Introducing the Intel i860 64-Bit Microprocessor

A million-transistor budget helps this RISC deliver balanced MIPS, Mflops, and graphics performance with no data bottlenecks.

Les Kohn
Neal Margulis

Intel Corp.

The single-chip i860 CPU—a 64-bit, RISC-based microprocessor—executes parallel instructions using mainframe and supercomputer architectural concepts. We designed the 1,000,000-transistor, 10 mm × 15 mm processor (see Figure 1 on the next page) for balanced integer, floating-point, and graphics performance, using the company's latest generation CAD tools and 1-micrometer semiconductor process.

To accommodate our performance goals, we divided the chip area evenly between blocks for integer operations, floating-point operations, and instruction and data cache memories. Inclusion of the RISC (reduced instruction set computing) core, floating-point units, and caches on one chip lets us design wider internal buses, eliminate interchip communication overhead, and offer higher performance. As a result, the i860 avoids off-chip delays and allows users to scale the clock beyond the current 33- and 40-MHz speeds.

We designed the i860 for performance-driven applications such as workstations, minicomputers, application accelerators for existing processors, and parallel supercomputers. The i860 CPU design began with the specification of a general-purpose RISC integer core. However, we felt it necessary to go beyond the traditional 32-bit, one-instruction-per-clock RISC processor. A 64-bit architecture provides the data and instruction bandwidth needed to support multiple operations in each clock cycle. The balanced performance between integer and floating-point computations produces the raw computing power required to support demanding applications such as modeling and simulations.

Finally, we recognized a synergistic opportunity to incorporate a 3D graphics unit that supports interactive visualization of results. The architecture of the i860 CPU provides a complete platform for software vendors developing i860 applications.

Architecture overview. The i860 CPU includes the following units on one chip (see Figure 2):

- the RISC integer core,
- a memory management unit with paging,
- a floating-point control unit,
- a floating-point adder unit,
- a floating-point multiplier unit,
- a 3D graphics unit,

0272-1732/89/0800-0015$01.0 © 1989 IEEE

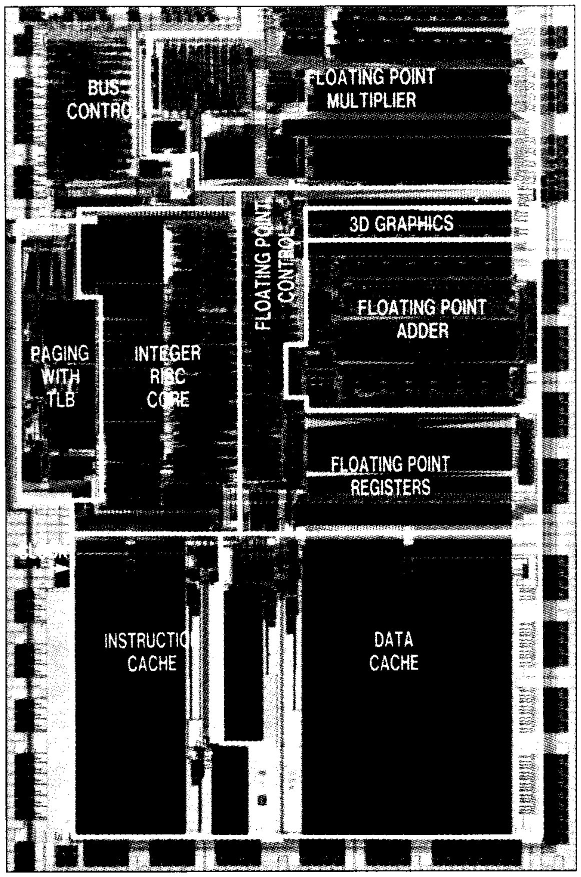


Figure 1. Die photograph of the i860 CPU.

- a 4-Kbyte instruction cache,
- an 8-Kbyte data cache, and
- a bus control unit.

Parallel execution. To support the performance available from multiple functional units, the i860 CPU issues up to three operations each clock cycle. In single-instruction mode, the processor issues either a RISC core instruction or a floating-point instruction each cycle. This mode is useful when the instruction performs scalar operations such as operating system routines.

In dual-instruction mode, the RISC core fetches two 32-bit instructions each clock cycle using the 64-bit-wide instruction cache. One 32-bit instruction moves to the RISC core, and the other moves to the floating-point section for parallel execution. This mode allows the RISC core to keep the floating-point units fed by fetching and storing information and performing loop control, while the floating-point section operates on the data.

The floating-point instructions include a set of operations that initiate both an add and a multiply. The add and multiply, combined with the integer operation, result in three operations each clock cycle. With this fine-grained parallelism, the architecture can support traditional vector processing by software libraries that implement a vector instruction set. The inner loops of the software vector routines operate up to the peak floating-point hardware rate of 80 million floating-point operations per second. Consistent with RISC philosophy, the i860 CPU achieves the performance of hardware vector instructions without the complex control logic of hardware vector instructions. The fine-grained parallelism can also be used in other parallel algorithms that cannot be vectorized.

Register and addressing model. The i860 microprocessor contains separate register files for the integer and floating-point units to support parallel execution. In addition to these register files, as can be seen in Figure 3 on page 18, are six control registers and four special-purpose registers. The RISC core contains the integer register file of thirty-two 32-bit registers, designated R0 through R31 and used for storing addresses or data. The floating-point control unit contains a separate set of thirty-two 32-bit floating-point registers designated F0 through F31. These registers can be addressed individually, as sixteen 64-bit registers, or as eight 128-bit registers. The integer registers contain three ports. Five ports in the floating-point registers allow them to be used as a data staging area for performing loads and stores in parallel with floating-point operations.

The i860 operates on standard integer and floating-point data, as well as pixel data formats for graphics operations. All operations on the integer registers execute on 32-bit data as signed or unsigned operations and additional add and subtract instructions that operate on 64-bit-long words. All 64-bit operations occur in the floating-point registers.

The i860 microprocessor supports a paged virtual address space of four gigabytes. Therefore, data and instructions can be stored anywhere in that space, and multibyte data values are addressed by specifying their lowest addressed byte. Data must be accessed on boundaries that are multiples of their size. For example, two-byte data must be aligned to an address divisible by two, four-byte data on an address divisible by four, and so on, up to 16-byte data values. Data in memory can be stored in either little-endian or big-endian format. (Little-endian format sends the least significant byte, D7-D0, first to the lowest memory address, while big-endian sends the most significant byte first.) Code is always stored in little-endian format. Support for big-endian data allows the processor to operate on data produced by a big-endian processor, without performing a lengthy data conversion.

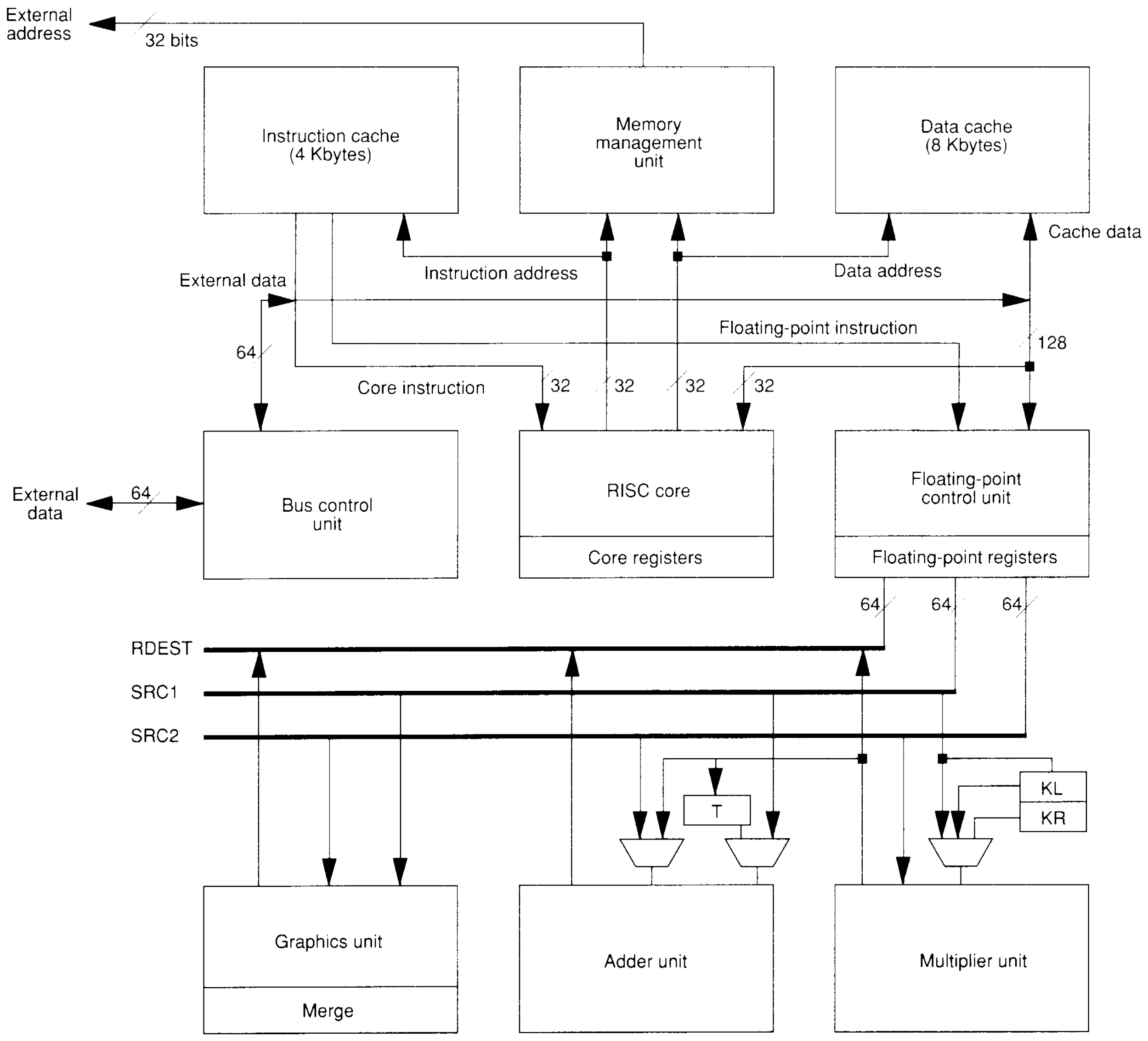


Figure 2. Functional units and data paths of the i860 microprocessor.

RISC core

The RISC core fetches both integer and floating-point instructions. It executes load, store, integer, bit, and control transfer instructions. Table 1 on page 19 lists the full instruction set with the 42 core unit instructions and their mnemonics in the left column. All instructions are 32 bits long and follow the load/store, three-operand style of traditional RISC designs. Only load and store instructions operate on memory; all other instructions operate on registers. Most instructions allow users to specify two source registers and a third register for storing the results.

A key feature of the core unit is its ability to execute most instructions in one clock cycle. The RISC core contains a pipeline consisting of four stages: fetch, decode, execute, and write. We used several techniques to hide clock cycles of instructions that may take more time to complete. Integer register loads from memory take one execution cycle, and the next instruction can begin on the following cycle.

The processor uses a scoreboarding technique to guarantee proper operation of the code and allow the highest possible performance. The scoreboard keeps a history of which registers await data from memory. The actual loading of data takes one clock cycle if it is held in the cache memory buffer available for ready access, but several cycles if it is in main memory. Using scoreboarding, the i860 microprocessor continues execution unless a subsequent instruction attempts to use the data before it is loaded. This condition would cause execution to freeze. An optimizing compiler can organize the code so that freezing rarely occurs by not referencing the load data in the following cycle. Because the hardware implements scoreboarding, it is never necessary to insert NO-OP instructions.

We included several control flow optimizations in the core instruction set. The conditional branch instructions have variations with and without a delay slot. A delay slot allows the processor to execute an instruction following a branch while it is fetching from the branch target. Having both delayed and nondelayed variations of branch instructions allows the compiler to optimize the code easily, whether a branch is likely to be taken or not. Test and branch instructions execute in one clock cycle, a savings of one cycle when testing special cases. Finally, another one-cycle loop control instruction usefully handles tight loops, such as those in vector routines.

Instead of providing a limited set of locked operations, the RISC core provides lock and unlock instructions. With these two instructions a sequence of up to 32 instructions can be interlocked for multiprocessor synchronization. Thus, traditional test and set opera-

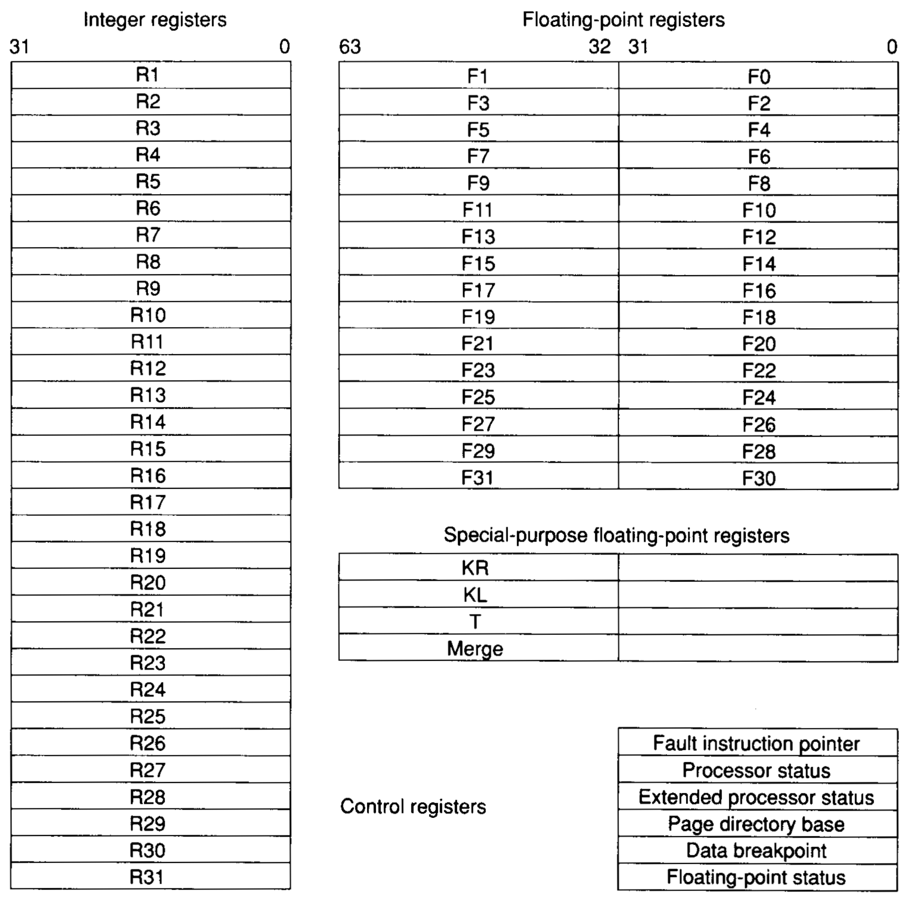


Figure 3. Register set.

Table 1.
Instruction-set summary.

Mnemonic	Description
Core unit	
Load and store instructions	
LD.X	Load integer
ST.X	Store integer
FLD.Y	F-P load
PFLD.Z	Pipelined F-P load
FST.Y	F-P store
PST.D	Pixel store
Register-to-register moves	
IXFR	Transfer integer to F-P register
FXFR	Transfer F-P to integer register
Integer arithmetic instructions	
ADDU	Add unsigned
ADDS	Add signed
SUBU	Subtract unsigned
SUBS	Subtract signed
Shift instructions	
SHL	Shift left
SHR	Shift right
SHRA	Shift right arithmetic
SHRD	Shift right double
Logical instructions	
AND	Logical AND
ANDH	Logical AND high
ANDNOT	Logical AND NOT
ANDNOTH	Logical AND NOT high
OR	Logical OR
ORH	Logical OR high
XOR	Logical exclusive OR
XORH	Logical exclusive OR high
Control-transfer instructions	
TRAP	Software trap
INTOVR	Software trap on integer overflow
BR	Branch direct
BRI	Branch indirect
BC	Branch on CC
BC.T	Branch on CC taken
BNC	Branch on not CC
BNC.T	Branch on not CC taken
BTE	Branch if equal
BTNE	Branch if not equal
BLA	Branch on LCC and add
CALL	Subroutine call
CALLI	Indirect subroutine call
System control instructions	
FLUSH	Cache flush
LD.C	Load from control register
ST.C	Store to control register
LOCK	Begin interlocked sequence
UNLOCK	End interlocked sequence
Floating-point unit	
Floating-point multiplier instructions	
FMUL.P	F-P multiply
PFMUL.P	Pipelined F-P multiply
PFMUL3.DD	Three-stage pipelined F-P multiply
FMLOW.P	F-P multiply low
FRCP.P	F-P reciprocal
FRSQR.P	F-P reciprocal square root
Floating-point adder instructions	
FADD.P	F-P add
PFADD.P	Pipelined F-P add
FSUB.P	F-P subtract
PFSUB.P	Pipelined F-P subtract
PFGT.P	Pipelined F-P greater-than compare
PFEQ.P	Pipelined F-P equal compare
FIX.P	F-P to integer conversion
PFIX.P	Pipelined F-P to integer conversion
FTRUNC.P	F-P to integer truncation
PFTRUNC.P	Pipelined F-P to integer truncation
PFLE.P	Pipelined F-P less than or equal
PAMOV	F-P adder move
PFAMOV	Pipelined F-P adder move
Dual-operation instructions	
PFAM.P	Pipelined F-P add and multiply
PFSM.P	Pipelined F-P subtract and multiply
PFMAM	Pipelined F-P multiply with add
PFMSM	Pipelined F-P multiply with subtract
Long integer instructions	
FLSUB.Z	Long-integer subtract
PFLSUB.Z	Pipelined long-integer subtract
FLADD.Z	Long-integer add
PFLADD.Z	Pipelined long-integer add
Graphics instructions	
FZCHKS	16-bit *z*-buffer check
PFZCHKS	Pipelined 16-bit *z*-buffer check
FZCHLD	32-bit *z*-buffer check
PFZCHLD	Pipelined 32-bit *z*-buffer check
FADDP	Add with pixel merge
PFADDP	Pipelined add with pixel merge
FADDZ	Add with *z* merge
PFADDZ	Pipelined add with *z* merge
FORM	OR with merge register
PFORM	Pipelined OR with merge register
Assembler pseudo-operations	
MOV	Integer register-register move
FMOV.Q	F-P register-register move
PFMOV.Q	Pipelined F-P register-register move
NOP	Core no-operation
FNOP	F-P no-operation

CC	Condition code
F-P	Floating-point
LCC	Load condition code

tions as well as more sophisticated operations, such as compare and swap, can be performed.

The RISC core also executes a pixel store instruction. This instruction operates in conjunction with the graphics unit to eliminate hidden surfaces. Other instructions transfer integer and floating-point registers, examine and modify the control registers, and flush the data cache.

The six control registers accessible by core instructions are the

- PSR (processor status),
- EPSR (extended processor status),
- DB (data breakpoint),
- FIR (fault instruction),
- Dirbase (directory base), and
- FSR (floating-point status) registers.

The PSR contains state information relevant to the current process, such as trap-related and pixel information. The EPSR contains additional state information for the current process and information such as the processor type, stepping, and cache size. The DB register generates data breakpoints when the breakpoint is enabled and the address matched. The FIR stores the address of the instruction that causes a trap. The Dirbase register contains the control information for caching, address translation, and bus options. Finally, the FSR contains the floating-point trap and rounding-mode status for the current process. The four special-purpose registers are used with the dual-operation floating-point instructions (described later).

The core unit executes all loads and stores, including those to the floating-point registers. Two types of floating-point loads are available: FLD (floating-point load) and PFLD (pipelined floating-point load). The FLD instruction loads the floating-point register from the cache, or loads the data from memory and fills the cache line if the data is not in the cache. Up to four floating-point registers can be loaded from the cache in one clock cycle. This ability to perform 128-bit loads or stores in one clock cycle is crucial to supplying the data at the rate needed to keep the floating-point units executing. The FLD instruction processes scalar floating-point routines, vector data that can fit entirely in the cache, or sections of large data structures that are going to be reused.

For accessing data structures too large to fit into the on-chip cache, the core uses the PFLD instruction. The pipelined load places data directly into the floating-point registers without placing it in the data cache on a cache miss. This operation avoids displacing the data already in the cache that will be reused. Similarly on a store miss, the data writes through to memory without allocating a cache block. Thus, we avoid data cache thrashing, a crucial factor in achieving high sustained performance in large vector calculations.

PFLD also allows up to three accesses to be issued on the pipelined external bus before the data from the first cache miss is returned. The pipelined loads occur directly from memory and do not cause extra bus cycles to fill the cache line, avoiding bus accesses to data that is not needed. The full bus bandwidth of the external bus can be used even though cache misses are being processed. Autoincrement addressing, with an arbitrary increment, increases the flexibility and performance for accessing data structures.

Memory management

The i860's on-chip memory management unit implements the basic features needed for paged virtual memory management and page-level protection. We intentionally duplicated the memory management technique in the 386 and 486 microprocessors' paging system. In this way we can be sure that the processors easily exist in a common operating environment. The similar MMUs are also useful for reusing paging and virtual memory software that is written in C.

The address translation process maps virtual address space onto actual address space in fixed-size blocks called pages. While paging is enabled, the processor translates a linear address to a physical address using page tables. As used in mainframes, the i860 CPU page tables are arranged in a two-level hierarchy. (See Figure 4.) The directory table base (DTB), which is part of the Dirbase register, points to the page directory. This one-page-long directory contains address entries for 1,024 page tables. The page tables are also one page long, and their entries describe 1,024 pages. Each page is 4 Kbytes in size.

Figure 4 also shows the translation from a virtual address to a physical address. The processor uses the upper 10 bits of the linear address as an index into the directory. Each directory entry contains 20 bits of addressing information, part of which contains the address of a page table. The processor uses these 20 bits and the middle 10 bits of the linear address to form the page table address. The address contents of the page table entry and the lower 12 bits (nine address bits and the byte enables) of the linear address form the 32-bit physical address.

The processor creates the paging tables and stores them in memory when it creates the process. If the processor had to access these page tables in memory each time that a reference was made, performance would suffer greatly. To save the overhead of the page table lookups, the processor automatically caches mapping information for the 64 recently used pages in an on-chip, four-way, set-associative translation lookaside buffer. The TLB's 64 entries cover 4 Kbytes, each providing a total cover of 256 Kbytes of memory addresses. The TLB can be flushed by setting a bit in the Dirbase register.

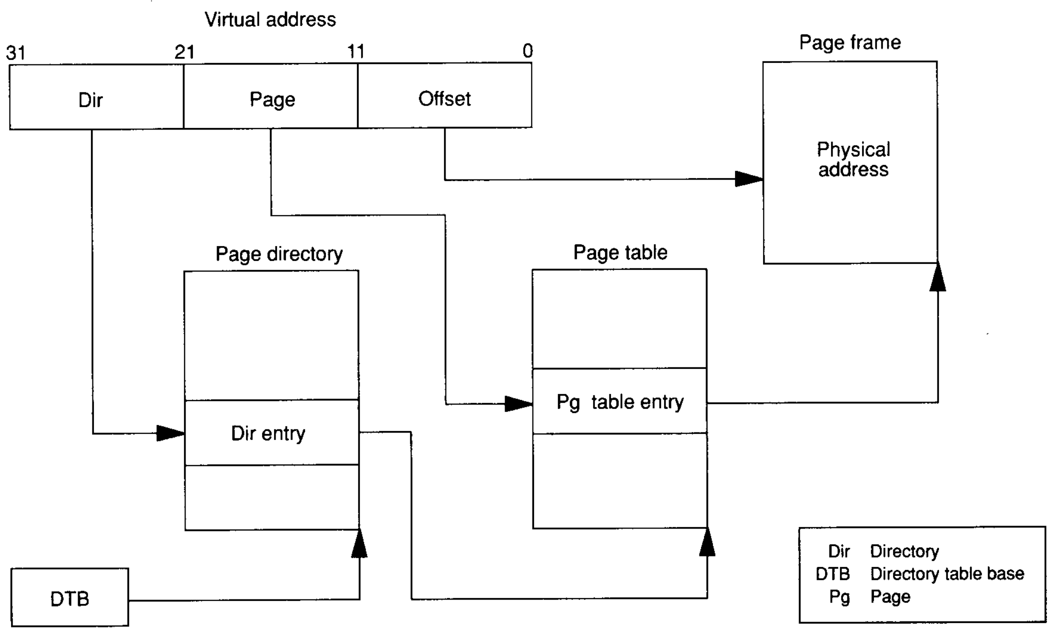


Figure 4. Virtual-to-physical address translation.

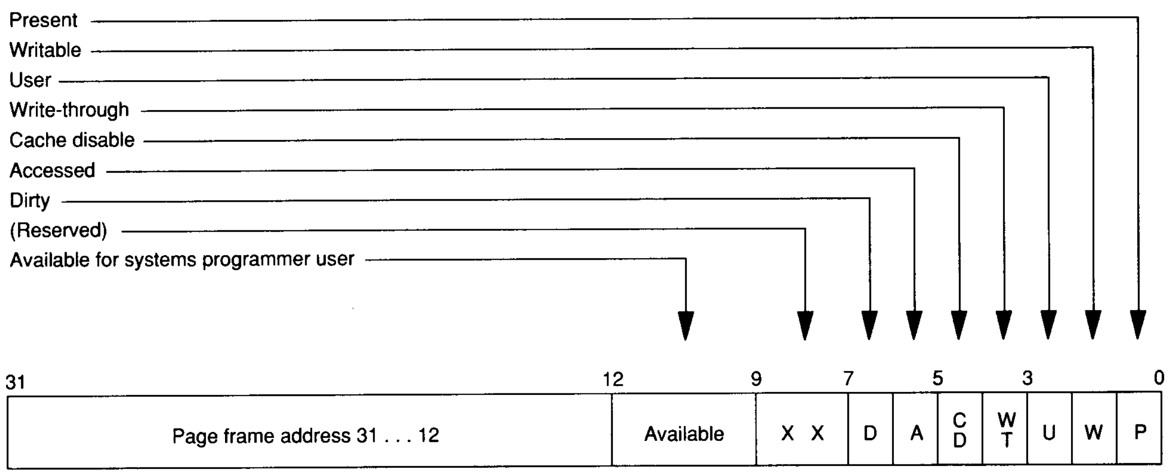


Figure 5. Format of a page table entry. (X indicates Intel reserved; do not use.)

Only when the processor does not find the mapping information for a page in the TLB does it perform a page table lookup from information stored in memory. When a TLB miss does occur, the processor performs the TLB entry replacement entirely in hardware. The hardware reads the virtual-to-physical mapping information from the page directory and the page table entries, and caches this information in the TLB.

The format of a page table entry can be seen in Figure 5. Paging protects supervisor memory from user accesses and also permits write protection of pages. The U (user) and W (write) bits control the access rights. The operating system can allow a user program to have read and write, read-only, or no access to a given page or page group. If a memory access violates the page protection attributes, such as U-level code writing a read-only page, the system generates an exception. While at the user level, the system ignores store control instructions to certain control registers.

The U bit of the PSR is set to 0 when executing at the supervisor level, in which all present pages are readable. Normally, at this level, all pages are also writable. To support a memory management optimization called copy-on-write, the processor sets the write-protection (WP) bit of the EPSR. With WP set, any write to a page whose W bit is not set causes a trap, allowing an operating system to share pages between tasks without making a new copy of the page until it is written.

Of the two remaining control bits, cache disable (CD) and write through (WT), one is reflected on the output pin for a page table bit (PTB), dependent on the setting of the page table bit mode (PBM) in EPSR. The WT bit, CD bit, and KEN# cache enable pin are internally NORed to determine "cachability." If either of these bits is set to one, the processor will not cache that page of data. For systems that use a second-level cache, these bits can be used to manage a second-level coherent cache, with no shared data cached on chip. In addition to controlling cachability with software, the KEN# hardware signal can be used to disable cache reads.

Floating-point unit

Floating-point unit instructions, as listed in Table 1, support both single-precision real and double-precision real data. Both types follow the ANSI/IEEE 754 standard.[1] The i860 CPU hardware implements all four modes of IEEE rounding. The special values infinity, NaN (not a number), indefinite, and denormal generate a trap when encountered; and the trap handler produces an IEEE-standard result. The double-precision real data occupies two adjacent floating-point registers with bits 31 . . . 0 stored in an even-numbered register and bits 63 . . . 32 stored in the adjacent, higher odd-numbered register.

The floating-point unit includes three-stage-pipelined add and multiply units. For single-precision data each unit can produce one result per clock cycle for a peak rate of 80 Mflops at a 40-MHz clock speed. For double-precision data, the multiplier can produce a result every other cycle. The adder produces a result every cycle, for a peak rate of 60 million floating-point operations per second. The double-precision peak number is 40 Mflops if an algorithm has an even distribution of multiplies and adds. Reducing the double-precision multiply rate saves half of the multiplier tree and is consistent with the data bandwidth available for double-precision operations.

To save silicon area, we did not include a floating-point divide unit. Instead, software performs floating-point divide and square-root operations. Newton-Raphson algorithms use an 8-bit seed provided by a

```
          Do 10, I = 1, 100
   10     X = X * A + C

          FMUL X, A, temp
          FADD temp, C, X
          ______
          1 result per 6 clock cycles
  (a)

          Do 10, I = 1, 100
   10     X[I] = A[I] * B[I] + C

          M12TPM A[I], B[I], X[I - 6]
          ______
          1 result per clock cycle
  (b)
```

Figure 6. Floating-point execution models: data-dependent code in scalar mode (a) and vector code in pipeline mode (b).

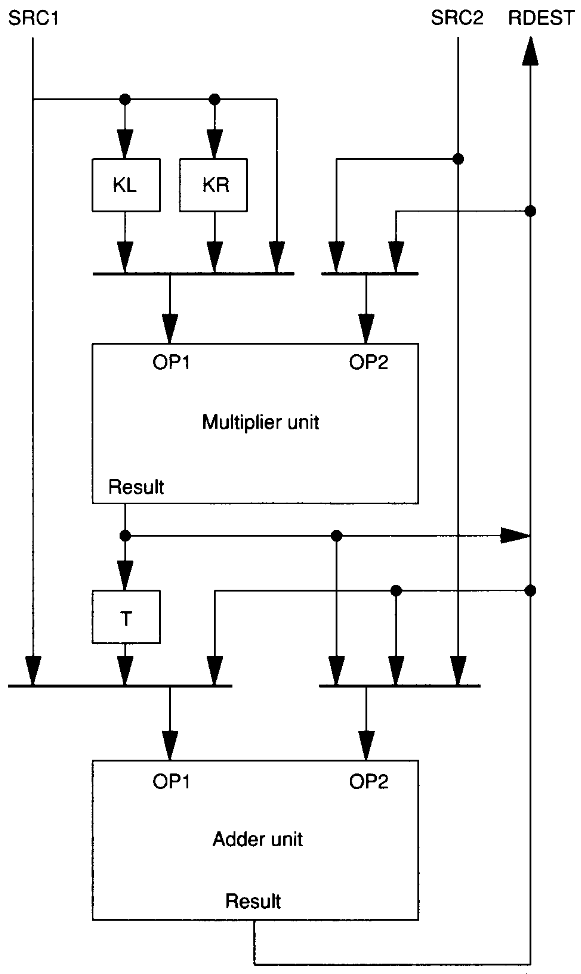


Figure 7. Dual-operation data paths.

hardware lookup table. Full IEEE rounding can be implemented by using an instruction that returns the low-order bits of a floating-point multiply. Therefore these algorithms can take advantage of the pipeline and allow 16-bit reciprocals used in many graphics calculations to be performed either in 10 clock cycles or four pipelined cycles.

The floating-point instruction set supports two computation models, scalar and pipelined. In scalar mode new floating-point instructions do not start processing until the previous floating-point instruction completes. This mode is used when a data dependency exists between the operations or when a compiler ignores pipeline scheduling. In the scalar-mode example of Figure 6 each iteration of the Do loop requires the results from the previous iteration and 6-cycle execution.

In pipelined mode the same operation can produce a result every clock cycle, and the CPU pipeline stages are exposed to software. The software issues a new floating-point operation to the first stage of the pipeline and gets back the result of the last stage of the pipeline. Destination registers are not specified when the operation begins, rather when the result is available. This explicit pipelining avoids tying up valuable floating-point registers for results, so the registers can still be used in the pipeline. Implicit pipelining, using scoreboarding, would cause the registers to become the bottleneck in the floating-point unit.

Pipelining also takes place in a dual-operation mode in which an add and a multiply process in parallel. Figure 7 shows the adder unit, the multiplier unit, the special registers, and the dual-operation data paths. Dual-operation instructions require six operands. The register file provides three of the operands, and the special registers and the interunit bypasses provide the remaining three. The instruction encodings specify the source and destination paths for the units.

Referring back to the pipeline-mode example of Figure 6, note that we show the dual-operation instruction M12TPM SRC1, SRC2, RDEST as M12TPM A[i], B[i], X[-6]. (The M12TPM mnemonic is a variation of the PFAN instruction.) This instruction specifies that the multiply is initiated with SRC1 and SRC2 as the operands. It also specifies that the add is initiated with the result from the multiply and the T register as the operands, and RDEST stores the result from the add. Because of the three stages of the add and multiply pipelines, the available result comes from the operation that started six clock cycles previously.

There are 32 variations of dual-operation instructions. Applications such as fast Fourier transforms, graphics transforms, and matrix operations can be implemented efficiently with these instructions. Some apparently scalar operations, such as adding a series of numbers, can also take advantage of the pipelining capability.

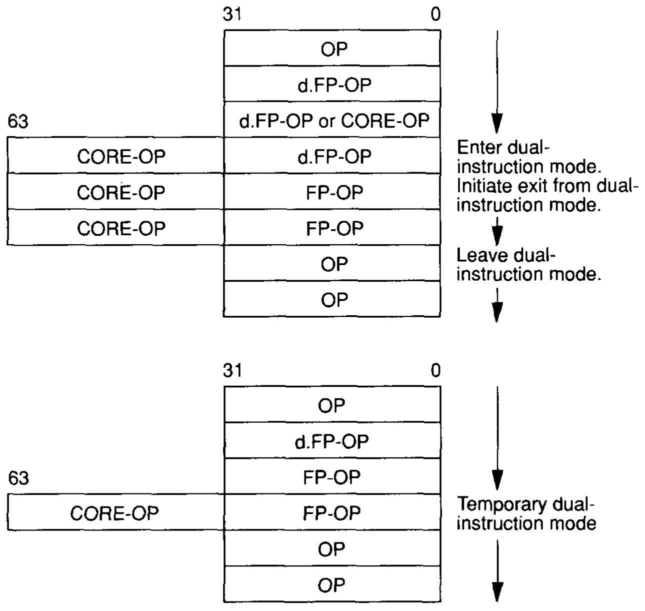

31
0
OP
d.FP-OP
63
d.FP-OP or CORE-OP
CORE-OP
d.FP-OP
Enter dual-instruction mode. Initiate exit from dual-instruction mode.
CORE-OP
FP-OP
CORE-OP
FP-OP
Leave dual-instruction mode.
OP
OP
31
0
OP
d.FP-OP
63
FP-OP
CORE-OP
FP-OP
Temporary dual-instruction mode
OP
OP

Figure 8. Dual-instruction-mode transitions.

The i860 microprocessor can provide its fast floating-point hardware with the necessary data bandwidth to achieve peak performance for the inner loops of common routines. The dual-instruction mode allows the processor to perform up to 128-bit data loads and stores at the same time it executes a multiply and an add. Figure 8 shows the dual-instruction-mode transitions for an extended sequence of instruction pairs and for a single instruction pair. Programs specify dual-instruction mode in two ways. They can either include in the mnemonic of a floating-point instruction a "d." prefix or use the assembler directives .dual . . . enddual. Either of these methods causes the dual or D-bit of the floating-point instruction to be set. If the processor while executing in single-instruction mode encounters a floating-point instruction with the D-bit set, it executes one more 32-bit instruction before beginning dual-instruction execution. In dual-instruction mode, a floating-point instruction could encounter a clear D-bit. The processor would then execute one more instruction pair before returning to single-instruction mode.

The floating-point hardware also performs integer multiplies and long integer adds or subtracts. Integer multiplies by constants can be performed in the RISC core using shift instructions. To perform a full integer multiply, the processor transfers two integer registers by using IXFR instructions. The FMLOW instruction performs the actual multiplication, and the FXFR instruction transfers the results back to the core. The total operation takes from four to nine clock cycles, depending on what other instructions can be overlapped.

Graphics

The floating-point hardware of the CPU efficiently performs the transformation calculations and advanced lighting calculations required for 3D graphics. The processor performs 500K transforms/second for 4 × 4 3D matrices, including the trivial reject clipping and perspective calculations. A 3D image display requires the use of integer operations for shading and hidden-surface removal. The graphics unit hardware speeds these back-end rendering operations and operates directly into screen buffer memory. It uses the floating-point registers and operates in parallel with the core.

Graphics instructions take advantage of the 64-bit data paths and can operate on multiple pixels simultaneously, realizing 10 times the speed of the RISC core when performing shading. Instructions support 8-, 16-, and 24/32-bit pixels, operating respectively on eight, four, or two pixels simultaneously.

In 3D graphics, polygons generally represent the set of points on the surface of a solid object. During transformation, the graphics unit calculates only the vertices of the polygons. The unit knows the locations and color intensities of the vertices of the polygons, but points between these vertices must be calculated. These points, along with their associated data, are called pixels. If a figure is displayed with only the vertices and simple lines, it appears as a wireframe drawing. The simplest wireframe drawing typically shows all vertices, even the ones that should be hidden from view by an overlapping polygon. To show shaded 3D images, the graphics unit must display the surface of the polygons. Where polygons overlap, it must display the polygon closest to the viewer.

In graphics calculations the z value represents the distance of a pixel from the viewer. Although the depth of each polygon's vertices is known, to overlay polygons not on a vertex, the graphics unit must interpolate the depths from the bordering vertices. This step is called z interpolation. In this step the depths of all points of a polygon can be determined. For overlapping points, the z values of different polygons can be checked and only the pixel data of the polygon closest to the viewer displayed.

To perform the procedure just described, the graphics instructions include intensity interpolation, z interpolation, and z-buffer checks. Intensity interpolation allows smooth linear changes in pixel intensity and color between vertices. This capability provides a smoother appearance than does the flat shading of the polygons. The more data bits per pixel, the smoother the interpolation becomes. The i860 CPU graphics instructions support both Gouraud and higher order shading techniques. Gouraud shading interpolates intensities along the scan lines. Figure 9 illustrates pixel interpolation for Gouraud shading of a triangle. The intensity level across the scan line shown is interpolated from 30 to 27.

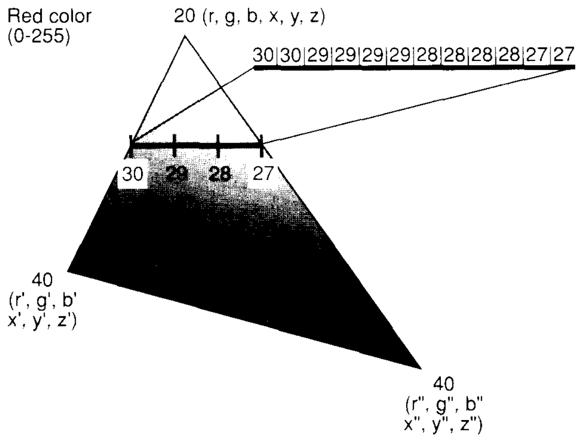


Figure 9. Pixel interpolation for Gouraud shading of a triangle for red colors and 0-255 intensity levels.

In graphics the z-buffer, which can reside in normal dynamic RAM, stores the depth of the pixel buffer currently being displayed. Instructions for z-buffer interpolation calculate the z values between vertices. Z-buffer check instructions compare the new pixels' z values to the values in the z-buffer, and if closer, the pixels are unmasked in the pixel mask register. The RISC core operates in parallel with the graphics unit and executes a pixel store instruction. The pixel store updates the pixels that are unmasked in the mask register. If a pixel is updated, the new z value needs to be stored to the z-buffer. The z-buffer check instruction updates the buffer with the minimum z value for each pixel.

Most workstations typically have a base graphics system of a simple frame buffer with simple display hardware. With a frame-buffer graphics system, the i860 CPU can perform Gouraud-shading operations on 50,000 triangles per second at 40 MHz. This level of performance exceeds that of workstations that include costly dedicated graphics processor boards.

Caches

The i860 CPU has a 4-Kbyte instruction cache and an 8-Kbyte data cache, each with its own address and data paths to support concurrent accesses. The data cache supports up to 128-bit accesses on each clock cycle, and the instruction cache supports up to 64-bit accesses. The aggregate bandwidth at 40 MHz is 960 Mbytes/second. Both caches combine two-way set-associative parallelism with a 32-byte line size. Additionally, the data cache uses write-back caching.

Both caches use virtual addresses to avoid a critical path in the cache access. Data cache accesses use the TLB lookup for enforcing the page-based protection. Since both caches use virtual tags, software must avoid the aliasing of data. Within a context, each physical address must only be accessed with one virtual address. During context switches, the instruction cache must be invalidated and the data cache flushed. The caches, although large enough to give hit rates above 90 percent within many applications, are too small to provide hits across context changes. Therefore, we did not feel process IDs or a duplicate set of physical tags to avoid flushing the cache between context switches were warranted.

Flushing the data cache is an easy way to avoid aliasing, and a simple calculation shows what little impact a small cache has on performance flushing. A typical i860 CPU context switch, including the data cache flush, takes approximately 65 microseconds. In the worst case, a workstation will change context 200 times per second; multiplying ($65 * 10^{-6}$ seconds $* 200$ times/second) equals a 1.3 percent performance degradation due to context switching.

Write-back data caching avoids propagating all writes to the external bus, which reduces bus traffic. It also prevents a bottleneck in vector operations where write traffic from the vector result collides with an incoming vector operand. With write-back caching, the hardware necessary to implement transparent caching for multiprocessor systems moved costs beyond the silicon budget of this implementation. Instead, we use software to manage cache coherency. Each processor can cache code, vector register data, and private stack data, while shared data remains uncached. Software controls the caching by using a cachable bit in the page table entries to prevent shared data from being cached. External hardware can also assert a cachable enable pin to control cachability of each line's read miss. The flush instruction forces all "dirty" blocks in the data cache back to memory. Flushing is needed before removing a page or changing to a new virtual address space.

We included optimizations for cache-miss processing. Each cachable read miss results in four bus cycles to fill the 32-byte cache line. First, the processor fetches the referenced data word and performs a wraparound fill to read the entire line. The processor can then continue execution when the first word is returned. The processor contains two 128-bit write buffers used for store misses and cache miss processing. When the processor issues a store instruction that misses the cache, it can continue execution while the write buffer carries out the actual memory write. The write buffers support two store misses and also support a delayed write back of the dirty cache line. If a cachable read miss displaces a dirty cache line, three operations take place. The processor writes the dirty line to the write buffer, the cache line read takes place on the external bus, and then the write back occurs.

A convenient software model for managing the data cache for vector computations on large matrices is to the treat the data cache as a "vector register set." Vectors, or their intermediate results, that are being reused are kept in the onboard cache by referencing with the normal floating-point load instruction. The vectorization process analyzes nested loops to determine which vectors are reusable in the second-loop level. Vector register references in the vector library routines use the normal floating-point load instruction. Vector memory references use the pipelined floating load instruction to stream the data from memory directly into the registers and not disturb the cache. Using the data cache as a vector register set is a more flexible concept than that found in many supercomputers with small, fixed-length vector registers. This concept offers the advantages of a vector register set for vector computations while retaining the flexibility of a data cache for scalar computations.

Bus interface

Designed for scalability to 50 MHz, the i860 CPU external bus performs a 64-bit transfer every two clock cycles. Thus, we achieve the design of a practical TTL (transistor-transistor logic) system, even at 50 MHz. The bus can interface either to a second-level cache or directly to a DRAM system. The bus allows optional pipelining for increasing the access time without decreasing the bandwidth. The full bus bandwidth can be realized from one bank of DRAMs, however, the latency will be greater than if a fast static RAM cache is used.

With the two-cycle transfer rate, the external bus can supply one memory operand for every double-precision add/multiply pair, or two contiguous single-precision operands for every two single-precision add/multiply pairs. The other two vector operands for an add/multiply pair must come from the onboard data cache. This approach provides the same ratio of floating-point rate to external memory bandwidth as the Cray 1. To avoid bus bottlenecks, the vectorization process must try to reuse two of the three vector operands in the second-level inner loop.

The i860 microprocessor contains a synchronous interface with a demultiplexed address and 64-bit-wide data bus. The address bus provides 32-bit addressing, consisting of 29 address lines and separate byte enable signals for each eight data bits. The bidirectional data bus can accept or drive new data on every other clock cycle, yielding a bandwidth of 160 Mbytes per second at 40 MHz.

The bus optionally allows for two levels of bus pipelining selected on a bus cycle-by-cycle basis. When pipelining, a new cycle starts prior to the completion of the outstanding cycles. Two levels of pipelining allow

Table 2.
Processor-pin summary.

Pin name	Function	Active state	Input/ output
Execution control pins			
CLK	Clock	—	I
RESET	System reset	High	I
HOLD	Bus hold	High	I
HOLDA	Bus hold acknowledge	High	O
BREQ	Bus request	High	O
INT/CS8	Interrupt, code size	High	I
Bus interface pins			
A31-A3	Address bus	High	O
BE7#-BE0#	Byte enable	Low	O
D63-D0	Data bus	High	I/O
LOCK#	Bus lock	Low	O
W/R#	Write/read bus cycle	High/Low	O
NENE#	Next near	Low	O
NA#	Next address request	Low	I
READY#	Transfer acknowledge	Low	I
ADS#	Address status	Low	O
Cache interface pins			
KEN#	Cache enable	Low	I
PTB	Page table bit	High	O
Testability pins			
SHI	Boundary scan shift input	High	I
BSCN	Boundary scan enable	High	I
SCAN	Shift scan path	High	I
Intel-reserved configuration pins			
CC1-CC0	Configuration	High	I
Power and ground pins			
V_{CC}	System power	—	—
V_{SS}	System ground	—	—

A # symbol after a pin name indicates that the signal is active when at the low-voltage level.

three cycles to operate at one time. Fast TTL latches can be used on the address and data bus. This method isolates the memory array from the processor pin timings, allowing easy scalability and providing the maximum time for memory accesses. With pipelining, the maximum data rate of the bus can be sustained even if the access time is six clock cycles. We achieve over 100 nanoseconds of address-to-data access time for a full bandwidth system at 40 MHz.

A summary of the processor pins appears in Table 2. We timed the processor with a single-frequency, TTL-level clock. An optional mode for executing out of one 8-bit-wide EPROM can be entered at reset by activating the INT/CS8 pin. In this mode the processor fetches instructions from the EPROM with the byte-enable signals BE2#-BE0# redefined as address lines A2-A0.

The HOLD, HOLDA, and BREQ signals activate arbitration of the processor's local bus. When a DMA controller, or another processor, needs access to the local bus of the CPU, it asserts HOLD. When the CPU completes all of its outstanding bus cycles, it floats the bus interface pins and returns HOLDA active high. The CPU will remain in this state with HOLDA active until HOLD is deasserted. The CPU can continue processing while in HOLD until the external bus is required. At this time it asserts the BREQ output signal. Arbitration logic samples the BREQ signal to arbitrate a shared bus.

The A31-A3 and BE7#-BE0# bus interface pins can access up to 4 gigabytes of address space. The address lines select the 8-byte word, and the byte-enable signals select the byte within the word. For read accesses to cachable memory, the processor caches the entire data bus so the byte-enable signals are ignored. For write operations the byte-enable signals determine which bytes in memory must be updated. The i860 microprocessor does not, however, allow misaligned accesses. Data of 32 and 16 bits must be placed on 4- and 2-byte boundaries, respectively. However, single-byte data can be placed at any byte location. The 64 bidirectional data pins can transfer 8-, 16-, 32-, or 64-bit quantities; pins D7-D0 signify the least significant byte and D63-D56 signify the most significant byte.

The processor asserts the ADS# output during the first clock cycle of each bus cycle to indicate the start of the bus cycle. The W/R# signal distinguishes the write and read bus cycles. The NENE# output indicates to the DRAM controller that the current address is in the same DRAM page as the previous cycle. As shown later, this information is useful for designing high-performance memory systems.

The NA# input to the CPU controls pipelining and can be asserted before the current cycle ends. When the processor samples NA# active, it can start driving the next bus cycle's address and definition. This can be done two times prior to returning data for any of the cycles.

While NA# controls the address and bus cycle definition signals, READY# controls the data operations. When READY# is sampled as active for a read, the processor latches the data from the data bus. When READY# is sampled as active for a write, the processor stops driving the data from that cycle. READY# also serves to end a bus cycle. The LOCK# signal output provides atomic (indivisible) sequences. Using LOCK# prevents the processor from relinquishing the bus even if HOLD is asserted. For multiprocessor systems, the external hardware only needs to lock the first address in a locked sequence.

This processor samples the KEN# input to determine

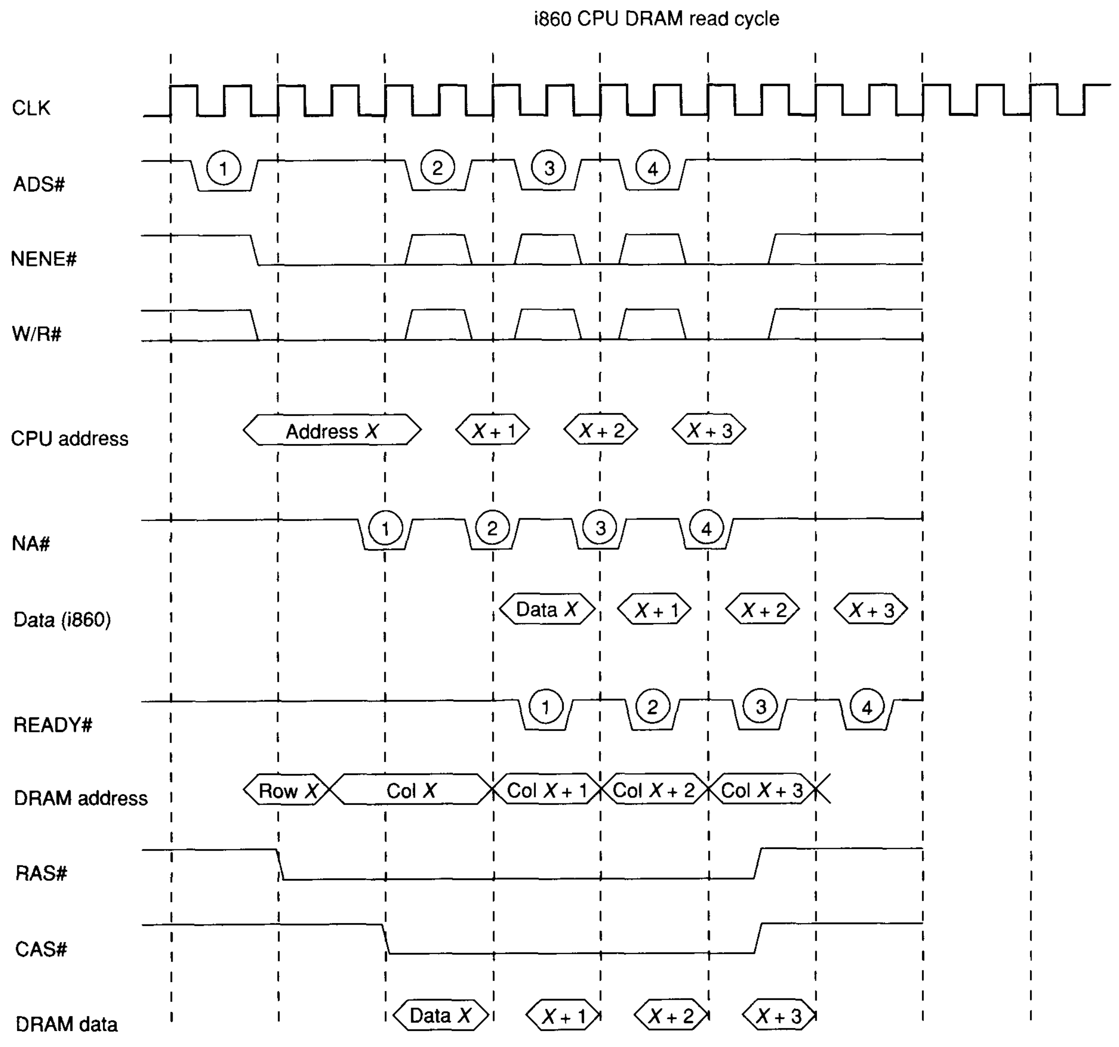


Figure 10. The CPU performs four read cycles to fill a cache line.

if the data for the current read cycle is cachable. Address space that is used for input and output can be decoded to deassert KEN# during I/O accesses. Software can also mark areas of memory as noncachable on a page-by-page basis. If the software has not disabled caching of the page, and KEN# is available for a read cycle, three additional 64-bit bus cycles will be generated to fill the 32-byte cache block.

Interfacing to a DRAM system

Figure 10 shows the processor performing four read cycles as it would do to fill a cache line. Also shown in the figure is the NA# signal returned to the processor, which indicates that the system can accept the next bus cycle. Two NA#s are returned before any of the cycles are completed. To complete a read cycle, the memory system provides the data on the bus and returns READY# to the processor. Once fully pipelined, the memory system provides data and READY# on every other clock cycle. Important for high performance, this data rate can be provided by ordinary static column DRAMs. The processor also provides the control signal NENE# to optimize DRAM control.

The memory system in Figure 11 on the next page consists of an address buffer; an address latch; eight latching data buffers; and a 64-bit-wide, static column-mode DRAM (256K x 4). This arrangement allows the memory size to be increased in increments of two megabytes. Using 256 x 4-memories also has advantages in reducing power and signal-drive requirements. To support the two levels of pipelining, the processor latches both address and data. The address latches hold

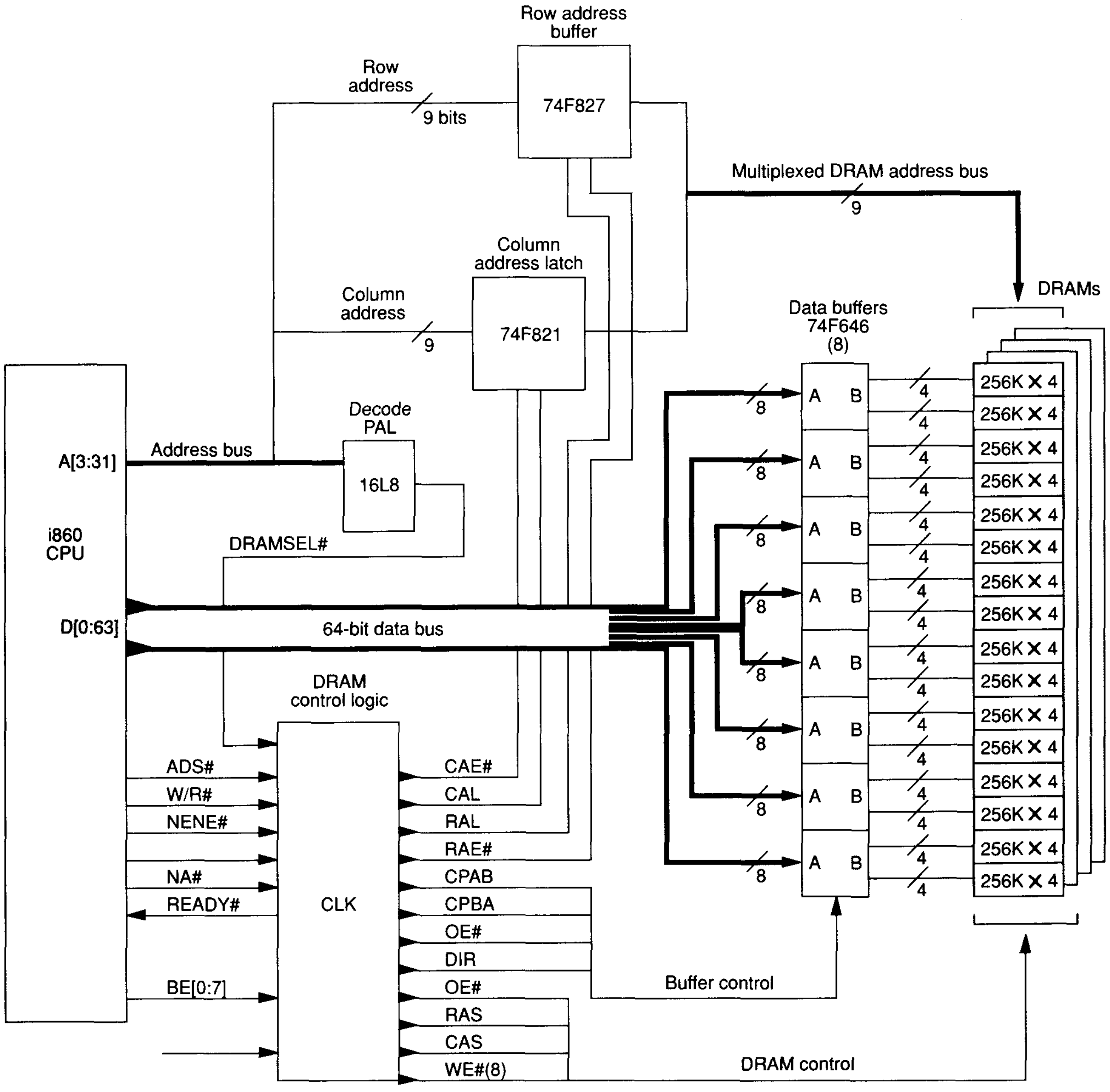


Figure 11. A DRAM system for the i860 microprocessor requires little "glue logic."

the address of the previous cycle, while the data from the cycle prior to that is held in the data buffers. Using TTL components on the address and data paths also has the advantage of isolating the memory system from the processor's pin timings.

The two address latches are used for multiplexing the row and column addresses from the processor to the DRAMs' address lines. When accesses occur within the DRAM page, only the column address needs to be supplied to the memory address lines. Most systems that use a fast-access DRAM mode need an additional hardware comparator. The i860 CPU has a comparator—which supplies the NENE# signal on each bus cycle—built into the bus unit. The controller uses this signal to determine if a fast static column-mode access can occur or if a full DRAM cycle needs to take place.

The bidirectional data buffers latch the data for both reads and writes. For reads, the buffers latch data and return READY# on the following clock cycle. With the two levels of pipelining the total access time is six cycles, while data is available every two cycles. Zero-wait-state operation does not require pipelining for write cycles. When a write occurs, the address and data latched in the buffers allow READY# to be returned to the processor. The actual write cycle occurs after READY# returns to the processor. This delayed write operation allows processor execution to continue even though the write has not fully completed.

Using 85-ns static column-mode DRAMs, the 33-MHz i860 microprocessor can operate at zero wait states for access within the DRAM page. The two-level pipelining and two-clock transfer rate allow the processor to sustain performance without the need for an external cache memory system.

Software support

Both internal development teams and independent vendors provide a full complement of software development tools and operating systems for the i860. Figure 12 shows the software development tools available: C and Fortran compilers, assembler/linker, simulator/debugger, Fortran vectorizer, plus mathematical, vector primitive, and 3D graphics libraries. To support the initial development environments, both Unix System V run on a 386 microprocessor and OS/2 host cross-compilers. The optimizations used in the compilers include coloring for register allocation, register-based parameter passing for calls, interblock common subexpression and loop invariant elimination, constant propagation, strength reduction, extensive peephole optimizations, and instruction scheduling.

Scientific-application support includes a Fortran vectorizing precompiler. Vectorization occurs in Do and If loops, outer loops, and forward-branching conditional operations. The precompiler recognizes these structures and generates calls to a set of preprogrammed procedures. The preprogrammed procedures are optimized for the processor's instruction set and for managing the data cache as a vector register. Additionally, other high-level languages can call these procedures. We plan to further increase the degree of parallelism that high-level languages can use in the processor. We also provide a library of assembly-language routines for scalar mathematics.

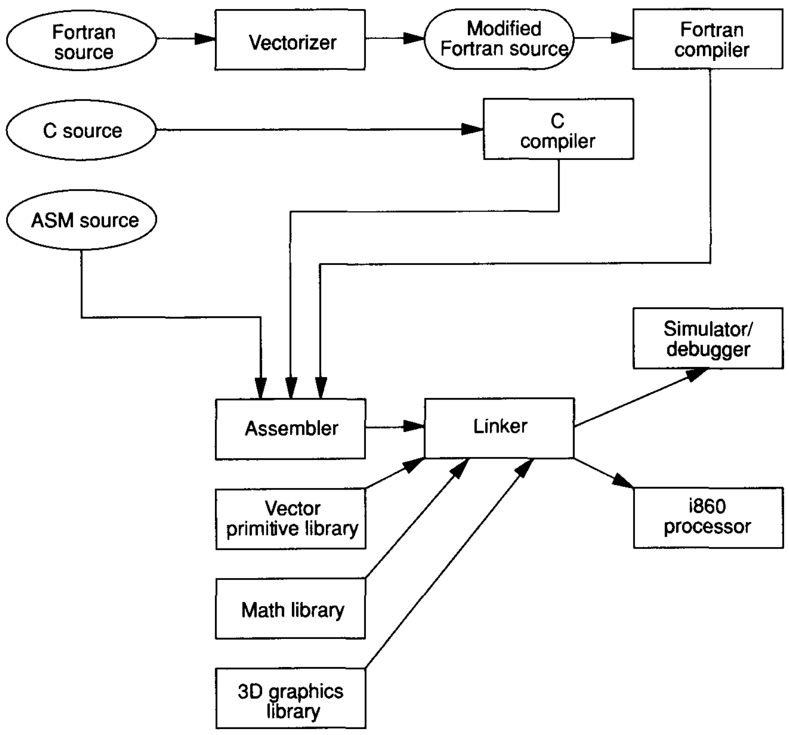


Figure 12. Software development environment supporting the i860.

The first 3D visualization tool ported to the i860 CPU is Ardent Computer's Dore. This tool supports both real-time, interactive 3D modeling and higher quality static images. Several windowing environments and other 3D tools and libraries are also being ported.

Application software can be run on either a software simulator or an add-in application accelerator. Both share a common debugging interface. The simulator allows the user to model different memory systems and measure their effects on performance. A Unix V/386 or OS/2 hosts the application accelerator, which includes a runtime operating environment that maps I/O requests back to the host processor.

A multiprocessing version of Unix System V Release 4.0 is under development for the i860 CPU. This is a joint effort by AT&T, Convergent Technologies, Intel, Olivetti, Prime Computer, and others. We plan to maintain source-code compatibility with the high-level languages between the 386, i486, and i860 microprocessors. Specifications for an applications binary interface standard (ABI) will allow portability of application software across multiple vendors' system implementations.

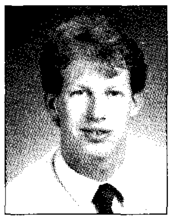
The i860 microprocessor begins the second generation of 32-bit RISC processors. By using a 64-bit architecture, the i860 delivers balanced MIPS, Mflops, and graphics performance. The million-transistor budget lets us integrate the RISC core and provide dedicated, fast floating-point hardware, graphics capabilities, and cache memories on one chip. The design allows maximum parallelism between the functional units while achieving a balance between computation speed and data bandwidth. Mainframe and supercomputer architectural concepts let the processor offer a complete solution to the requirements of high-computation applications.

References

1. *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, IEEE Computer Society Press, Los Alamitos, Calif., 1985.

Les Kohn is a chief architect for high-performance processors at Intel Corporation of Santa Clara, California, where he has worked on various 32- and 64-bit microprocessor design projects. Before joining the company, he worked as a software manager and architect for the NS32000 family at National Semiconductor. His interests include computer architectures and compilers and electronic synthesizers.

Kohn received his BS degree in physics from the California Institute of Technology in Pasadena.



Neal Margulis is a senior engineer for high-performance processors at Intel. His interests include processor architecture and system design. Margulis received his degree in electrical engineering from the University of Vermont in Burlington. He is a member of the IEEE Computer Society and Tau Beta Pi.

Questions concerning this article may be directed to the authors through Michael Sullivan at Intel Corporation, SC4-42, 2625 Walsh Avenue, Santa Clara, CA 95051.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 150 **Medium** 151 **High** 152