecture that it does. (For a more complete review of the i860 architecture, see "The Intel 80860," December 1989 BYTE.)

The i860 in Scalar Mode

I normally use a benchmark that I call the "Whetscale" (a variation of the Whetstone) to compare scalar numeric devices. Because scalar operations aren't repetitive and can't be pipelined, the Whetscale doesn't give the i860 a chance to really stretch its legs.

The i860's raw speed is impressive. Table 1 shows the Whetscale results for a variety of numeric devices. The improvement in scalar performance over the last eight years has been stunning—roughly a factor of 250 from the 80287 to the i860. This correlates nicely with Moore's law, which states that semiconductor performance has been doubling annually.

Notice, however, that the i860 is not dramatically faster than the Weitek 4167 at single-precision scalar operations. That shouldn't be a surprise. The state of the art in fast FPUs has been at or below the 100-nanosecond mark for the last several years.

A number of chip companies, including Weitek, Analog Devices, and Texas Instruments, have made a business of selling special floating-point data paths for minicomputers to companies such as Sun Microsystems and Alliant. The challenge now is not simply to build faster data paths (i.e., the parts of the device that carry out the arithmetic), but to organize the remainder of the system so that it is able to feed the data paths as fast as they consume numeric data.

It brings to mind the hot-rod speedshop business. The first solution to building faster cars is bigger engines. Soon everyone has huge engines running in cars that can't make corners. The speed problem then becomes one of cornering and, after that's mastered, of reducing aerodynamic drag.

Weitek was the first of the "engine" companies. It started out building 16-bit flash multipliers and expanded into a complete line of FPUs.

Intel, recognizing that none of its OEMs was likely to take the Sun approach to incorporating Weitek support, contracted with Weitek for a chip that glued Weitek engines to a 386 using a memory-mapped interface. That worked as a stopgap measure: Intel could claim Weitek performance for its 386 line and compete with RISC machines while maintaining DOS compatibility.

WHETSCALES

Table 1: *These are the "Whetscales" for Intel and Weitek numeric devices, in MFLOPS. Note the 250-fold speedup from the 80287 to the i860.*

	80287 10 MHz	80287 20 MHz	80387SX 20 MHz	80387DX 20 MHz	3167 20 MHz	80387 33 MHz	3167 33 MHz	i486 25 MHz	4167 25 MHz	i860 33 MHz
Single-precision	0.061	0.185	0.612	0.615	2.27	1.61	4.05	3.31	9.95	12.36
Double-precision	0.051	0.133	0.554	0.560	2.00	1.43	3.57	2.94	7.71	12.36

WHETMATS

Table 2: *These are the "Whetmats" for Intel and Weitek numeric devices, in MFLOPS. Even in scalar mode, the i860's raw speed gives it an edge over the Weitek 4167.*

	80287 10 MHz	80287 20 MHz	80387SX 20 MHz	80387DX 20 MHz	3167 20 MHz	80387 33 MHz	3167 33 MHz	i486 25 MHz	4167 25 MHz	i860 33 MHz
Single-precision	0.028	0.181	0.282	0.378	1.32	0.866	2.56	1.87	4.55	5.88
Double-precision	0.024	0.059	0.204	0.328	0.62	0.672	1.12	1.70	1.93	4.91

WHETMAT/WHETSCALE RATIO

Table 3: *The Whetmat/Whetscale ratio describes how well a processor copes with the addressing overhead associated with vector operations. For single-precision work, the 4167 and the i860 have comparable vector efficiencies, but in double-precision the i860's 64-bit external data bus pulls it significantly ahead of the 4167.*

	80287 10 MHz	80287 20 MHz	80387SX 20 MHz	80387DX 20 MHz	3167 20 MHz	80387 33 MHz	3167 33 MHz	i486 25 MHz	4167 25 MHz	i860 33 MHz
Single-precision	0.460	0.441	0.469	0.614	0.581	0.537	0.632	0.565	0.457	0.475
Double-precision	0.470	0.440	0.368	0.585	0.307	0.469	0.313	0.578	0.250	0.397

Intel got busy back in the "frame" shop building a device that could properly take advantage of today's wide numeric data paths. Think of the Whetscale as a drag race. Both the 4167 and the i860 have plenty of what it takes to post a good mark: good compilers and brute force. But when it comes to the Le Mans of the numerics business—double-precision vector operations (as exemplified by the LINPACK benchmark)—the i860, with its 160-MB-per-second memory interface, runs the course as much as 10 times faster than a 4167-equipped i486.

Jacking into the Matrix

I use a second benchmark, called the "Whetmat," to evaluate performance on

The DGIS™ SDK and a TI 34010-based High-Performance Graphics Board for one amazing price.

High performance, high resolution graphics are the wave of the future. With the DGIS Software Developer's Kit™ (SDK), qualified software developers can write for the future today.

The DGIS Developer's Kit provides everything needed to develop applications and drivers for DGIS-compatible 34010 graphics boards—boards from companies such as Compaq, Dell, Hewlett-Packard, NCR, NEC, TI and more than 30 others worldwide. Software developed with this kit can access the full power of the 34010, supporting the greatest number of high resolution graphics boards at the highest levels of performance, resolution and color.

DGIS, the premier and most widely-shipped interface for the TI 340X0 family of graphics coprocessors, provides an outstanding feature-rich programming model with 100+ graphics functions. The DGIS SDK includes documentation and language bindings for the DGIS interface, device drivers for Windows 3.0, utilities, and the GSS AT1050™ 1024X768 34010 graphics board (which normally sells for $1295 alone).

The DGIS SDK is compatible with most C compilers and supports the XMS standard as well as DOS Extenders from Rational and Phar-Lap.

Stepping up to the big screen has never been easier or more attractive. Call today.

THE POWER OF HIGH RESOLUTION GRAPHICS PROGRAMMING CAN BE REACHED WITH ONE EASY NUMBER:

Graphic Software Systems
A Subsidiary of SPECTRAGRAPHICS

Call (503) 641-2455. Ask for Dept. DGIS-1.

All prices subject to change without notice. GSS, DGIS, The DGIS Software Developer's Kit, GSSAT1050 are trademarks of Graphics Software Systems Inc. All other trademarks belong to their respective owners.

Circle 122 on Reader Service Card (RESELLERS: 123) a typical vector operation—a matrix multiplication. The Whetmat, in conjunction with the Whetscale, gives you a way to measure the relative efficiencies of scalar and vector operations.

On scalar processors, vector operations run slower than scalars for two reasons: They have to access operands from memory instead of registers, and they have to compute the address of each operand as it is used. Table 2 shows raw Whetscale results, and table 3 displays "vector efficiency"—that is, Whetmats divided by Whetscales, which I take as a measure of how effectively a scalar processor copes with the addressing overhead of vector problems.

I'm still restricting the i860 to scalar mode, but even without the advantages of pipelining and parallelism, notice how the i860 begins to distinguish itself from the i486 and 4167. The i860 continues to perform well on the double-precision Whetmat, while the i486 and the 4167 are hardly better than an i486 running on its internal FPU. Moreover, the i860 outdoes the Weitek devices in terms of double-precision vector efficiency.

The problem with the 4167 is that, for large matrices, it's bound by the data bus. (You see the same thing happening with the 80387SX, which keeps up with its DX cousin on the Whetscale but falls behind on the Whetmat.) What turns out to be the biggest asset of the i860 for vector operations performed in scalar mode is its 64-bit-wide external data bus.

If the 4167 were attached to the i486 with a 64-bit-wide bus, you could drive a double-precision, memory-accessing operation with two lines of i486 code (instead of the four that it actually requires) and thereby double the 4167's performance for certain vector operations.

Even in scalar mode, then, the i860's raw speed and wide external data bus give it a significant edge over competing numeric devices. But 4.91 double-precision MFLOPS falls far short of the chip's rated maximum: 66 MFLOPS (at 33 MHz). How do you get the i860 to live up to its full potential?

Henry Ford Had the Right Idea

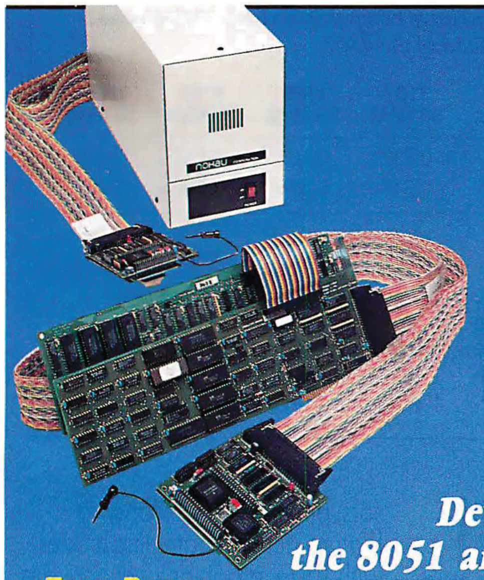
A pipelined processor works just like one of Ford's assembly lines. The i860 has four of them, and you can use them or not, depending on the problem and how you decide to code it. The four pipelines are the external memory loader, the adder, the multiplier, and the graphics unit.

The adder and multiplier can load from registers, the data cache, or external memory. The ability to pipeline loads from external memory is crucial for vector operations on large matrices.

The adder and the multiplier, both of which are three-stage pipelines, are available to both scalar and pipelined instructions. In scalar mode, these units produce a result every three cycles. But pipelined instructions can control the units on a cycle-by-cycle basis, producing new results each cycle.

The amount of work you can get out of the i860 pipes is simply the speed of the pipes times the number of pipes in operation. At 33 MHz, with both the adder and multiplier yielding new results each cycle, that's 66 MFLOPS!

Of course, the problem that is being solved must require some sequence of additions and multiplications that alternate, so the adder and multiplier can work together. But that's not as artificial as it may seem. Vector dot products, which are the core of other vector operations, such as matrix multiplication, have



8051 & 68HC11

PC-Based In-Circuit Emulators

Nohau Covers All Your Development Needs for the 8051 and 68HC11 Families!

Free Demo

You can start your debugging with this **FREE** demo simulator. You can load up to 512 bytes of code, assembler, C, or PL/M and do full debugging/simulation in assembly and source level. A great way to get started for **FREE.** Fantastic for schools! Just call and we'll send it!

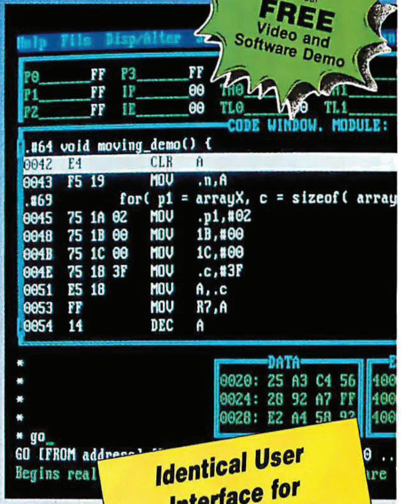
Call Today for Your **FREE** Video and Software Demo

Full Simulator

The full-blown simulator is an extension of the DEMO. You can load up to 64K of code and use 64K of XDATA space. You can program an "external environment" to interact with your code to simulate your target system. The emulator is the hardware extension of the simulator!

In-Circuit Emulation

The 24MHz real-time emulator has been the industry standard for years. With its complex breakpoint logic and advanced trace, nobody can beat it for performance. Plug-in or RS-232 configuration. All 8051 derivatives are supported!



```
P0______FF  P3______FF  TH0
P1______FF  IP______00
P2______FF  IE______00  TL0  TL1
.#64 void moving_demo() {
0042  E4          CLR   A
0043  F5 19       MOV   .n,A
.#69        for( p1 = arrayX, c = sizeof( array
0045  75 1A 02    MOV   .p1,#02
0048  75 1B 00    MOV   1B,#00
004B  75 1C 00    MOV   1C,#00
004E  75 18 3F    MOV   .c,#3F
0051  E5 18       MOV   A,.c
0053  FF          MOV   R7,A
0054  14          DEC   A
*                        0020: 25 A3 C4 56  400
*                        0024: 28 92 A7 FF  400
*                        0028: E2 A4 5A 92  400
*
* go_
GO [FROM address]                          0 ..
Begins real                                 re
```

Identical User Interface for All Three Products — You Can't Go Wrong!

nohau CORPORATION
51 E. Campbell Avenue, Campbell, CA 95008
(408) 866-1820 • FAX (408) 378-7869

Australia (02) 654 1873, Austria (0222) 38 76 38, Benelux +31 1858-16133, Canada (514) 689-5889, Denmark (42) 65 81 11, Finland 90-452 1255, France (01)-69 41 28 01, Germany 08131-25083, Great Britain 0962-73 31 40, Hungary 01-137 2182, Israel (03) 48 48 32, Italy (011) 771 00 10 Korea (02) 784 784 1, New Zealand (09) 392-464, Portugal (01) 81 50 454, Sweden, Norway (040) 92 24 25, Singapore (065) 284-6077, Spain (93) 217 2340, Switzerland (01) 740 41 05, Taiwan (02) 7640215, Thailand (02) 281-9596, Yugoslavia 061-57 19 49.

← *Circle 221 on Reader Service Card*

exactly this sort of interleaved add/multiply behavior.

Given the right sort of problem, you've got to arrange to keep your numeric factory fed with numbers, and to get rid of the results as fast as they come out the back end. Here, the i860's Harvard architecture comes into play.

The 64-bit external data bus can shuffle 8 bytes of data to or from memory every other cycle—that's 4 bytes per cycle, or 160 MBps. To start a single-precision dot product on every cycle—and thereby keep the load and numeric pipelines fed—you will have to read one operand from memory while grabbing the second operand from a register or out of the cache.

The i860's 8K-byte data cache can hold entire rows when multiplying matrices as large as 2000 elements. As a matter of fact, the i860 is said to be operating in "Cray" mode when its cache emulates the vector registers of a Cray. That's feasible because the i860 can move data between its cache and the FPU's register file at a whopping 640 MBps. Moreover, the two kinds of loads—pipelined loads from external memory straight into registers, and cached loads that fill the "vector register"—can proceed in parallel.

Wiring for Dual-Operation Mode

I've said that the i860 supports two forms of parallelism. In *dual-operation mode*, the adder and multiplier work in concert. In *dual-instruction mode*, the RISC core loads floating-point registers while the FPU runs in parallel. The two modes are complementary; I'll tackle dual-operation mode first.

Before you can understand "dual-ops," though, let me review basic pipelining. Tables 4a and b show the pipelined multiplication of two arrays of single-precision floating-point numbers. As you can see, it's a series of instructions of the form

```
pfmul.ss src1, src2, dest
```

where the `p` in `pfmul` selects pipelined mode, and the `.ss` specifies single-precision operands and a single-precision result. The special register `f0` acts as a dummy destination for the first three instructions, while the pipeline fills. Thereafter, each instruction yields a result that began its trip through the pipeline three instructions ago. At the end, register `f0` acts as a placeholder again, this time supplying dummy operands to flush the last three results out of the pipeline.

Now, a vector dot product boils down to a sequence of operations like this:

REGISTER SETUP

Table 4a: *Floating-point registers f4 to f11 hold the first array, and registers f12 to f19 hold the second array. Results appear in registers f12 to f19 after a three-cycle delay.*

src1	src2	Destination
f4	f12	f12
f5	f13	f13
f6	f14	f14
f7	f15	f15
f8	f16	f16
f9	f17	f17
f10	f18	f18
f11	f19	f19

PIPELINED MULTIPLICATION IN ACTION

Table 4b: *Once you prime the multiplier, it produces a new result each cycle (G=garbage).*

	Multiplier			
Instruction	Stage 1	Stage 2	Stage 3	Result
pfmul.ss f4,f12,f0	f4×f12	G	G	None
pfmul.ss f5,f13,f0	f5×f13	f4×f12	G	None
pfmul.ss f6,f14,f0	f6×f14	f5×f13	f4×f12	None
pfmul.ss f7,f15,f12	f7×f15	f6×f14	f5×f13	f12<—f4×f12
pfmul.ss f8,f16,f13	f8×f16	f7×f15	f6×f14	f13<—f5×f13
pfmul.ss f9,f17,f14	f9×f17	f8×f16	f7×f15	f14<—f6×f14
pfmul.ss f10,f18,f15	f10×f18	f9×f17	f8×f16	f15<—f7×f15
pfmul.ss f11,f19,f16	f11×f19	f10×f18	f9×f17	f16<—f8×f16
pfmul.ss f0,f0,f17	G	f11×f19	f10×f18	f17<—f9×f17
pfmul.ss f0,f0,f18	G	G	f11×f19	f18<—f10×f18
pfmul.ss f0,f0,f19	G	G	G	f19<—f11×f19

$(1\times2)+(3\times4)+(5\times6)+\ldots$

For this job, you'd need to interleave addition and multiplication. There are 62 ways to chain together the i860's adder and multiplier. Figure 2a shows the full set of possibilities. The adder, for example, can receive operands from floating-point registers, the special T (temporary) register, the multiplier, or itself.

To perform the dot product, combine the adder and multiplier to create a "multiply-accumulate" instruction, which, as shown in figure 2b, recirculates the adder's results back through the adder.

Take a look at the dot product example in tables 5a and b. In table 5b, the cryptic `m12apm.ss` is the multiply-accumulate instruction. It wires the FPU so that the multiplier gets two operands from registers, and so that the adder's two operands are the multiplier's result and its own prior result. During the priming phase, you fill up the multiplier with the first three product terms: `1×9`, `2×10`, and `3×11`. When terms reach stage 3, you start referring to them by their value.

By the fourth instruction, you have a problem. The third stage of the adder is about to feed back into the adder's first stage and get added to the garbage value there. Instructions 4 through 6 therefore prime the adder with 0s using pipelined additions involving the dummy register `f0`. Since `pfadd.ss` is a pipelined operation, it will take three cycles to complete. The `pfadd.ss` instructions affect only the adder; the values in the multiplier are untouched.

Now you can enter the steady-state part of the algorithm. After another three-cycle latency, during which the adder combines multiplier results with the 0s in its pipeline, the adder begins its real work: accumulating partial sums in each of its stages.

Figure 3 shows what's happening in a more graphical way. For the first six cycles of the journey through the pipeline's stages, all terms progress from left to right, just like on an assembly line. Then the pattern abruptly reverses, as adder results feed back to the first stage of the adder. In a real program, the steady-state part would be a loop with dozens or hundreds of operand pairs.

When all the product pairs have been fed in from their registers, start harvesting the sums. First, flush the multiplier with 0s. You use the same multiply-accumulate instruction, since, while you're flushing the multiplier, you want the adder to keep accumulating sums.

After three cycles, the multiplier's job is done. The adder contains partial sums in each of its stages; once you combine these, you'll have the answer. You could load them off to three registers and then use scalar operations to combine them, but this would cost at least three cycles to unload the pipes plus nine cycles to perform the three additions. Instead, use a series of pipelined additions with a single scalar addition, for a total of eight cycles.

The final code sequence in table 5b is worth a close look. Begin by taking what was in the adder pipe at the end of the last

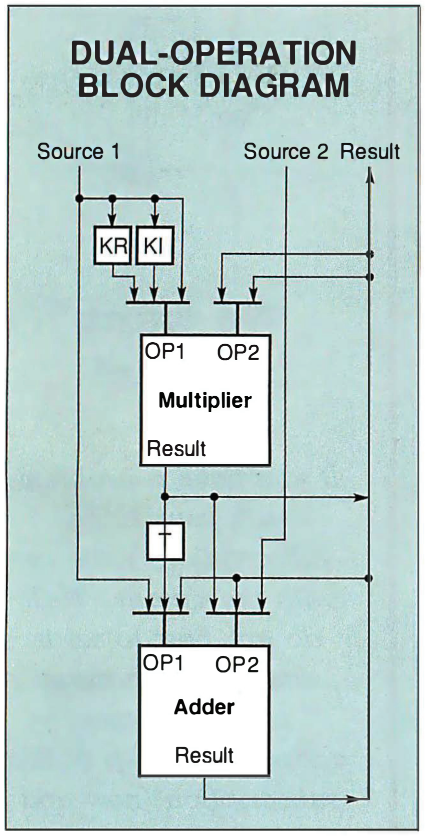


Figure 2a: *The multiplier and adder can receive inputs from registers, their own outputs, or each other's outputs. They can be wired 62 different ways to create special-purpose instructions, such as "multiply-accumulate."*

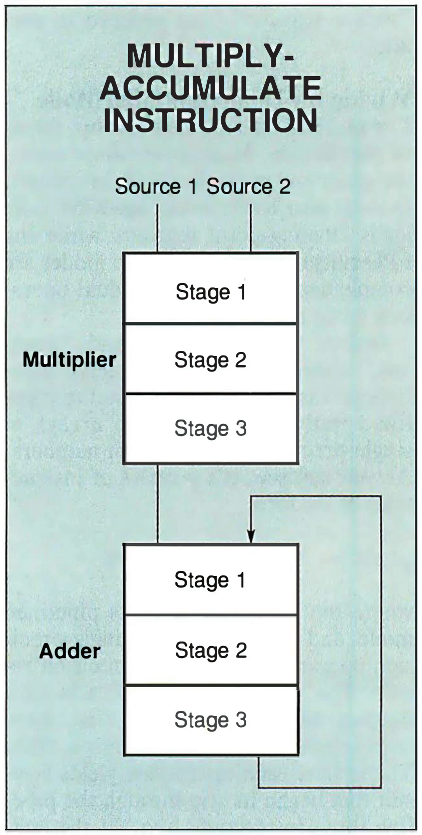
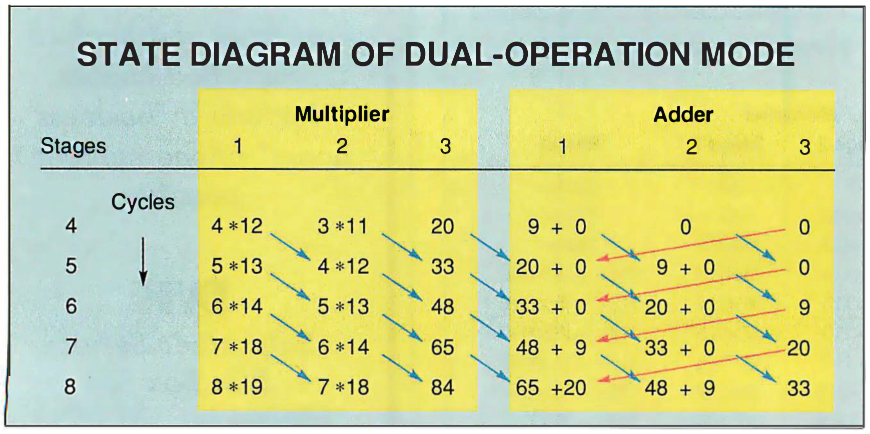


Figure 2b: *The multiply-accumulate instruction is formed by combining the adder and the multiplier. The multiplier feeds the adder, while the adder's results recirculate back through the adder.*



STATE DIAGRAM OF DUAL-OPERATION MODE

	Multiplier			Adder		
Stages	1	2	3	1	2	3
Cycles						
4	4 *12	3 *11	20	9 + 0	0	0
5	5 *13	4 *12	33	20 + 0	9 + 0	0
6	6 *14	5 *13	48	33 + 0	20 + 0	9
7	7 *18	6 *14	65	48 + 9	33 + 0	20
8	8 *19	7 *18	84	65 +20	48 + 9	33

Figure 3: *After seven cycles, partial sums begin to accumulate as the adder's results recirculate back through the adder.*

multiply-accumulate, 117, and put it in register f20. The next instruction should strike you as bizarre. It appears to place f20 and f21 into the pipeline, while putting the stage 3 adder result into f21. But this is the first use of f21. How can the result you're about to place in f21 also be the same operation's input value?

Simple: The i860 works backward. Its internal clock breaks pipelined operations into three parts. On cycle 1 of the internal clock, the last stage of the adder gets stored to the destination. On cycle 2, stages 1 and 2 advance to stages 2 and 3. On cycle 3, the inputs latch into stage 1.

This backward way of doing things actually makes a lot of sense, as it starts off the most time-consuming part of the process (storing results) early. It's also what makes it possible for results to recirculate back through the adder with a single instruction.

Prepare for Lift-off: Entering Dual-Instruction Mode

Until now, the assumption has been that operands are simply available in registers. To load those registers without stalling the pipeline, you'll have to tackle the second form of i860 parallelism: dual-instruction mode.

In that mode you will be doing pipelined loads and pipelined computation at the same time. But, again, let's start by looking at a simple pipelined load. The memory subsystem uses a three-stage

REGISTER SETUP

Table 5a: *Registers f4 to f11 contain the first array, and f12 to f14 contain the second.*

src1	Value	src2	Value
f4	1.0	f12	9.0
f5	2.0	f13	10.0
f6	3.0	f14	11.0
f7	4.0	f15	12.0
f8	5.0	f16	13.0
f9	6.0	f17	14.0
f10	7.0	f18	15.0
f11	8.0	f19	16.0

MULTIPLY-ACCUMULATE IN ACTION

Table 5b: *During the steady-state part of the algorithm, each instruction drives the three stages of the multiplier and the three stages of the adder in parallel (G=garbage).*

	Multiplier stages			Adder stages			Result
Priming: Fill multiplier with first three products							
m12apm.ss f4,f12,f0	1×9	G	G	G	G	G	Ignore
m12apm.ss f5,f13,f0	2×10	1×9	G	G	G	G	Ignore
m12apm.ss f6,f14,f0	3×11	2×10	1×9	G	G	G	Ignore
Priming: Prepare adder for first product							
pfadd.ss f0,f0,f0	3×11	2×10	1×9	0	G	G	Ignore
pfadd.ss f0,f0,f0	3×11	2×10	1×9	0	0	G	Ignore
pfadd.ss f0,f0,f0	3×11	2×10	1×9	0	0	0	Ignore
Steady state							
m12apm.ss f7,f15,f0	4×12	3×11	20	9+0	0	0	Ignore
m12apm.ss f8,f16,f0	5×13	4×12	33	20+0	9+0	0	Ignore
m12apm.ss f9,f17,f0	6×14	5×13	48	33+0	20+0	9	Ignore
Now the first product term feeds back to the adder							
m12apm.ss f10,f18,f0	7×18	6×14	65	48+9	33+0	20	Ignore
m12apm.ss f11,f19,f0	8×19	7×18	84	65+20	48+9	33	Ignore
We've multiplied all terms, now flush the multiplier							
m12apm.ss f0,f0,f0	0×0	8×18	126	84+33	65+20	57	Ignore
m12apm.ss f0,f0,f0	0×0	0×0	152	126+57	84+33	85	Ignore
m12apm.ss f0,f0,f0	0×0	0×0	0×0	152+85	126+57	117	Ignore
Combine adder stages and store result							
pfadd.ss f0,f0,f20	G	G	G	0+0	152+85	183	f20<—117
pfadd.ss f20,f21,f21	G	G	G	183+117	0+0	237	f21<—183
pfadd.ss f0,f0,f20	G	G	G	0+0	183+117	0	f20<—237
pfadd.ss f0,f0,f0	G	G	G	0+0	0+0	300	f0<—0
pfadd.ss f0,f0,f21	G	G	G	0+0	0+0	0+0	f21<—300
fadd.ss f20,f21,f20	G	G	G	G	G	G	f20<—537

Listing 1: *Note the use of both pipelined* (`pfld`) *and scalar* (`fld`) *load instructions. Pipelined loads are appropriate when you're going to use an operand once and then throw it away. Use scalar loads when you want operands to get stored in the cache.*

```
// Multiply eight elements of row A by column B.
// Row A is contained in registers f4..f11.
// Row B is contained in registers f12..f19.

inner::
d.m12apm.ss     f4,f12,f0      //Start f4*f12 into multiply-accumulate pipe.
        fld.q   16(r29)++,f8   //Load 4 elements of A into f8..f11
                               //from cache, and increment r29 by 16.
d.m12apm.ss     f5,f13,f0      //Start f5*f13 into multiply-accumulate pipe.
        pfld.d  8(r24)++,f16   //Load third stage of pipe into f16,f17
                               //and increment f24 by 8.
d.m12apm.ss     f6,f14,f0      //Continue with multiply-accumulate pipe.
        pfld.d  8(r24)++,f18   //Load and service B pipeline.
d.m12apm.ss     f7,f15,f0      //Continue with multiply-accumulate pipe.
        fld.q   16(r29)++,f4   //Load A now for use at top of loop!
d.m12apm.ss     f8,f16,f0      //Continue with multiply-accumulate pipe.
        nop                    //Dual-instruction mode always requires pairs.
d.m12apm.ss     f9,f17,f0      //Continue with multiply-accumulate pipe.
        pfld.d  8(r24)++,f12   //Load B now for use at top of loop!
d.m12apm.ss     f10,f18,f0     //Continue with multiply-accumulate pipe.
        bla     r27,r28,inner  //Start branching to the label now!
d.m12apm.ss     f11,f19,f0     //Last multiply-accumulate in inner loop.
        pfld.d  8(r24)++,f14   //Load B for next loop now!
```

pipeline that is controlled by the instructions of the form:

```
pfld.z src1(src2), freg
```

or

```
pfld.z src1(src2)++, freg
   //autoincrement
```

In both forms, src2 provides a base address to which src1 gets added. In the auto-increment mode, each instruction increments src2 by src1; that makes it possible to load arrays with constant stride factors stored in src1.

The z stands for the number of bytes to load into memory: 4 or 8. Because you're working with a three-stage pipeline, the destination register, `freg`, receives the data specified in the third prior `pfld` instruction, not the current one. As you can imagine, it's just about impossible to write pipelined code for the i860 without drawing stage diagrams to visualize what is happening in the pipelines.

In dual-instruction mode, you execute pipelined loads and pipelined add/multiply operations simultaneously. To

BIOS **SOURCE CODE**

The **AT BiosKit** gives you a complete Bios with source code in C you can modify for your own applications! The BiosKit includes a Bios on diskette ready for programming Eproms, and includes the utilities you need to Rom the sources Code. The Bios also has a Rom Monitor/Debug and Setup. At last **you have control** over the core of your system. Over 380 pages, with diskette, $199. The **XT BiosKit** is only $99. **The Intel Wildcard Supplement** for the XT BiosKit is $49. Software tools: You need MS C and MASM 5.1 for modifying the Bios.

FREE We'll include a free copy of the pocket-sized **XT-AT Handbook by Choisser and Foster** with each BiosKit if you mention this ad when you order. Of course, this $9.95 value is also available by itself. Or buy five or more for only $5.00 each.

VISA **800-462-1042** MasterCard
619-271-9526

Annabooks
12145 Alta Carmel Ct., Suite 250
San Diego, CA 92128

FAX 619-592-0061
Money-back guarantee

A MESSAGE TO OUR SUBSCRIBERS

FROM TIME TO TIME WE MAKE THE BYTE subscriber list available to other companies who wish to send our subscribers material about their products. We take great care to screen these companies, choosing only those who are reputable, and whose products, services, or information we feel would be of interest to you. Direct mail is an efficient medium for presenting the latest personal computer goods and services to our subscribers.

Many BYTE subscribers appreciate this controlled use of our mailing list, and look forward to finding information of interest to them in the mail. Used are our subscribers' names and addresses only (no other information we may have is ever given).

While we believe the distribution of this information is of benefit to our subscribers, we firmly respect the wishes of any subscriber who does not want to receive such promotional literature. Should you wish to restrict the use of your name, simply send your request to the following address.

BYTE MAGAZINE
ATTN: SUBSCRIBER SERVICE
P.O. Box 555
HIGHTSTOWN, NJ 08520

accomplish this feat, you exploit the i860's ability to fetch two instructions at once from the instruction cache.

Listing 1 shows the inner loop of a matrix multiplication in dual-instruction mode. The `d.` prefix that precedes each multiply-accumulate instruction tells the processor to execute this floating-point instruction and the following core instruction simultaneously.

Note the use of both pipelined (`pfld`) and scalar (`fld`) load instructions. With pipelined loads, you bypass the cache; that's appropriate for large arrays that you're going to touch just once. Scalar loads fill the cache; that's useful for small matrices that will fit entirely in the cache, or for larger matrices whose rows can be cached.

There are many points of interest in this short piece of code, which takes just eight cycles (200 ns) to execute at 40 MHz. On every cycle, the i860 schedules four or five processor activities. For example, the third and fourth lines of code start the multiplier (and adder) pipes, store the third previous pipelined load to f16 and f17, and increment r24 by a constant stride factor of 8. That means the i860 performs five tasks every 25 ns, or one every 5 ns, which is the equivalent of 200 million operations per second on a conventional system. That's what transistor productivity is all about.

The code has a unique rhythm. The pipelined loads at the head of the loop deliver their goods at the bottom half of the loop, while the loads at the bottom are arranged to feed the top of the loop. The whole loop has the feel of the antique push-pull amplifiers used to power radio transmitters back in the old days.

After rewriting the Whetmat to call a hand-coded matrix multiply like the one that is shown in listing 1, the i860 hit 62 MFLOPS. That's quite close to the theoretical limit of 66 MFLOPS (at 33 MHz), and much faster than the 4.9 MFLOPS the i860 achieves in scalar mode.

The i860 can make your dreams of personal supercomputing come true. My i860-powered Compaq 386/20 portable computer turns in over 10 LINPACK MFLOPS. How good is that? The top-of-the-line VAX 8800 produces 1.2 LINPACK MFLOPS; an IBM 3081K does only slightly better at 2.1. Of course, a Cray X cranks out over 60 LINPACK MFLOPS, but I can't carry one home. ■

Stephen S. Fried is president of MicroWay, Inc. (Kingston, MA), whose products include NDP Fortran-386 and an i860-based coprocessor for PCs. He can be reached on BIX c/o "editors."

"Parlez-vous Q-TEL™?"

Oui...Si...Ja...now the answer is Yes wherever you go internationally–thanks to our *new* Q-TEL International database. Domestically, you've seen how our Q-TEL databases bring you the most current and accurate rate and tariff information. Now Q-TEL International database brings it to you *worldwide*. It has the same superior performance packaging, and easily adapts to your applications.

If you're speaking bottom line savings, parlez with CCMI today. Call **1-800-526-5307 ext.: 290.**

Ask for our new and expanded catalog!

CCMI *First-Rate Information*

McGraw-Hill

IT'S 11:35 p.m.

SOMEONE TWO THOUSAND MILES AWAY JUST SAW YOUR AD...AND WANTS MORE INFORMATION, NOW.

HOW DO YOU SATISFY THIS POTENTIAL CUSTOMER?

TRY THIS DEMO

1. Pick up the phone connected to your FAX machine and dial:
   USA: 708-924-7465 UK: (+44) 672 84242
2. Follow the simple voice instructions, using the phone keypad to respond.
3. You will be asked to press in the code number for FaxFacts.
   USA—Press: 889812# UK—Press: 201#
4. Continue to follow instructions being given over the phone.
5. Once all steps are completed, you will be asked to press the "START" or "RECEIVE" button on your FAX.
6. Within seconds, after pressing this button, you will be given information on this instantaneous literature delivery system.

FaxFacts is a product of Copia International, Inc.

USA: (708) 682-8898
UK: (+44) 672 84535
USA FAX: (708) 665-9841