# AN ARCHITECTURAL CONTRAST:

The M68000 Microprocessor Family
and the
8086/iAPX 286

November, 1983

## READ THIS

before your competition does.
Be certain your company is not
investing in obsolete designs.

ROUTE TO: _____

_____

_____

_____

_____

Additional copies may be obtained from your nearest Motorola sales office (see back cover).

# TABLE OF CONTENTS



## AN ARCHITECTURAL CONTRAST:

**The M68000 Microprocessor Family and the 8086/iAPX 286**

# LIST OF ILLUSTRATIONS

## SUMMARY

The definition of a microprocessor's architecture involves many tradeoff decisions and impacts in very important ways the speed, efficiency and reliability of any computer system based on that architecture.

The Motorola M68000 Family of microprocessors implements a clean and powerful 32-bit architecture with general purpose registers and orthogonal addressing modes. A break with full compatibility to older 8-bit architectures was necessary so that advanced concepts could be introduced by the all-new 8/16/32-bit M68000 Family. The result was a new architecture which encompasses features required of today's system solutions and those for the rest of this decade and beyond, with complete user object-code compatibility.

In contrast to M68000 architecture, the iAPX 286 is weighted down with the instruction set of the much older 8086, with all of the cumbersome attributes associated with such ancient chip designs.

As a result, such crippling concepts as segmented addressing and special purpose registers rule the inflexible philosophy of iAPX architecture. These exact their harsh penalties in a multitude of ways, including excessive object code generation — which causes painful complexities in the integration of large, intricate software systems — and forcing at least seven times slower-than-M68000 execution in the accessing and addressing of large array and data structures.
The decision to implement direct 16-megabyte linear addressing was one of the greatest factors in the industry-wide acceptance of the M68000 Family. Contrast this to the mere 64K-byte segmented accesses of the iAPX 286. This forces extra manipulations on programmers, who must constantly attempt to contort their way around the severe limitations of restrictive registers and addressing modes.

The iAPX 286 does not even support virtual memory, but only a virtual segment restart capability. Contrast this with the virtual I/O, virtual machine and virtual window concepts available with Motorola's M68000 architecture.

The 32-bit architecture of M68000 Family microprocessors stands in stark contrast to the 16-bit segmented 8086/iAPX 286. Remarkable differences appear in performance, implementations, ease-of-use, code and execution efficiency between the two architectures when they are applied to actual programming environments.

In all important respects, the M68000 Family is seen to be far superior to the antiquated 8086/iAPX 286 architecture and fulfills the more sophisticated needs and requirements of today's demanding software systems and methodologies.

This document details the contrasting M68000 and 8086/iAPX 286 architectures.

**In the final analysis and selection of a microprocessor vendor, you will undoubtedly require information beyond the scope of this document. Please refer to the back cover to locate your closest Motorola sales office.**

# FORWARD

## The M68000 Family: Some Design Philosophies

Many topics must be considered when an Original Equipment Manufacturer (OEM) selects a microprocessor for a target product. The importance of each of these topics varies between OEMs, and the evaluation of each is individual to each OEM.

This forward presents a general set of topics that are typically evaluated and how the M68000 Family relates. M68000 and iAPX 286 processors' architectures will then be compared in detail.

In analyzing the suitability of a particular microprocessor for use in a product, some of the areas of concern that arise are:

Performance
Need for product
Capability
Effort to design in
Ease of use
Reliability
Future

There is much to consider in each of these areas, spanning both the software and hardware issues about a microprocessor. But, perhaps the most important consideration in choosing a microprocessor is the architectural philosophy around which the chip was designed. Almost all of the other considerations are heavily influenced by that architecture.

All of these issues will be examined here, with an emphasis on architectural concepts.

### ARCHITECTURE

There are many different philosophies for computer architectures; each has its own relative merits. In designing a powerful, general-purpose microprocessor, a manufacturer must choose an architecture that best serves a wide variety of applications for a large number of customers. Certainly, all applications cannot be best served by a single processor, but a manufacturer must target product design for the largest percentage of applications to remain profitable.

To address the needs of the largest share of an open market, a microprocessor's architecture must not impose numerous restrictions on the user. A general-purpose architecture provides maximum flexibility for the user. This is the common thread of the M68000 Family — power, with flexibility.

**Application.** Processors designed with a general-purpose architecture can be easily used in far more applications than a processor which was designed for a particular application. However, general-purpose processors will usually not perform as well in a specific application as a processor that was specifically designed for that particular application.

For example, a processor designed specifically for navigational applications would operate quite well in an aircraft guidance system, but quite poorly in a word processor. A general-purpose processor would perform reasonably well in both applications.

**Instructions.** Instructions which can use any register that is available relieve the resource bottlenecks caused by instructions requiring specific registers. A general-purpose architecture can be adapted to many high-level languages, whereas, while a processor which is designed for a specific language will execute that language extremely well, it will execute most others quite poorly.

For example, a processor designed to run Fortran will execute Fortran very well, but not Pascal. The general-purpose processor will perform reasonably well with both languages.

**Bus Structure.** Bus structure should aid, not hinder, the hardware designer. It is easier and less expensive to build a system if the bus allows mixing of many memory and peripheral speeds and configurations. Special bus controllers and peripherals only raise the cost of the system and restrict flexibility.

Built-in support for 32 bits is mandatory if painful redesigns are to be avoided in the future.

## COMPATIBILITY

**Existing Software.** Another architectural consideration should be compatibility with an existing architecture, allowing use of the existing software base of that older architecture. This very significant factor could sometimes offset the costs of developing similar software for the new architecture.

**Compromise.** However, without quite a bit of care and forethought, compatibility with too old an architecture can lead to too many compromises being made and subsequent loss of the computing power available in modern microprocessors. This is especially true when the original architecture was not designed to allow much room for expansion, or was already an extension of another design. Extensions of some architectures may compromise performance or ease-of-use unless those extensions were planned in the original product.

Eventually, in order to effectively implement all of the new-and-improved as well as tried-and-true operations in a microprocessor, compatibility with the older architectures must be abandoned. Also, the extent to which code compatibility is maintained, short of 100%, can degrade the advantage of compatibility and place greater burden on the user.

This desire for compatibility is justifiably ended when the advantages of the newer operations available in processors with a different architecture outweigh the disadvantages of incompatibility with the older architecture.

As an example: older programs often perform functions which are no longer needed, have been superceded by faster algorithms, or simply have been patched so many times that they are much too burdensome to use. These programs have become useless as such, and, when easily rewritten for a new architecture, will always run faster.

**Future.** When designing a new architecture for a processor, future code compatibility can be "built in." An architecture must be designed with future requirements in mind, allowing for those ideas which have not yet come to fruition and leaving sufficient room for extention beyond what is possible for the present design.

Designing an architecture for performance higher than what can be realized at present, and then adjusting that architecture to today's capabilities, leaves plenty of room for future expansion. This allows expansion by **design** rather than by compromise.

## PERFORMANCE

**Data Types.** The speed of program execution is related to many facets of a microprocessor's architecture. One of these is the applicability of the data types offered.

If a system makes high use of 32-bit data, for example, but the microprocessor's architecture only has 16-bit registers, then obviously unacceptable performance will result. If bit manipulations are heavily involved, such as with operating system execution decisions, then an architecture without bit instructions is likely to be awkward and slow at performing such functions.

**Register Store Size.** A wide register space has several very important advantages. Most arithmetic results can be completely contained in the high-speed registers instead of being swapped in and out of auxiliary memories. A large register space offers another advantage: significant optimizations can be done by high-level compilers.

If an architecture supports only a single register for multiply and divide, think of the disastrous bottleneck caused by the management of values through this one register.

**Address Space Size.** High performance systems demand quick access to large data stores. Modern computer systems must manipulate large structures for such things as sorting, graphics processing, database searches, and heavy number crunching, plus support multi-user, multi-tasking operations.

Any architecture which prohibits or compromises large data structures cannot run any of these applications (and many others) efficiently.

## THE CREATION OF AN ADVANCED ARCHITECTURE

The 16-bit MC68000 microprocessor pioneered an entirely new architecture. Much of the experience from the MC6800, MC6801, and MC6809 microprocessors, many of the ideas appreciated by programmers in popular minicomputers, and some innovative current and future concepts in computer architectures, all culminated in the design of the MC68000's architecture. Ties to the more primitive architectures of 8-bit machines were severed in all but the most necessary areas — such as the obvious advantages of already-available 8-bit peripherals. This left the computer architects the freedom to incorporate modern computer science techniques in the flagship microprocessor of a new family of compatible 8/16/32-bit microprocessors: the M68000 Family.

**Architectural Size.** An early architectural concern in the design of the MC68000 microprocessor was the realization that the 64-kilobyte address range of a 16-bit address would not be sufficient for a new breed of microprocessor. Though expressing 16-bit addresses in a 16-bit microprocessor seemed easy and code efficient, this small code and data space would eventually render the processor useless.

Various address sizes greater than 16 bits were studied. Ways of prefixing a few extra bits to the 16-bit address would encumber the programmer and/or require the additional bits to be carried in the opcode. If additional bits had to be maintained, the next logical question was how many: 24 or 32?

It soon became obvious that the next logical address size was 32 bits, and the decision was made to go ahead and design the address size as a 32-bit value with a temporary limit at 24 bits. This limit was made primarily to make the number of pins on the package consistent with current packaging technology.

**Linear Addressing.** There are many techniques of extending an address beyond 16 bits. These include: 1) retaining a linear address space; 2) paging fixed-size memory blocks; and 3) segmentation of variably-sized blocks. Paging techniques have been used for many years to simply extend the address range of processors. Segmentation is a little more involved and allows more flexibility in the placement of the memory blocks. While paging and segmentation are fairly easy to implement in processor design, their monumental disadvantages evolve from the fact that they are only ways of working around a basic shortcoming — the 16-bit address.

The MC68000 designers realized that although 32-bit linear addressing required their design of more addressing bits than either paging or segementation schemes, the approach did not impose any artificial boundaries on the programmer and allowed vastly more freedom in writing programs. Therefore, the MC68000 was designed to support a full 32-bit linear address space and not force the programmer to maintain time-consuming paging or segmentation registers.

**Data Register Size.** With the 32-bit address size, it then seemed natural to design the data handling capability to 32 bits. This would ease the programmer's job by eliminating discrepancies between different data and address register sizes.

Data quantities of 32 bits would also extend the architecture a good distance into the future, not only by handling the 16-bit needs of the present, but also the 32-bit needs to come. Again, practical considerations in the pin count of the present day dual-in-line packages directed the present limitation of the external data path to 16 bits. Now that additional data paths are being added with the Pin Grid Array (PGA) package, the M68000 Family microprocessors allow you to take greater advantage of the Family's 32-bit capability. Bus control circuitry allows 32-bit transfers to take place using multiple memory accesses.

**General Purpose Registers.** General purpose registers are placed in processors to provide high speed access to commonly used variables. Programmers must have total freedom

in the way those registers are allocated and used or else the original intent of the registers (high speed access) is defeated.

If the registers have dedicated uses, and this dedication forces the programmer to constantly swap information in and out of them, the data might just as well stay out in memory. This emphasizes the need for true general-purpose registers — those which may be assigned at will — to provide real programming efficiency.

**Register Set Size.** The MC68000 general purpose register set consists of eight 32-bit data registers in which all types of data can be manipulated and eight 32-bit address registers where memory pointers may reside. Of these sixteen registers, the only dedicated use involves the processor's use of one as the stack pointer for subroutine and interrupt services. Otherwise, any data or address register may be used as an operand or as a pointer.

Thus, the design of M68000 Family microprocessors' architecture was finialized to a full 32-bit linear addressing range and data handling capability, with all registers handling 32-bit quantities in an undedicated manner.

Time has proven the wisdom of those decisions as the limitations of a 64-kilobyte addressing range have caused programmers numerous headaches as they find that present-day data and program sizes easily require much more than this range. Additionally, the simplicity of the linear address space, without architecturally imposed boundaries, has made programming the MC68000 — and subsequent M68000 Family microprocessors — easy.

## THE BEGINNING OF A FAMILY

The MC68000 was designed as the beginning of a family of high-performance microprocessors. As such, many features were considered at length in the design. Many of these, while not practical for implementation in the initial product, were anticipated in the architecture so that they could be incorporated at a later date. Also, space is available to allow additional enhancements in the future. Because of this foresight, the MC68000 is already available in a variety of new orientations.

**Family Members At Present.** The M68000 Family consists of the original MC68000, and the MC68008, the MC68010 and the MC68020. The MC68008 is identical to the MC68000 except that it has an 8-bit data bus and a 20-bit address bus, for use in reduced-cost systems. The MC68010 is an enhanced version of the MC68000 which supports true virtual memory, eliminating the need for the additional processors required by previous schemes. The MC68020 is a complete 32-bit implementation with many new instructions, an automatic co-processor interface, and an on-chip instruction cache which will allow the attainment of an 8 million-instructions-per-second execution rate.

**Extensions.** The chip implementation of all of these processors has proven the advantage of designing an architecture for future needs. These microprocessors are upward object code compatible for all user programs and data. Each is a planned upward progression of the original MC68000, with the MC68008 providing the MC68000 power and versatility for 8-bit systems. The extension capabilities of the MC68000 were built in; they were not an afterthought.

# AN ARCHITECTURAL CONTRAST

**The M68000 Microprocessor Family and the 8086/iAPX 286**

## GENERAL FEATURES

Though a look at the general features of both the M68000 Family and the 8086/iAPX 286 family is rather extensive, some of the more vital comparisons must be made.

The MC68000 introduced a totally new architecture. It was carefully thought out to aid the modern programmer by providing the tools with which to write the fastest programs with the least amount of effort. The iAPX 286 is an outgrowth of the 8086 (which was an outgrowth of the 8080), essentially providing the same operations with some added functionality. That added functionality includes faster execution and an on-chip memory manager.

**Register Set Alternatives.** The register set of a microprocessor sets the theme for the entire architecture and reflects the philosophies of the instruction set. It is the most frequently used area during program execution. Because of its frequent use, the register set needs to be easily accessable to allow streamlined programming and fast program execution.

Instructions which require specific registers in their operation may be more efficiently encoded (a chip implementation matter), but they force the programmer into much more rigid register use planning. Instructions which allow the selection of any register for use as a constant, variable, or pointer give the programmer the freedom to select the most available or appropriate register.

To make fast on-chip registers easy to use, they must be available for use without restriction. A truly general-purpose register set allows any register to be used for any operand or pointer to operands with any data or instruction.

All M68000 Family microprocessors have 16 general-purpose registers, each of which can contain 32 bits of information. Half of these registers are used to hold data, and the rest are used as pointers to memory. Within these categories, any of the eight registers may be used in any instruction which may use a register. No M68000 processor instruction requires the use of a specific register. Most M68000 instructions can use a data register as an operand, or an address register to point to an operand in memory. The programmer has free choice which of the eight data or eight address registers to use.

This means that if variables exist in registers D0 and D2-D5, the programmer may use D1 or D6-D7 for values needed for another operation. Both programmers and compilers may more easily assign registers to use as they see fit.

Contrast the general purpose register set to a register set which has architecturally-imposed registers used for particular operations.

The problem with dedicated registers can be demonstrated with the iAPX 286 and its ancestors. Intel architecture only allows one register, the AX register, to be used to perform multiplies and divides. If a desired instruction requires the use of this register and the AX register has been recently used, the value in AX must be saved in memory and a new value placed in AX, even if that new value is already in an adjacent register, say CX, for example. After the desired instruction is executed, although the result may be needed later, the new result will have to be saved, probably in memory. Then the old value from register AX must be reloaded for use in the next few instructions and, finally, the new result will have to be brought back in to the processor when it is needed.

This overhead of constantly swapping frequently used, dedicated registers significantly burdens the programmer and slows down processing, especially in compiler-generated code. Not only does the iAPX 286 multiply and divide require the exclusive use of the AX register,

but the unsigned multiply and both the signed and unsigned divides do not even offer an immediate addressing mode for the source operand. Thus, extra instructions and time are needed to load yet another register merely to hold the source.

**Register Scheme — General Purpose versus Dedicated.** In a general-purpose register scheme with a large number of registers, there will most likely be enough register space available to perform an instruction series. Let's say that a particular routine needs five 16-bit registers to hold data, two source pointers, a destination pointer, and two table pointers. Using any M68000 microprocessor, five data and five address registers could be used with three of each left over. No swapping of registers would be necessary.

The dedicated registers of the iAPX 286 would require at least four registers to be swapped each time the use was changed, consuming valuable execution time and code space as well as causing a programming headache. Programmers readily say that they can always use just one more on-chip register. This emphasizes the fact that the more general register space available, the better.

The M68000 Family's register set has eight 32-bit data registers and eight 32-bit address registers, as shown in Figure 1. Address register A7, although used by the processor as a stack pointer for subroutine and exception processing, is accessed and treated just the same as all other address registers. Also, due to its use as the stack pointer, a second register A7 exists to give distinction between the supervisor (operating system) stack and the user (application programs) stack. There is also a program counter and a status register.

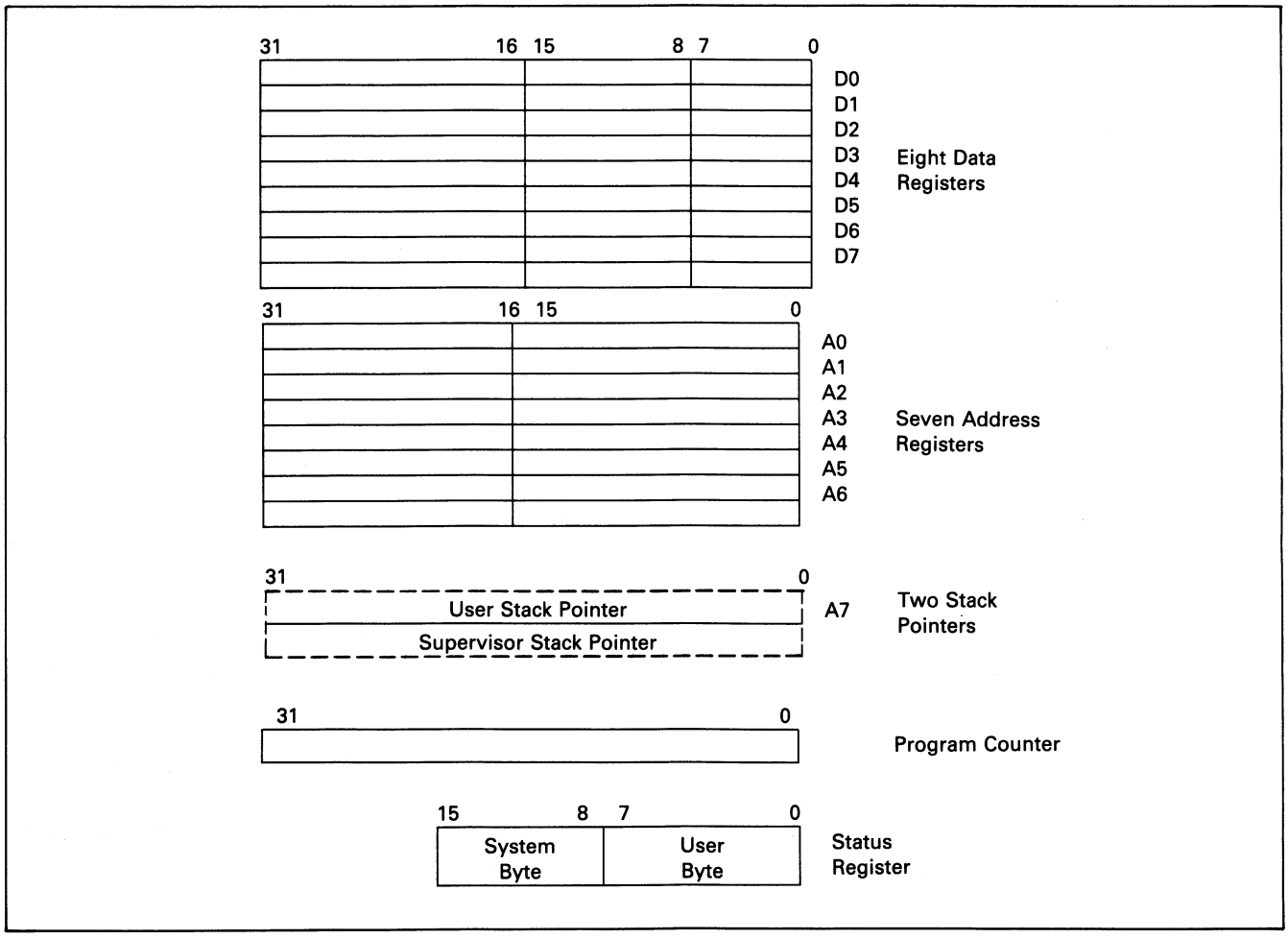



**Figure 1. M68000 Register Set**

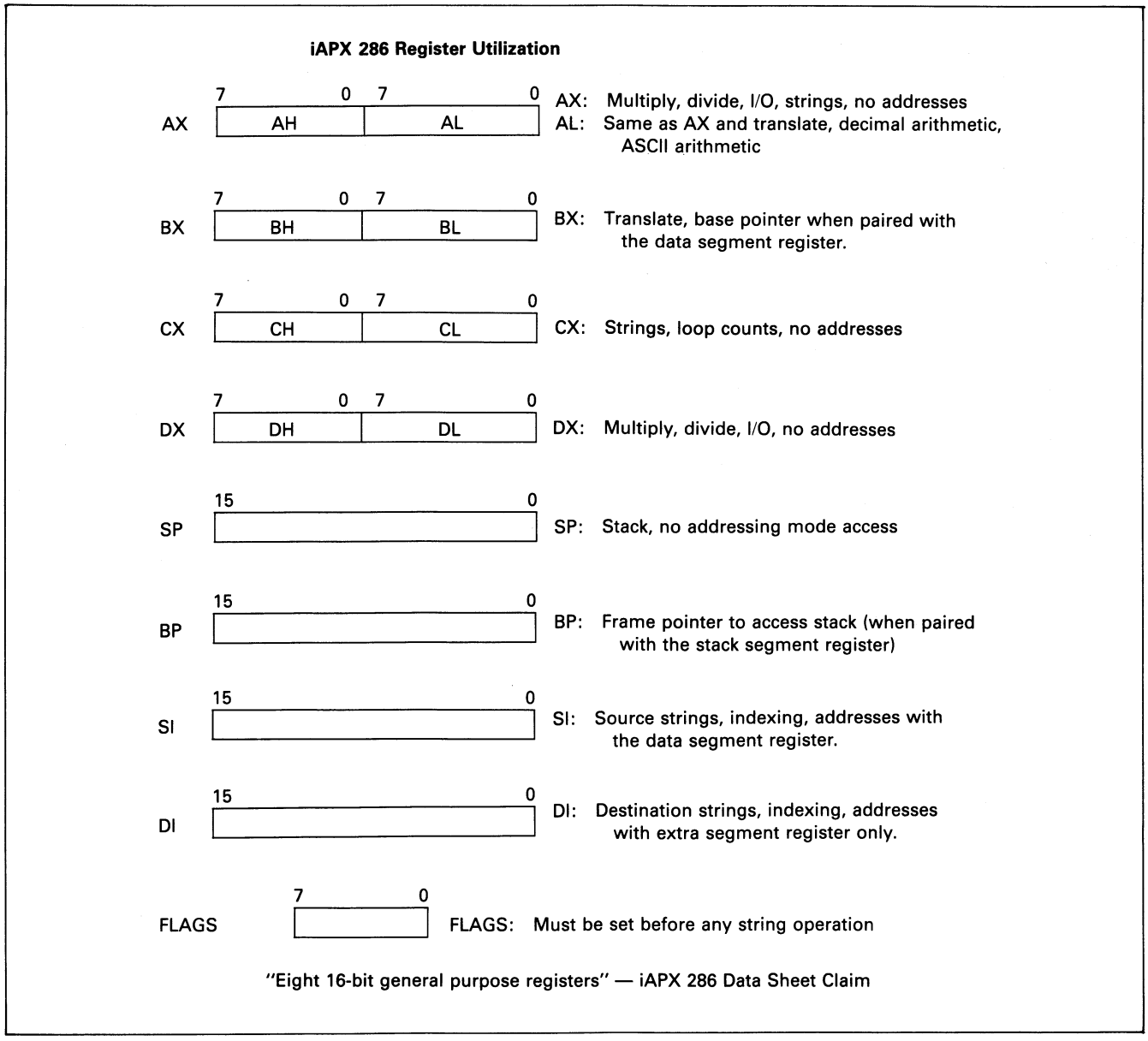



**Figure 2. Intel iAPX 286 Register Set**

The iAPX 286 has four 16-bit data registers which double as eight 8-bit registers, as shown in Figure 2. These registers have special purposes required by the architecture, as noted. Four 16-bit memory pointers are used as index pointers and as the stack pointer. Each of these registers also has particular requirements when used with certain instructions. For instance, the stack pointer is the only register which may be automatically incremented or decremented when data is accessed from it. In use, each of these four registers often has a particular data register paired with it.

To extend the addressing range of the iAPX 286, there are four segment registers: one each to point to a 64-kilobyte code, data, stack and extra segment. Again, during use, these registers are frequently paired with specific memory pointers by the architecture. The iAPX 286 also has a flag, instruction pointer and machine status word register.

**Source of the iAPX 286 Register Set Problem? — 8086.** Part of the problem with the iAPX 286 register set stems from its source — the 8086. The register set of the 8086 was based on that of the 8080, which was based on the original 8008. The 8080 is a second generation 8-bit microprocessor, usable but not very advanced from today's viewpoint. The concepts it embraced were not so much computer science, but solutions for increasingly complex logic functions which were previously built in medium scale integration. Thus, while these first generation processors were useful, most computer architects and scientists scoffed at them.

To design the third generation of microprocessors, such as the 8086, around the 8-bit 8080, might have been justifiable to retain some compatibility. However, to then extend that primitive architecture yet one more time to an advanced 16-bit microprocessor, so compromises the architecture that the result is not practical.

This genealogic background of the iAPX 286 architecture is the underlying reason why the future of upgradable products is doomed.

As shown in Figure 2, the right-hand side of the iAPX 286 data sheet register set illustration provides a list of the types of instructions which require that register for the operation. This list clearly points to the dedicated purpose registers of the iAPX 286.

Contrast this with the use of any data register or address register available in instructions for M68000 Family microprocessors.

M68000 Family processors provide eight address and eight data registers, all of which are 32-bits long and available for unrestricted allocation. The iAPX 286 has complex register use restrictions in which each register is forced to be dedicated to a selected group of operations. Not even a single register is free for the programmer to use at will. Segments on the iAPX 286 must be overwritten, registers swapped for proper setup, temporarily saved and then restored because of the dedicated register scheme.

Practically any 8086/iAPX 286 code can be examined to see the problems which plague a programmer using the now-archaic Intel architecture.

For example, let's take a very primitive string search. The EDN magazine benchmark study has a very simple string search where a subroutine searches a major string for the match of a minor string (Benchmark E). The Motorola code for that benchmark uses only four work registers (a mere quarter of its register set). However, the Intel architecture not only forces the use of its entire on-chip register set, but also requires that two of the working variables used in this short benchmark be saved and restored on the stack due to insufficient register work space! (See Appendix C). When trivial functions cause such complications in an architecture, the penalty for something of even modest complexity leads to very nasty performance problems and obese code generation. It is also interesting to note that the Intel architecture requires 41 lines of code to execute this benchmark and the MC68000 only 18 lines. Such instances of architectural limitations on the 8086/iAPX 286 are the rule rather than the exception, and several more examples given later will underscore this gross inadequacy.

Not only does the iAPX 286 severely limit the programmer to using only two data segment registers, but they interact with the offset registers which use them, and often overrides must be involved to properly match a segment register with its proper offset. This matching scheme is shown in Figure 3. It is hard to imagine an assembly language programmer remembering the defaults of which segment register match what base register and what possible overrides optionally exist. Even worse, imagine a compiler writer trying to implement these restrictions in a code generator! Performance is always going to be harshly limited by all of the juggling of registers. This is proven time and time again in independent benchmarks (Appendix B).

The iAPX 286 programmer must assure that proper segment registers are matched, and

| Memory References | Default Segment | Alternate Segment | Offset |
|---|---|---|---|
| Instruction Fetch | CS | none | IP |
| Stack Operation | SS | none | SP |
| Variable (except below) | DS | CS,ES,SS | [EA] |
| String Source | DS | CS,ES,SS | SI |
| String Destination | ES | none | DI |
| BP Used As Base Register | SS | CS,DS,ES | [EA] |

**Figure 3. iAPX 286 Segment Register Matching Scheme**

this must be managed with only four registers which are allowed to hold offset values, each of which have additional special purpose demands made on them. Registers BX, BP, SI, and DI are the only offset registers allowed. Register BX is heavily used for arithmetic/logical operations and its contents destroyed every time the translate instruction is used. The BP register is almost always dedicated as a frame pointer, so it is completely unavailable for most programs and it usually defaults to the stack segment (unless used as a double index).

The SI and DI registers which can be used by the programmer are overwritten whenever any string operation is executed. The SI and DI registers use the DS segment register by default, but, during string operations, the DI register switches to the ES segment register. The facts suggest frequent loading and unloading of these four offset registers, coupled with the required setup or overriding of the proper segment registers, heavily contribute to the poor performance of the 8086/iAPX 286 architecture. As dramatic proof of this, every Pascal program we have seen has compiled with less code generated on Motorola's Pascal 2.3 compiler compared with Intel's Pascal X125. (Refer to Appendix A for more details.)

## VIRTUAL MEMORY

It is imperative that any microprocessor in today's world support virtual memory capability. Virtual memory allows programs to run in what appears to be a large high-speed memory address space much larger than the actual physical memory present. This is accomplished by the capability of a microprocessor to detect access to memory pages which are not presently in the available memory pool. When a virtual memory system detects such a reference, it will, unknown to the program, make resident the proper memory page or allocate a new page as required. The program then continues, none the wiser.

Thus, virtual memory support provides four valuable attributes:

1. A program may act as though it has as much contiguous memory as it wishes for any program or data segment. One example would be for arithmetic arrays which may be several megabytes in size. Another quite common one is where a compiler simply keeps its entire symbol table in a memory segment and lets it extend without limit as each new symbol is entered. This latter capability completely obsoletes the burdensome effort of creating, processing and managing "spill" files on direct access devices.
2. The programmer need not be aware of this virtual memory capability and thus need not be concerned with the processor's memory protection mechanism.
3. The ability to support demand paging.
4. When a memory access faults, the operating system can completely recover and continue the program.

**iAPX 286's Virtual Claims.** The iAPX 286 completely lacks the first three attributes and only partially posesses the fourth.

First of all, no program on the iAPX 286 can ever ignore the crippling 64K segment limits imposed by its segmented architecture. There is absolutely no practical way around this serious flaw due to the 16-bit offset limit restriction. Programmers have no choice but to confine data structures to within an individual 64K data segment. Those that attempt to circumvent that severe limit are punished by handicaps of several times the lines and bytes of code required. Worse yet, the execution time slows down an order of magnitude because of the massive amount of descriptor loading then required. (An actual example is given later.)

The second attribute of true virtual memory is that the programmer need have no idea of the

underlying memory management protection mechanism, as the virtual facility operates in a totally transparent manner. An MC68010/68020 programmer codes programs without any regard for the target system. The program will execute properly regardless of whether the target system is virtual machine, virtual dynamic paged, virtual segmented, real segmented, or even has no MMU at all.

The iAPX 286, however, forces the programmer to tell the memory management unit (MMU) via special instructions about each memory segment just before it is referenced. There is no way out. Only if each segment is "banked" or "based" can the data therein be used. And, since there are only two segment registers which can be used for data (and even these are required for special purpose uses, such as string instructions), there is a considerable overhead for the loading and unloading of segment descriptors in the iAPX 286 architecture.

For example, the EDN sort benchmark time for the iAPX 286 just to do descriptor loading is far longer than the complete execution time for the entire MC68000 benchmark! (30.098 milliseconds for the iAPX 286 — 12 instructions — versus 17.348 milliseconds for the entire MC68000 benchmark.) Each descriptor load is two microseconds on the iAPX 286.

Contrast the negative effects of forcing the programmer to constantly base segments to the total freedom an M68000 programmer has with no size restrictions or MMU descriptor worries.

The third attribute of virtual memory is that of supporting demand paging. The direction of sophisticated microprocessor-based computer systems is toward virtual memory configurations based on demand paged physical memory. "Paging" means the subdivision of physical memory into equally-sized blocks called page frames. This division is invisible to tasks running within the system. This is a desirable method for memory allocation since entire code or data segments do not need to be resident in physical memory. Only those pages currently being used by a task need to be assigned page frames.

The iAPX 286 segmented approach is entirely incapable of supporting a demand page environment.

The term "demand" means that a program does not need to specify in advance what areas of its local address space it requires. An access to an address is interpreted by the system as a request to provide that memory. In a demand paged system, pages are loaded into page frames by the operating system when the program addresses them.

The iAPX 286 segmented scheme has just the reverse philosophy. The program must tell its MMU in advance what areas are to be used. Only a processor with real virtual memory capability, such as the MC68010 or MC68020, can provide a demand paging enviroment. The iAPX 286 MMU does not allow use of dynamic paged memory management, by far the most popular mode in use for large sophisticated systems. This is another drawback of Intel's design where the programmer is burdened with the MMU setup.

Demand paging allows system programmers to choose the page size which is most appropriate to their design. Such pages are usually 2K or 4K bytes. The iAPX 286, on the other hand, forces a complete segment to be made resident, even if only a few bytes are required by the program in execution. Thus, complete segments up to 64K may have to be loaded yet most of the memory never referenced. The same iAPX 286 system can also suffer from the other extreme of a large number of tiny segments causing excessive overhead due to the loading of segment registers. This happens due to the module concept of the iAPX 286 MMU, because there may be a large number of small routines.

For example, high-level language runtime libraries are typically full of a large number of small interdependent modules, and each, when called, requires the basing of one or more segments on the iAPX 286 MMU. Since segment basing requires from two to four microseconds depending on memory speed, this quickly adds up to a significant portion of the execution time accrued by a runtime system. Worse is the fact that if many or even all of the rou-

tines are resident (never causing segment faults) the segment load and checking overhead is still the same!

With demand paging there is absolutly no overhead for accessing loaded pages since there are no MMU setup instructions in the code. As can be seen in Appendix B, the iAPX 286 MMU philosophy causes excessive waste in execution time; Intel's own benchmarks prove this to be true. This is only the tip of the iceberg, however. Even more severe inefficiencies are presented later in this document.

As for the fourth attribute of virtual memory, once an iAPX 286 descriptor fault is detected then the operating system may furnish the required segment and continue the program undetected. However, the problem here is that only descriptor faults are detected AND NOT MEMORY FAULTS.

The following paragraphs discuss the wide variety of features a true virtual memory operating system can provide for its users, none of which will work on the iAPX 286. This is because the iAPX 286 DOES NOT PROVIDE VIRTUAL MEMORY BUT ONLY VIRTUAL SEGMENTS INSTEAD. One massive problem with the Intel "solution" is that minor bus problems create catastrophic system crashes. If a bus error occurs on the iAPX 286, there is no action that can be taken other than to TERMINATE THE AFFECTED PROGRAM IMMEDIATELY, EVEN IF IT IS THE OPERATING SYSTEM. The iAPX 286 will not allow any program hitting a bus fault to be continued. System integrity and reliability go down the tubes. Unfortunately, fault tolerance is next to impossible to provide with Intel's virtual segment memory scheme.

**True Virtual Memory Facilities.** True virtual memory support provides many other features beyond mere page fault recovery. The MC68010 and MC68020 allow an interrupted bus cycle to be either rerun or simulated. This simulation allows a new host of features which allow pseudo accesses to be granted to tasks or programs.

Since the National 16032 does not support bus simulation but only instruction retry, and the iAPX 286 does not support virtual memory faults at all, only the M68000 Family provides for virtual machine and virtual I/O notions.

Virtual I/O means that an operating system may provide a complete set of psuedo (virtual) peripherals for its tasks or even another operating system. Virtual I/O is mandatory for the realization of a Virtual Machine, since the entire set of physical hardware may have to be simulated—even MMU control registers. Only the M68000 Family allows the power of Virtual Machine, which means that an M68000 system can be built which can properly emulate the environment of any other given M68000 system.

Virtual windows are like virtual I/O except that a block of memory is simulated instead of I/O control registers. This block of memory is actually physically owned by another entity, be it a task or another system entirely.

For example, a simple parameter area can be arranged between all tasks of a system such that a write into that area by any one task will "wake up" all other tasks. The creation of such event or semaphore windows allows much greater flexibility in the design and use of operating system primitives. Another example would be for the construction of parameter windows which pass information to and from a data-base system. In this way, virtual connections can be allocated for given functions almost without limit, and with access as easy as performing a memory reference.

**More Advantages with Motorola's True Virtual Memory.** Creative use of the bus error retry facility allows for dramatic advances in fail-safe systems, as well. For example, let's say the operating system is updating a system table and a write hard-fails due to a defective memory location. Since the operating system bus error handler would be given control by an M68000 Family processor, it can then copy the affected page from the defective memory into a new page, while marking the old page unusable. The write then could be successfully continued and system integrity preserved.

The iAPX 286 however, would have no alter-

native but to halt or, at most, run in a crippled state with the affected operating system function totally disabled. This usually results in a complete system stall.

Since bus reads also can be restarted, this means that the M68000 Family provides complete fail-safe recoverability for any program code loaded into the system. Being read-only, program code can always be re-loaded from its source and the memory fetch attempted again. If that still fails, then another memory page may be mapped into place and recovery made as with the previous example.

Again, programs on the iAPX 286 would simply be terminated without any other recourse.

The bottom line is that only with Motorola's M68000 Family can you have real mainframe performance with real virtual memory and virtual machine capabilities, fault tolerance, and the added bonus of its superior instruction set and linear address space.

## DATA TYPES AND OPERATIONS

The variety of data types that a microprocessor can inherently operate on exemplifies its flexibility. The data types supported by M68000 processors and the iAPX 286 are shown in Figure 4. Generally, the more useful the data types, the more universally applicable the processor.

As shown in Figure 4, the M68000 has significantly more data type capability than the iAPX 286. The iAPX 286 needs to perform multiple instruction sequences and loops to do what native instructions offer on the M68000. The iAPX 286 offers no memory-to-memory arithmetic capability either.

**32 Bits Since the Beginning.** Certainly it has been seen over the years that data sizes are growing. As 4-bit processors gave way to 8-bit, and 8-bit to 16-bit, so 16-bit processors will bow to 32-bit processors. The key to each of these is the maximum data size that can be universally operated on. To be considered an "advanced" microprocessor today, a CPU should be designed with its operations extended all the way to 32-bit data types. This is particularly important to systems which anticipate expansion in the future, as well as today's high performance machines. The MC68000, in its original design, was made to perform arithmetic and logical operations on 8-, 16-, and 32-bits of data.

| Data Type | M68000 | iAPX 286 |
|---|---|---|
| Bits | X | none |
| Bytes | X | X |
| Boolean Bytes | X | none |
| Strings | X | X* |
| ASCII Arithmetic | none | X* |
| Word Integer | X | X |
| Long Word Integer | X | DIV/MUL only* |
| 8-Bit Binary Multiple Precision | X | Register Only |
| 16-Bit Binary Multiple Precision | X | Register Only |
| 32-Bit Binary Multiple Precision | X | none |
| BCD Byte | X | only with register and adjusts* |
| BCD String | X | none |
| Translate | X | Register Only* |
| 16-Bit Blocks | X | X |
| 32-Bit Blocks | X | none |
| 16-Bit Pointer | X | none |
| 32-Bit Pointer | X | X |
| Word Logical | X | X |
| Long Word Logical | X | none |

* Each of these data types requires exclusive use of the AX register.

**Figure 4. Data Types Supported Directly by Instructions**

In many ways, 32-bit capability gives the M68000 Family twice the capability of other 16-bit microprocessors like the iAPX 286. The only iAPX 286 instructions which handle 32-bit data are multiplication and division. And they are forced to use a single register pair (AX, DX)! M68000 processors perform these operations and many more. A comparison of the 32-bit instructions in M68000 Family processors and the iAPX 286 is given in Figure 5.

It is no wonder that Intel consistently avoids providing benchmarks with 32-bit arithmetic.

**Bit Manipulation.** Individual bit manipulation is very important for I/O operations, flag manipulations, resource allocation, and a host of other applications. Early 8-bit microprocessors typically had to perform bit manipulation in registers, by using the logical instruction (AND, OR, and Exclusive OR) on the appropriate bits.

| | M68000 Family (any 32-bit data register) | iAPX 286 (2 dedicated 16-bit registers) |
|---|---|---|
| Add | ADD | — |
| Add with carry | ADDX | — |
| And | AND | — |
| Arithmetic shift left | ASL | — |
| Arithmetic shift right | ASR | — |
| Bit test and change | BCHG | — |
| Bit test and clear | BCLR | — |
| Bit test and set | BSET | — |
| Bit test | BTST | — |
| Clear | CLR | — |
| Compare | CMP | — |
| Divide | DIV | DIV |
| Exclusive Or | EOR | — |
| Exchange | EXG | — |
| Sign Extend | EXT | CWD |
| Load Effective Address | LEA | — |
| Logical Shift left | LSL | — |
| Logical Shift right | LSR | — |
| Move | MOVE | — |
| Move Multiple Registers | MOVEM | — |
| Move Peripheral | MOVEP | — |
| Multiply | MUL | MUL |
| Negate | NEG | — |
| Negate with X bit | NEGX | — |
| Logical Not | NOT | — |
| Logical Or | OR | — |
| Push Effective Address | PEA | — |
| Rotate Right | ROR | — |
| Rotate Left | ROL | — |
| Rotate Right with X bit | RORX | — |
| Rotate Left with X bit | ROLX | — |
| Subtract | SUB | — |
| Subtract with carry | SUBX | — |
| Test | TST | — |

**Figure 5. 32-Bit Instruction Set Comparison**

The M68000 Family has four bit manipulation instructions that allow the testing, or testing and setting, clearing or changing of any bit in any data register or memory or I/O location. Single bit manipulation in the M68000 is very straightforward, easy to remember, and easy to use.

The iAPX 286 follows the 8086's lead in bit manipulation, only providing the ability to test individual bits by the primitive ANDing of the desired bit in a register after first using another instruction just to load it in. A rather complex routine would be necessary for an iAPX 286 to duplicate an M68000 Family processor's simple bit test and clear (BCLR) instruction. Simple bit manipulation operations are requisite in modern computer systems.

Many conditional branches are determined by testing bit flags in memory. This is especially true of most operating system code, and the lack of a bit test instruction on the iAPX 286 causes significant speed degradation for system routines.

**Register and Addressing Mode Flexibility.** The flexibility of M68000 Family processors in registers and addressing modes is perhaps better observed by looking at the operation codes (op codes) of the two machines, including the effective address fields. An example of the ADD instruction for M68000 Family processors and the iAPX 286 is illustrated in Figure 6. Note that each requires essentially 16 bits for the opcode, though the M68000 uses 16 additional bits for one indexed addressing mode. There are subtle differences in the exact operation of each ADD instruction, but the most significant can be seen when one looks at the variety of options available to each machine's ADD.

Each ADD instruction has three general options which must be settled to determine the exact opcode: direction of operation, data size and effective address (EA) calculation. Here is where the differences in the two processors stand out. There is little discrepancy in the direction selection either to memory or to a register. However, the M68000 allows not only byte and word data sizes like the iAPX 286, but also a 32-bit long word operand to be added as well as an order of magnitude more mode choices.

The real difference in programming each processor becomes apparent when one inspects the various combinations of accessing data available, as shown in Figure 7. Here the additional addressing modes as well as the additional registers add up to a real advantage with M68000 Family Processors. Totalling all of the register combinations that may be included in the address calculation, M68000 processors are by far more flexible, with almost 7000 different ways of accessing the two different operands. The iAPX 286 can address 8- and 16-bit data in only 1200 ways. The MC68020 has approximately $10^5$, or one hundred thou-

| M68000 Family Processors | Address Modes | Data Types |
|---|---|---|
| ADD Dn to EA | 8*(8+8+8+8+8+128+1) | 3 types |
| ADD EA to Dn | 8*(8+8+8+8+8+128+1+1+16+1) | 3 types |
| ADD EA to An | 8*(8+8+8+8+8+128+1+1+16+1) | 3 types |
| ADD Immed to EA | 8*(8+8+8+8+128+1) | 3 types |
| ADD Quick to EA | 8*(8+8+8+8+8+128+1) | 3 types |
| | 6984 modes to select from (20952 total) | 3 types |
| **iAPX 286** | **Address Modes** | **Data Types** |
| ADD reg to EA | 8*(4+4+4+4+4+1)*3 | 2 types |
| ADD EA to reg | 8*(4+4+4+4+4+1)*3 | 2 types |
| ADD Immed | (4+4+4+4+4+1)*3 | 2 types |
| ADD to Ax | (4+4+4+4+4+1)*3 | 2 types |
| INC EA | (4+4+4+4+4+1)*3. | 2 types |
| | 1134 modes to select from (2394 total) | 2 types |

**Figure 6. Comparative Opcode Example for ADD Instruction**

sand combinations, of addressing modes! Thus it has 100 times the options of the iAPX 286.

| MODE | No. of Register Options | |
|---|---|---|
| | M68000 Family | iAPX 286* |
| Data register | 8 | 4** |
| Address register | 8 | 4 |
| Address register deferred | 8 | 4 |
| Address deferred postincrement | 8 | unavailable |
| Address deferred predecrement | 8 | unavailable |
| Address deferred with offset | 8 | 4 |
| Index deferred with offset | 128 | 4 |
| Absolute short | N/A | N/A |
| Absolute long | N/A | unavailable |
| Relative with offset | N/A | unavailable |
| Relative Index with offset | 16 | unavailable |
| Immediate | N/A | N/A |

* Each access typically requires an implied segment register as well.
** 4 more, if using 8-bit only.
N/A Not Applicable

**Figure 7. Addressing Combinations Available for Accessing Data**

**iAPX 286 Just Doesn't Stack Up.** Addressing modes are very important to a programmer. They provide the means for expressing to the processor where to find the data. The more means to express the location, the better the chance the programmer will find just the way that is best for each set of circumstances.

A comparison of the addressing modes available in all M68000 Family processors and the iAPX 286 is shown in Figure 8. Each processor allows direct access to its on-chip registers, and has registers that can point to operands in memory. Each chip supports immediate addressing for the use of constants. However, M68000 Family processors have a pair of extremely useful addressing modes — postincrement and predecrement, which are missing on the Intel, Zilog, and National chips. These two modes combine to allow the programmer to easily:

1. scan through a sequence of data,
2. push data onto a programmer-built stack,
3. pull data from a programmer-built stack,
4. push data onto a programmer-built queue,
5. pull data from a programmer-built queue.

The importance of these two addressing modes is that they are available to virtually all instructions, and can use any of the eight address registers, eliminating the need for specialized stack instructions that only work on a dedicated stack. These addressing modes allow any M68000 instruction to operate on an M68000's system stack or on any number of stacks the programmer wishes to build. Eight of these may be immediately accessible at any time. While the M68000 processors are not what computer architects call stack machines, they are very simply programmed as such by always accessing operands on the stack.

The iAPX 286 has only two stack operations. The POP and PUSH operations of the iAPX 286 place or remove data from the top of the stack and they are limited to 16-bit values only. These are the only data stack operations available, and they operate on one specific stack. There is no other simple way of using another

| M68000 Family | | iAPX 286 | |
|---|---|---|---|
| 000 rrr | Dn | reg | mod = 11 |
| 001 rrr | An | " | " |
| 010 rrr | (An) | (SI/DI/BP/BX) | mod = 00 |
| 011 rrr | (An), An = An + incr | –no auto increment– | |
| 100 rrr | (An), An = An – incr | –no auto decrement– | |
| | see next | (SI/DI/BP/BX) + d8 | mod = 01 |
| 101 rrr | (An) + d16 | (SI/DI/BP/BX) + d16 | mod = 10 |
| 110 rrr | (An) + (Xn) + d8 | (BX/BP) + (SI/DI) + d8 | mod = 01 |
| | | (BX/BP) + (SI/DI) + d16 | mod = 10 |
| 111 000 | absolute16 | absolute16 only | mod = 00 & r/m = 110 |
| 111 001 | absolute32 | –no 24-bit absolute !! | |
| 111 010 | (PC) + d16 | –no IP relative– | |
| 111 011 | (PC) + (Xn) + d8 | –no IP relative– | |
| 111 100 | #Immediate | #Immediate | special |
| Conditional Program Transfers | | | |
| | 8-bit displacement | 8-bit displacement | |
| | 16-bit displacement | –no long jumps– | |

**Figure 8. Comparison of Available Addressing Modes**

stack in the iAPX 286. No other instruction will operate on the SS:SP stack without manual manipulation. Imagine a compiler which has a parameter stack, operator parsing stack, and an operation push down queue along with the standard system stack executing on an iAPX 286. This would amount to the simulation of three stacks with drastic performance handicaps and excessive compiler size.

M68000 processors take on such implementation requirements with ease, efficiency and no overhead.

Another problem with stack operations in the iAPX 286 is that, in order to access any data that is buried in the stack, a frame pointer must be set up. The frame pointer takes up another iAPX 286 register (in fact, the most accessible base register), further congesting the register set.

**Indexed and Absolute Addressing.** Both architectures have indexed addressing modes that allow the summing of one or two registers with an offset. These very powerful addressing modes are useful for indexing and referencing data in complex tables or lists. The iAPX 286 provides for both 8- and 16-bit displacements, while M68000 processors allow a 16- or 8-bit displacement, depending on the addressing mode. The general difference in these two is the 8-bit offset makes slightly more dense code.

M68000 processors allow indexing of 32-bit values, something which is an order of magnitude slower on the iAPX 286. See the paragraphs on large data structures for a dramatic example of how efficiently the M68000 processors handle 32-bit indexes compared to what is required on the iAPX 286 due to its MMU overhead.

The M68000 Family allows any of the eight address registers to be selected in the one-register-with-offset mode, and any of the sixteen data and address registers, as well as the eight address registers in the two-register-with-offset index mode. This gives a total of 8 and 128 options as opposed to the 4 and 4 options available to iAPX 286 programmers — a significant difference, with less chance of a register bottleneck with M68000 processors.

**iAPX 286 Does Not Offer Absolute Addressing.** Due to its forced segmentation, the Intel architecture mandates that all addresses have two components: a selector and an offset. The supposed code savings generated (which ap-

pears only in Intel's benchmarks and not independent benchmarks) are quite insignificant to the extra amount of instruction code it takes to handle the segments in many operations.

In fact, to make matters worse, the iAPX 286 architecture has forced dedication of segment registers for certain operations. For example, if one wants to move a string from "HERE" to "THERE" one must ensure that the "HERE" segment pointer is in one specific segment register (DS) and that the segment pointer for "THERE" is in another very specific segment register (ES). The previous values in these segment registers are lost and must be restored as later required.

The iAPX 286 further compounds the segment overhead problem because it must do a complete segment verification check every time a segment register is loaded. To illustrate the point, let's look at a simple memory-to-memory move in the MC68000, the MC68020, the 8086 and the iAPX 286:

| | Cycles | | | |
|---|---|---|---|---|
| | **MC68000/MC68010** | **MC68020** | **8086** | **iAPX 286** |
| * M68000 MOVE A WORD FROM HERE TO THERE | | | | |
| MOVE.W HERE,THERE | 28 | 8 | — | — |
| * iAPX 286/8086 MOVE A WORD FROM HERE TO THERE | | | | |
| LDS SI,HEREPTR | — | — | 22 | 21 |
| LES DI,THERE | — | — | 22 | 21 |
| MOVS | — | — | 18 | 9 |
| | 28 | 8 | 62 | 51 |

As can be seen, both the iAPX 286 and the 8086 are significantly slower. However, one point to mention is the fact that the iAPX 286 is running with twice the speed memory as both the 8086 and MC68000. Since the iAPX 286 time will be actually half when equal memory speeds are considered, the real results are more like:

| **MC68000/MC68010** | **MC68020** | **8086** | **iAPX 286** |
|---|---|---|---|
| 28 | 8 | 62 | 102 |

The surprising conclusion is that segment overhead causes substantially more problems with the iAPX 286 than with even the 8086. Note that the Intel architecture is not 32-bit, like M68000 processors, but only 16 bits. Just for interest we also give this benchmark, which moves a long word:

| | Cycles | | | |
|---|---|---|---|---|
| | **MC68000/MC68010** | **MC68020** | **8086** | **iAPX 286** |
| * M68000 MOVE A WORD FROM HERE TO THERE | | | | |
| MOVE.L HERE,THERE | 36 | 8 | — | — |
| * iAPX 286/8086 MOVE A WORD FROM HERE TO THERE | | | | |
| LDS SI,HEREPTR | — | — | 22 | 21 |
| LES DI,THERE | — | — | 22 | 21 |
| MOVS | — | — | 18 | 9 |
| MOVS | — | — | 18 | 9 |
| | 36 | 8 | 80 | 60 |

Again, when when we compare with equal memory speeds:

| MC68000/MC68010 | MC68020 | 8086 | iAPX 286 |
|---|---|---|---|
| 36 | 8 | 80 | 120 |

**Program Counter Variances.** Program counter relative addressing makes it easy for the M68000 Family programmer to write position-independent object code. By being able to reference source data, relative to the location of the instruction to use the data, one can access data that is tabular, and known to be a specified distance away. This way,if the instruction is in ROM #17/18 and the data is in ROM #21/22 in one system, those ROMs may be moved to another system and execute identically so long as the data ROMs remain 4 chips apart from the instruction ROMs.

On the other hand, the iAPX 286 expressly forbids the moving of ROMs between systems! This is because the MMU on the iAPX 286 requires that all programs contain segment IDs which only have meaning to one individual operating system running on just a single hardware system. Thus, yet another drawback to Intel's method of segmented memory management appears. This, like many of the others, is due to the fact that the programmer has to contend with the iAPX 286 MMU directly, which leads to static non-changeable segment constants directly in object code.

M68000 Family processors provide two modes of program counter relative addressing: one with an offset and one using any of the 16 index registers plus an offset. These two modes allow the programmer to access data and still have the object code relocatable, without changing the code.

The iAPX 286 has no instruction pointer relative addressing modes.

Overall, M68000 Family microprocessors have a greater quantity of more flexible and more useful addressing modes than the iAPX 286. This has the final advantage of making accessing data by instructions of M68000 processors more simple, convenient and applicable, beyond the more fundamental advantage of a 32-bit linear address space over the restrictive segmented addressing of the iAPX 286.

## INSTRUCTION SET

One of the most important considerations in a microprocessor is the instruction set. From an inspection of the instruction set, one can quickly grasp the functionality of the processor. The more functional the instruction set, the more useful the processor will be. Data types and addressing modes are a significant part of the instruction set. The exact operations supported in a processor with specific instructions give a broader outlook on the capabilities of the machine.

M68000 processors were designed with an entirely new instruction set. No older microprocessor was used as a mold from which the instructions were chosen. However, all of the experience Motorola has had with 8-bit microprocessor products, and the experience of the designers and users of popular minicomputers, contributed directly to the concepts and philosophies integrated into the M68000 Family's instruction set.

The instructions available in M68000 microprocessors span many areas of programming, from bit manipulation to high-level language support. Powerful, yet flexible instructions were included which perform data movement, bit manipulation, BCD arithmetic, logical functions, integer arithmetic, shift and rotate operations, program control, and high-level language and operating system support.

M68000 Family processors have a complete, orthogonal set of instructions that form a core of generally useful operations. Special purpose operations which would appeal to only a small group of users, which have limited utility or which aid only one high-level language, were avoided in favor of more and flexible general operations.

Instruction execution was designed to be as fast as possible, with the speed of execution targeted to the frequency of use of each instruction. This was based on in-depth studies on the static and dynamic use of instructions.

Such operations as register data movement were streamlined to execute fastest, while exclusive ORing to memory was given a lower priority. This provides the optimum in software efficiency with M68000 processors.

The 8-bit MC68008 shares the identical instruction set of the MC68000. The 16-bit virtual memory MC68010 has an enhanced MC68000 instruction set, adding a handful of operations that aid the programmer in new areas of microprocessing. Also, speed enhancements of some of the original instructions have been included. The 32-bit MC68020 adds a number of new operations to the MC68010 instruction repertoire. These instructions broaden the reach of M68000 Family instructions even more. Bit-field and coprocessor operations are typical of these new instructions.

The instruction set of the iAPX 286 is essentially that of the earlier 8086, with two major additions. Some "new" instructions were added which try to duplicate features already present on the M68000 Family. Another group of instructions which was added was necessary for the control of the memory management scheme. No other real operations were added to the iAPX 286 beyond the capability of the 8086.

The iAPX 286 is an 8086 with a memory manager on-chip. No more!

Figure 9 illustrates the new iAPX 286 instructions and the M68000 Family instructions that they duplicate. Note that the new Intel instructions are still limited to 16 bits.

There are some significant differences in the instruction sets of the processors, and many of these differences appear in the discussions of other facets of them, such as addressing modes. Here, some of the more significant differences will be looked at.

The iAPX 286 retains all of the instructions of the 8086. A couple of these were enhanced, but most stayed the same. Most of the critical shortcomings of the 8086 instruction set remain.

**Stack Operation.** As mentioned in the addressing mode discussion, the M68000 Family allows essentially all data operations to a stack, including a simple addressing mode, to access data deep in the stack (An + d16). Bytes, words or long words of data may be placed on or manipulated in a stack. M68000 microprocessors also have a move multiple register (MOVEM) instruction which allows either all or any subset of the general registers to be moved to or from any stack or any data location. This is very handy for context switches.

| iAPX 286 | M68000 Family |
|---|---|
| POPA<br>(regs not selectable)<br>(stack only) | MOVEM −(SP),reg set<br>(specified registers)<br>(any addressing mode) |
| PUSHA<br>(regs not selectable)<br>(stack only) | MOVEM reg set,(SP)+<br>(specified registers)<br>(any addressing mode) |
| PUSH immed | MOVE #immed,(SP)+ |
| IMUL immed | MULS #immed,Dn |
| Shifts immed | Shifts #cnt,Dn |
| Rotates immed | Rotates #cnt,Dn |
| ENTER | LINK |
| LEAVE | UNLINK |
| BOUND | CHECK |

**Figure 9. New iAPX 286 Instructions and their M68000 Family Equivalents**

With two addressing modes (An)+, −(An) of M68000 processors, stacks can be built anywhere. There are nine registers available at any one time for use as stack pointers. This gives the programmer complete freedom in working with stack or queue data structures.

The iAPX 286 has only the POP and PUSH instructions of its 8086 ancestor with which to work on the stack. They will only pull or push a 16-bit quantity on the stack. Byte quantities must be placed in a register and sign-extended before they can be placed on the stack. There is no real way of manipulating data on the stack. There is an enhancement which allows all of the registers to be popped or pushed, but, if some subset of the registers is needed, many PUSH and POP instructions must be run, just as they were on the 8086.

Because there is only one stack, there is only one stack pointer register on the iAPX 286. Due to the necessary use of this register for sub-

routines and interrupts, its integrity must always be retained.

There is no real help given the programmer of the iAPX 286 in solving the stack problem. An increment (INC) and decrement (DEC) instruction can be used to adjust memory pointers, but even these have serious shortcomings. They may only be incremented or decremented by a value of one. Should the size of the data being used be 16 bits, pointers must be incremented twice to step to the next value. Rather than two INC instructions being used, the usual "add immediate" instruction would be better. For source and destination pointers, even this must be done twice.

M68000 processors' addressing modes take care of most stack operations. To ease the manual manipulation of pointers and data by incrementing, M68000 processors provide the "add quick immediate" (ADDQ) and "subtract quick immediate" (SUBQ) instructions. These allow the programmer to increment or decrement any data register, address register or memory location by a value from one to eight. This gives considerable flexibility and power to the programmer when incrementing or decrementing quantities — far better than the iAPX 286.

Another problem with stack operation on the iAPX 286 is the fact that the programmer has to use a segment override each and every time data is referenced on the stack. This is necessary because there are no addressing modes which address the stack.

**String Manipulation.** M68000 microprocessors, with auto increment and auto decrement addressing modes, provide complex string manipulations with ease. Almost any instruction can be used as a primitive in a string operation. The decrement and branch on condition (DBcc) instruction is the key looping instruction. When placed at the end of one or a series of instructions, it will repeat the series until a counter is exhausted or the desired (cc) condition is met.

The DBcc instruction placed with just one other instruction, such as the MOVE, allows many simple string operations to be formed. Using DBcc with a handful of other instructions builds extremely powerful and flexible string or other repeated functions. Without the auto increment and auto decrement addressing modes of M68000 processors, these would not be possible. Some examples of string operations are shown in Figure 10. Notice the varied use of registers in the operations.

In the MC68010, some of these repeated operations execute even faster. The MC68010 microprocessor enters a special microcode routine when it fetches most single-word opcodes followed immediately by a "DBcc" which branches back to that opcode. This will latch both instructions inside the processor, executing them as designed, without further re-fetching of opcodes. Only data fetches and writes will take place as long as the processor remains in the loop. This technique speeds up these loops anywhere from 20% to 70%. It essentially provides the equivalent of thousands of special microcoded string operations.

| | | |
|---|---|---|
| MORE | MOVE.L (A3)+,(A5)+<br>DBEQ D3, MORE | block move D3 32-bit words |
| MORE | ADDX.L −(A4),−(A1)<br>DBRA D2, MORE | sum two multiprecision numbers |
| MORE | SBCD −(A5),−(A0)<br>DBRA D3, MORE | subtract two BCD digit strings |
| MORE | MOVE.B (A2,D1),(A1)+<br>MOVE.B (A5)+,D1<br>DBEQ D6, MORE | translate string A [A5] to string B [A1] using table [A2] until a null value is hit or end [count=D6] is found |

**Figure 10. M68000 Family String Operations**

The iAPX 286 has some repeatable instruction capability, but, like its forerunner, the 8086, it only provides a few "dedicated" string instructions. Other or complex string operations revert to relatively slow instruction streams. Block moves, block fills, string comparisons, string scans, and string input and output operations are given special instructions. But they use dedicated registers and are limited in flexibility.

Should an iAPX 286 programmer need to perform more complex, yet common, string operations, the normal instructions must be used with a looping control (LOOPxx) instruction. A comparison of the iAPX 286 operations which duplicate M68000 functions shown previously is shown in Figure 11.

| | M68000 (DBcc following:) | 8086/iAPX 286 (REP with:) |
|---|---|---|
| Add BCD | ABCD | — |
| Add binary | ADD | — |
| Add with carry | ADDX | — |
| Logical AND | AND | — |
| Arithmetic shift | ASL | — |
| | ASR | — |
| Bit test | BTST | — |
| and change | BCHG | — |
| and clear | BCLR | — |
| and set | BSET | — |
| Bound check* | CHK | — |
| Clear | CLR | STOS |
| Compare | CMP | CMPS |
| Divide* | DIV | — |
| Exclusive OR | EOR | — |
| Input | MOVE | INS |
| Link Stack* | LINK | — |
| Load element | MOVE | LODS |
| Logical Shift | LSL | — |
| | LSR | — |
| Move data | MOVE | MOVS |
| Move Multiple reg* | MOVEM | — |
| Move Peripheral | MOVEP | — |
| Multiply* | MUL | — |
| Negate BCD | NBCD | — |
| Negate | NEG | — |
| Negate with carry | NEGX | — |
| No Operation | NOP | — |
| Logical NOT | NOT | — |
| Logical OR | OR | — |
| Output | MOVE | OUTS |
| Rotate | ROL | — |
| | ROR | — |
| Rotate with carry | ROLX | — |
| | RORX | — |
| Scan | CMP | SCAS |
| Subtract BCD | SBCD | — |
| Subtract binary | SUB | — |
| Subt with carry | SUBX | — |
| Set conditionally* | Scc | — |
| Test and set | TAS | — |
| Test | TST | — |
| Translate | MOVE | XLAT |
| Unlink* | UNLK | — |

* Note: Limited utility.

**Figure 11. String Operations Contrast**

**Branching.** Conditional branching in programming is an all-important concept. With it, the program can make decisions based on a set of conditions. From those decisions, one of a number of actions will subsequently take place. This is a fundamental basis of computer programming. The ability of a processor to make these decisions, and the flexibility by which it can take alternate action, directly influences its utility.

M68000 microprocessors provide both conditional branching and program jumps. The jumps are unconditional and can use any addressing mode to describe the target routine. Branches in an M68000 processor are conditional and direct the processor to the next routine relative to the location of the present routine. Sixteen different conditions may be specified as the qualifier to the branch, including: always, greater than, less than, etc. The target routine may be specified as being up to +127 or −128 bytes or up to + or −32 kilobytes away. This 8- or 16-bit displacement gives quite a bit of flexibility to the programmer. The MC68020 also allows a full 32-bit displacement to the conditional branch instructions. This gives the programmer the entire address range of an M68000 processor over which to conditionally branch (16 million times larger than the iAPX 286 branch range).

The iAPX 286 has a serious shortcoming in program control operations. It only allows a conditional jump to reach up to +127 or −128 bytes away. This short range promises to be a constant reminder to the programmer of one of the limits of the iAPX 286. To perform any significant task, iAPX 286 routines will easily go over 50 instructions in length — the range of the conditional branch. Multi-level program branches will be necessary to compensate for this.

Programmers and compilers generating iAPX 286 code must constantly handle branch conditions by branching around a jump instruction to get around the small branch range, as follows:

```
            Jcc     AROUND
            JMP     FALSECONDITION
AROUND      ...
```

Notice that this forces the loss of position independent code, since JMPs allow only absolute addresses on the iAPX 286. Compilers have even further difficulties determining when forward branches must generate the two-instruction sequence.

M68000 processors provide a series of boolean byte functions that allow high-level languages easy access to the result of boolean expressions. The Scc instruction allows conditional setting of a byte in a register or memory location depending on condition codes. Thus, branching decisions can be easily implemented.

The iAPX 286 has no such feature, and its absence of bit manipulation instructions guarantees inefficient branch testing on that part.

**Memory-to-Memory Arithmetic.** Memory-to-memory arithmetic operations are yet another set of instructions on M68000 processors which are completely missing on the iAPX 286. Arithmetic-intensive programs have a high percentage of memory-to-memory requirements, such as when one array of values is added to another array. Another use of memory-to-memory architecture is for multiple precision calculations. M68000 processors allow unbounded restrictions on the data size of extended precision operations. For example, a 20-digit BCD number can easily be subtracted from another 20-digit BCD number in either memory or registers.

The iAPX 286, without direct support for such things, requires very time-consuming manipulations since, not only are 32-bit operations missing, but all values must be loaded into registers and the results stored back. The following example shows the times for a single pass through an extended precision operation. The iAPX 286 must use a decrement so as not to disturb the carry. The decrement only subtracts one, so it must be coded twice. Also note that the LODS requires both a dedicated source index register (SI) and a dedicated segment register (DS), whereas any address registers may be used on M68000 processors.

As usual, we not only find that M68000 code is shorter and easier to understand, but significantly faster as well.

**The iAPX 286 Single Accumulator.** The ancient lineage of the iAPX 286 clearly shows up in the many operations which can only be performed in the AX register. This is a throwback to the earliest days of microprocessors, indeed the earliest days of computers, when only a single accumulator was existent. It was entirely justifiable back then to provide one accumulator, since the circuitry for just that element made up a large portion of the computer and the gating required to emulate multiple accumulators was prohibitive.

Today, however, there is no excuse for such a restriction. The iAPX 286 AX register serves as a dedicated accumulator for such important functions as multiply and divide. Each and every multiply and divide must be done only in this one 16-bit accumulator. Most of the multiplies and divides do not even allow an immediate operand.

| M68000 Family | | | | iAPX 286 | | | |
|---|---|---|---|---|---|---|---|
| LOOP | ADDX.L | -(An),-(Am) | 30 | LOOP | LODS | WORD | 5 |
| | DBRA | Dn,LOOP | 2 | | ADC | [BX],AX | 7 |
| | | | | | DEC | BX | 2 |
| | 32 Cycles per Long Word | | | | DEC | BX | 2 |
| | | | | | LOOP | LOOP | 10 |
| | | | | 26 X 2 = 52 Cycles per Long Word | | | |
| Results in microseconds: | | | | | | | |
| MC68010 | | (12.5 MHz) | 2.56 | iAPX 286 | | ( 8.0 MHz) | 6.50 |
| MC68010+MMU | | (10.0 MHz) | 3.80 | iAPX 286 (equal speed memories | | | |
| MC68020+PMMU | | (16.0 MHz) | 1.06 | 200 ns) | | | 9.75 |

As if this were not enough, there are a host of other functions which also require the dedicated use of the iAPX 286's accumulator. All I/O operations, string operations, translates, and decimal and ASCII arithmetic operations must use the same accumulator. Another reason why the AX accumulator is so overworked is that there are some special instruction types which work more efficiently with the AX accumulator than with the other registers. Compares, adds, subtracts and logical operations have special opcodes for more efficient access to the AX accumulator than to the other registers. The result of all this is massive congestion.

For example, just to compute a subscript address during a sequence of arithmetic calculations means that the intermediate value in the AX accumulator must always be saved and restored over the subscript calculation.

In almost every instance, identical programs coded on both the 8086/iAPX 286 and M68000 processors show significantly more lines and bytes are required for the Intel processors. This is due, in part, to the dedicated register scheme of the Intel architecture.

Benchmarks such as the EDN Quicksort dramatically reveal the differences between the stark simplicity of the M68000 code versus the undecipherable machinations required by Intel. (See Appendicies D and E.) A single, quick glance at the two versions is enough to convert the most ardent skeptic. It is obvious that the Intel programmers had to have put in several times the effort over that required of M68000 programmers.

## CODE COMPATIBILITY ACROSS A FAMILY

There is only one real justification for extending a processor's architecture far beyond where it was originally designed to go, with all of the compromises that must be made. That is, if the programs written on the original processor must also run on the extended processors.

To do this, the programs must run exactly the same. If they do not, then each piece of code has to be individually examined to determine what to modify so that it will perform the same operation.

This is a significant chore, and can often consume more time than would be needed to rewrite the same routine for another processor. If just one condition code operates differently for any number of instructions, the task of manually verifying all code is required. If sequences of code will not execute the same on the upgraded processor, these sequences must be found and modified so that they will run.

Trying to retrofit an existing architecture with code-compatible upgrades is rather difficult unless the architecture was designed for such upgrades. Even then, there is a limit to where the compromises made for code compatibility become too restrictive to really significantly improve the overall utility of the instruction set.

The best opportunity for code compatible upgrades is on products which, in their initial design, properly anticipated future needs in enhancements. Such a product will easily adapt to the improvements without having to compromise in register selection, addressing capability, generality and special conditions.

## M68000 FAMILY COMPATIBILITY

The M68000 Family of processors started with the MC68000. The architecture for the MC68000 was an original design. Backward compatibility to earlier 8-bit Motorola microprocessors was traded away for the significantly improved performance of advanced, general instructions. Without being bound by older, more primitive register combinations, addressing schemes and spaces, and limited philosophies, the designers of the M68000 Family developed a completely new, advanced architecture. This architecture was designed for future programming needs, not the past.

In designing an architecture for future needs, room was left for the enhancements to address those needs. Not all operations that might be desired could be implemented in a producible microprocessor. The fundamental functions were included, designed with allowance for desirable enhancements which could be implemented as technology permitted. Instructions were designed so that they were not limited in concept to their current form, but could be easily enhanced to fulfill the requirements of future programmers.

Instructions in the M68000 Family are in a very

flexible, fundamental form. To provide more specific operations would limit both programmer duties today and the expansion of operations later as the needs arise. For instance, the M68000 Family does not require specific string instructions as its instruction set is so flexible, so fast, that it outperforms the special purpose instructions that are required of the limited 8086/iAPX 286 architecture. Also, 32-bit addressing was incorporated into M68000 Family architecture, even though only 24 bits of it are presented in the MC68000.

Motorola is now seeing the fruits of some of its earlier planning in the MC68010 and the MC68020 microprocessors and in the wide industry acceptance of its M68000 Family.

As previously mentioned, designing a microprocessor to be code compatible with an earlier processor can be advantageous. However, the extent to which the code is compatible becomes extremely important when weighing the benefits of such compatibility. If there is not 100% compatiblility, or too many restrictions are forced on the programmer to maintain compatibility, then it essentially makes the processor less useful than a completely different processor.

To analyze just how compatible one processor is with a previous processor, you must examine the differences in the code and its execution. In the analysis, it is important to note the impact of the differences on a programmer. To require a programmer to search every routine and assure that a particular condition code is not used after a non-compatible instruction is executed is not realistic.

Application programs are the most likely type of programs that will need to be transferred from an old processor to a new one. These programs typically include the routines that perform various tasks essential to the assigned job of the processor. They include everything from number crunching routines, data movement and manipulation, and so on.

Utility and operating system routines are also likely to be transported to the new processor. Though these types of programs need to be compatible, they are also typically subject to and candidates for change to run more efficiently on the new processor. This is because the original equipment manufacturer (OEM) frequently includes an operating system in the end product, and the operating system is so frequently executed.

An operating system is really overhead as far as completing the assigned task is concerned. Therefore, the quicker the operating system performs its duty, the more efficient the system is at handling its designed purpose. By incorporating some of the new features of a new processor into an older operating system, efficiency is improved.

**MC68000 to MC68008 Compatibility.** The MC68000 has a 100% code compatible 8-bit counterpart, the MC68008. These processors share the same instruction set, which is a basis for the later microprocessors. All programs written for the MC68000 will execute identically on the MC68008, and vice-versa. Every operation is identical in each processor.

**MC68000 to MC68010 Compatibility.** The MC68010 is an M68000 Family microprocessor which does everything the MC68000 can do, and more. In order to put additional features in the MC68010, two compromises had to be made that could potentially affect the execution of supervisor state programs (user programs are not affected). Except for these two minor differences, all MC68000 code will run identically on an MC68010. There are some new instructions available on the MC68010, which add capabilities that were not on the MC68000.

**MC68000 to MC68020 Compatibility.** The MC68020 microprocessor is an enhancement of the MC68010. It contains the instruction set of the MC68010 with significant additions. The MC68020 also has a co-processor interface and an on-chip instruction cache. The MC68020 is truly a 32-bit microprocessor with:

| | |
|---|---|
| 32-bit Arithmetic Units | 32-bit Registers |
| 32-bit Logic Units | 32-bit Data Bus |
| 32-bit Address Bus | 32-bit Data Types |

**Application Level Object Code Compatibility.** All MC68000 application (user) level object code is guaranteed to be 100% compatible when run on the MC68008, MC68010, MC68020 and all planned future M68000 processors. This means that any media containing MC68000 application code will execute the prescribed

function on the MC68020 the same as if run on the MC68000.

Other than two minor differences between the MC68010 and the MC68000, there are no further deviations from the execution of the original instructions of the MC68000 at the supervisor level; only additions. These two changes have only minimal impact in all existing MC68000 software.

**MC68010 Change 1.** The first change was made to ease operation of the MC68010 as a virtual processor. The last remaining unprivileged instruction (MOVE from SR), which allowed a user-level program to view the supervisor-restricted resources [i.e., supervisor stack pointer, and the trace (T) bit supervisor (S) bit, and interrupt ($I_2$-$I_0$) status bits], was made a privileged instruction.

To allow this operation in the MC68010, a move from condition code register (MOVE from CCR) instruction was added. This instruction may be used to replace the MOVE from SR instruction in MC68000 code that needs to run at the user level in an MC68010, since the T, S, and I bits are not to be used.

Another way of making the MC68000 code run with MOVE from SR instructions in the user mode is simply to extend the privilege violation trap handler program with a small routine.

**MC68010 Change 2.** The other condition that could vary the execution of MC68000 supervisor code on an MC68010 deals with the handling of exceptions (traps and interrupts). While a problem should not arise, some exception handlers may perform unusual operations which need to be checked.

The possibility for incompatibility is due to the fact that the MC68010 places one additional word on the stack. Again, this change does not affect user programs.

Affected supervisor-level programs need only have a few instructions changed to run on an MC68010. The modification is simply a matter of increasing by two the displacement used in the addressing mode when accessing the stack data beyond the return address from the exception routine.

The two described conditions are the only differences that could possibly alter the way an MC68000 program executes on an MC68010. One simple routine added in the operating system will handle the first case, and good programming practices or changing a few locations in some operating system routines will handle the second case.

## iAPX 286 Compatibility

The iAPX 286, as an outgrowth of the 8086/8088, is promoted as being upward compatible with those two processors. Indeed, it retains most of the registers, addressing modes, and instructions of the 8086. However, the exact functionality of the instructions is not the same in some cases. Worse, some of the concepts introduced by the native mode of the iAPX 286 not only make it incompatible with the 8086, but, should one use 8086 code in an iAPX 286 system, those new concepts cannot be used without a comprehensive rewrite of the programs. Of course, this completely destroys the intent of code compatibility.

**Operational Modes.** The iAPX 286 has two operational modes: compatibility or real address mode, and the native or protected address mode. In the compatibility mode, the iAPX 286 is designed to run 8086 code exactly, with a few new instructions added. The 8086 code should run the same (if not faster) with a possibility of a difference where the 64K byte segment boundary limit is pressed.

**Segment Wrap.** In the 8086 it is perfectly legal for a word of data or code to be accessed from the offset $FFFF, getting the high byte from $FFFF and the low byte from $0000 (wrapping around in the segment rather than flagging a overrun fault). In the iAPX 286, this specific access would not be allowed, causing a special trap to occur. The occasion for this should be infrequent, and a mechanism to correct it is provided in the trap.

**Bus Locking.** Another possibility for iAPX 286 and 8086 systems to not run the same is due to the automatic assertion of the bus lock signal upon execution of the exchange (XCHG) instruction in iAPX 286 systems. This could possibly cause a system to halt if it were not decoded in hardware. In the 8086, the LOCK

prefix may precede any instruction. In the iAPX 286, the LOCK instruction is privileged and will cause a trap if executed at a lower privilege level. The operating system will have to emulate the desired, bus-locked operation.

**Protected Mode Operation.** A significant problem with compatibility exists in the iAPX 286 when it is run in the native, or protected, mode. This is the mode that the iAPX 286 is designed to run in, affording the user all of the features it contains.

Unfortunately, using most of these new features so erodes the concepts on which the original 8086 programs were written that they will not run properly without alteration. The problem is not that the code sequences will not execute, but, by executing as written, they will very likely violate the principles of the new privilege and protection features.

Because the 8086 has no concept of execution privilege levels, any program written for the 8086 may have free access of any resource. To perform its assigned chore, an application program might read some I/O ports, perform some function, and then wish to save the data in a file. In a native iAPX 286 system, a special I/O handling routine must make all I/O accesses at a designated I/O privilege level. The application program running at a lower level would massage the data and then an operating system task would store the data on disk at the operating system privilege level.

**8086 Application Program Compatibility.** How much of the original routines could be retained in the iAPX 286? The I/O handler would have to be rewritten to run at the I/O level within the task definition. The disk driver routines, which will now be run by the operating system, must be rewritten to run at the zero privilege level. Perhaps the transfers to the disk I/O port will be handled by a separate routine at the I/O level. The actual manipulation of the incoming data can still be performed at the applications program privilege level only after all of the program transfers to the file and I/O routines have been redirected.

The end result is that a significant amount of work will have to be done to modify even low-level applications programs so that they will run on the iAPX 286 in its protected mode. An alternative to this would be to run the program at a higher privilege level, but to do so would completely destroy the system integrity that the privilege levels were supposed to bring in. Programs designed to run on an 8086 will run on an iAPX 286 in the native mode only after significant reworking by the programmer.

**Segment Base Address.** The fundamental change in the utility of the segments in an iAPX 286 from the 8086 can cause significant problems for many routines. In the 8086, the segment registers directly established a 20-bit base address from which to access code and data. This base was exactly predictable to any programmer in the 8086. In the iAPX 286, the segment registers **indirectly** specify a 24-bit base address. The writer of iAPX 286 code can not reliably predict what that base may be because the operating system will establish it. This is one source of problems with iAPX 286 code compatibility.

Because the 8086 has so few and dedicated registers, and the general instructions are somewhat limited, programmers frequently make creative use of the resources they have. Unfortunately, when an architecture is extended beyond where it was originally destined, changes can defeat this creativity. This is vividly illustrated in some of the Carnegie-Mellon benchmarks that Intel programmers assembled.

**Carnegie-Mellon Benchmark Code.** The Carnegie-Mellon benchmark A, an I/O interrupt kernal, as coded for the 8086, used the CS segment register to locate some counters. While this is not a good programming practice, it is quite possible in the 8086. Unfortunately, the iAPX 286 is more restrictive, and, due to its protection mechanisms, will not allow variable data to reside within a code segment. Thus, this fairly simple routine would not run on an iAPX 286.

In the Carnegie-Mellon benchmark I, a Quicksort routine, Intel programmers once again wrote code which will not run on the iAPX 286. (See Appendix E.) Here, they used program-

ming practices which they explicitly characterized as dangerous. They used the DS and ES segment registers as base registers to point to individual records which were being sorted. The resulting code will not run on the iAPX 286 in the native mode because it can not be predicted where the segment registers will be pointing.

The Intel programmers developed the code for these algorithms as any programmer would, for space savings and speed. It points out one of the serious architectural flaws in the 8086/iAPX 286 — the lack of registers (the lack of base registers in this particular algorithm). The programmer is forced to resort to using segment registers for purposes other than what they were designed. In the iAPX 286 these restrictions grow tighter, resulting in a violation of the purpose of the segment registers in the memory management scheme. It also illustrates that the iAPX 286 has only two data segment registers, the DS and ES registers.

And so, once again, the 8086 and the iAPX 286 are not code compatible, even for application programs.

**Guide for an 8086 Code Writer.** From another approach, in an Intel-written application paper, all of the things are noted that an 8086 code writer must be aware of to assure the code will run on the iAPX 286 as well. A person can only wonder about what precautions will be needed for future extentions to the architecture. This paper begins, "Due to the iAPX 286 enhancements for memory protection and virtual addressing, some 8086 programs may not work correctly when run on an iAPX 286."

Would all this be necessary if the part were code compatible? The cautions given are not trivial. Some are listed in Figure 12.

Included in the paper are considerations for high-level language compatibility. Many of these cautions are required due to the new privilege levels introduced by the iAPX 286 and the new function of the segment registers.

However, there are other difficulties. Earlier high-level languages allowed any procedure

Do not use overlapping segments.
Do not store temporary values in segment registers.
Avoid operations which may be restricted by the iAPX286 to assure system integrity, low interrupt latency, or low bus request latencies (i.e. . . . STI, CLI, HALT, and I/O instructions).
Write programs independent of operation changes required by protection (e.g., XCHG asserting LOCK, different single step priority, POPF and IRET not changing the interrupt enable flag or I/O privilege level, invisible interrupt table, or RET affected by the nested task flag in the flag word.)
Do not rely on the value pushed onto the stack by PUSH SP.

And for **operating system** upgrade:
To use the hardware task switch function or multiple address spaces, the scheduler must change.
Code which accesses task context (e.g., register state) must change.
Code which accesses the interrupt table must change.Add pointer validation code to the prologue of any procedure expecting pointers from less trusted callers.
Structure the operating system to use the iAPX286 privilege hierarchy.
Validate pointers from all callers.

**Figure 12. iAPX 286 Programming Guidelines for 8086 Compatibility (Abridged)**

to be an interrupt procedure. In the iAPX 286, no user level program may access the interrupt table and therefore it will not load properly. Pointer arithmetic in PL/M is common with all of the segment registers, offsets and addressing mode rules. Few pointer arithmetic operations are compatible between the 8086 and iAPX 286.

**Operating System Compatibility.** Operating system compatibility between the 8086 and the iAPX 286 is almost non-existent.

With the exception of running in the compatibility mode, with almost none of the new features of the iAPX 286, every 8086 operating system will have to be entirely rewritten to be used on the iAPX 286. The new privilege levels of the iAPX 286 will have to be assigned and handled by the operating system and each application program. To prevent invalid pointers from being passed to the operating system by lower level programs, the operating system must verify that any pointer passed is valid. Any code which accesses any of the vectors for interrupts must be changed, since these vectors can only be accessed by the operating system.

These are just a few of the changes that must take place in an 8086 operating system for it to run on the iAPX 286. As a general rule, a totally new operating system will have to be written for the iAPX 286.

Additionally, most application level programs will have to be rewritten to protect the various operating system and I/O operations to their proper privilege level.

**Future Operating System Compatibility.** To make matters even worse, information released by Intel indicates that yet another complete rewrite of major portions of iAPX 286 operating systems will be required to move up to the iAPX 386. This is because the iAPX 386 will handle segment faults in an entirely different manner, in that segments will not be allowed to be treated as complete entities for residency decisions. Therefore, the operation of loading a segment register will no longer be sufficient to indicate what working set of memory needs to be made resident. As a result, the iAPX 286 methods of main storage management will no longer be appropriate.

**Summary.** All in all, the iAPX 286 is a microprocessor which does use the same instructions as the 8086. It also has a few new instructions. But it has many very new concepts which need to be incorporated into the existing software. Including these new concepts into old programs is a major undertaking.

Having had no foresight as to the future growth of the 8086 family, programmers writing 8086 code had no reason to allow for such developments as privilege levels, segment integrity, memory management, and even operating system isolation. Therefore, when a new microprocessor demands that these concepts become part of the original code, a significant burden is placed on the programmer who must upgrade it.

Failure to make the required changes nullifies any benefit of moving to the new part.

## PRIVILEGE LEVEL PROTECTION

The Z8000 and the MC68000 were the first to introduce the concept of privilege to microprocessors. The MC68000 has two privilege levels: supervisor and user. Operating system functions typically take place at the supervisor level, while application programs typically execute at the user level. The supervisor level has access to all of the resources of M68000 processors, and typically all external resources. The user typically has access to almost all of the resources on an M68000 processor but has restricted access to external resources such as memory and I/O.

The distinction of privilege allows the operating system to keep control of all processor and even external functions, while protecting itself and other code, data and users from the careless activities of a poorly written or unfriendly application or user program. The MC68010 has even gone one step farther in this privilege scheme to secure the supervisor resources from even being seen by the user. This facility permits the MC68010 and the MC68020 to operate as a virtual processors.

The two levels of privilege in M68000 Family processors are more than sufficient for providing security to an advanced computer system. All operating system functions and I/O functions can easily be implemented in a secure manner, without burdening the programmer with needless overhead. Various automatic and explicit traps give control to the supervisor each time the user attempts a questionable operation. Notification of improper off-chip accesses is simply enacted with either an interrupt or the unique bus error (BERR) signal.

The iAPX 286 implements a four-level ring privilege structure. It is suggested that application programs run at the lowest level (level 3), while the operating system maintains the highest level (level 0), much the same as on an M68000 processor. A floating I/O privilege level lets I/O operations take place at any level the operating system assigns. It is suggested by the manufacturer that the operating system be segregated throughout the three highest privilege levels based on the significance of each task.

Switching between programs running at different privilege levels is effected through complex gates which are designed to assure the

security of the iAPX 286 system. Hardware assures that each privilege level may have its own set of registers and processor resources. Violation of certain rules forces traps up to the controlling operating system.

The most important thing to be said for the iAPX 286 ring architecture is that you have no choice but to implement the ring mechanism whether you like it or not.

Even worse, you must accept the segment hangups which go along with the rest of the architecture. There are some system implementers who prefer hierarchical levels in their architecture, though the vast majority do not. The Intel philosophy is one of "take-it-or-leave-it." You use the iAPX 286 definition of a ring architecture or you do not use memory management at all.

Motorola does not force any concept of memory management. You can use demand paging, demand segments, virtual versions of each of these or no memory management at all. And the best news is that absolutely NO progamming changes are required by user mode programs to implement or change over from one to another, unlike the iAPX 286, where absolutely all programs must have MMU segment loading operations.

The MC68020 adds the concept of multiple access levels to the M68000 Family. The access concept not only provides automatic module connection but also the expansion of up to 256 hierarchical levels without the cumbersome segment limitations forced by the iAPX 286.

The hierarchical levels are a superset of ring architecture with much more flexibility. For example, finer gradations are supported which allow concepts such as data abstraction to be supported. The end result is that a system designer can, with the MC68020, choose almost any type of memory management classification and, due to the flexible nature of the Motorola architecture, implement specific customizations as desired. For example, a classical ring structure can be developed with ring objects of up to 4 billion bytes in size instead of the restrictive 64K byte limits on the iAPX 286, and with several times more levels than just the four allowed by the Intel processor. On the other hand, MC68020 hierarchical levels are totally optional and do not extract any penalties for those interested in implementing, say, a classical virtual demand paging mechanism.

One could even combine the two and implement a demand paging ring structure!

The bottom line here is M68000 Family power and flexibility versus the iAPX 286 philosophy of rigidity and limitation.

## MEMORY MANAGEMENT

There are many aspects of memory management. There are many reasons for a system to require its memory to be managed. The exact concerns of the designer must be examined before the technique of memory management can be decided. Fixed schemes of memory management which only fit a particular set of criteria suffer the same fate as processors which force the programmer into certain restrictions and practices — they are undesirable, unpopular and fail.

Memory management can be required in a system to control a number of things. Ideally, the more the isolation between the memory management scheme and the programs and data which the processor will operate on, the better. The higher the level of program which must have knowledge of the memory management technique, the better. Ideally, only the highest level of the operating system kernel — or even a completely separate, intelligent memory manager — even knows that there is some form of memory management in use. That kernel manages the memory according to the chosen scheme.

Some computer systems, however, require every program — whether operating system, utility, library or application — to have knowledge of the memory management scheme. These programs have to obey certain rules of the scheme, and have to provide certain information to the manager based on each routine to be run, resulting in excessive memory management overhead. This is undesirable and unnecessary.

It is more typical in advanced computers for the application-level programs to be written as though they had free access to any resource located anywhere within the address space of the processor. Should the operating system decide that it is dangerous for the lower level application program to have access to a set of resources, the operating system will make certain that all attempts to access those resources will be blocked and only the operating system may make them.

This is one of the typical duties of the memory manager: to protect various memory spaces from unauthorized users. Another purpose of the memory manager is to translate addresses.

A microprocessor generates logical addresses, based on internal address calculations associated with instruction processing. These logical addresses might not correspond to physical memory or I/O locations, due to hardware constraints. The memory manager needs to control which logical memory or I/O locations correspond to actual physical locations, and properly translate the logical addresses to physical addresses.

The use of a smaller (or even larger) physical memory space than the logical address space is called "virtual memory." The exact partitioning of physical memory, and the algorithms used, comprise the virtual memory scheme. A variety of virtual memory techniques exist and are popular. Each has positive and negative aspects, and the correct virtual memory technique for each system must be selected.

A third form of memory repositioning, banking or segmentation, is not so much a technique of managing memory as it is a technique of expanding memory. Banking and segmentation were initially introduced in computer designs as a means of working around an otherwise small address space. They allow a smaller size address space to be duplicated a number of times by categorizing each duplicate address space by its purpose. This purpose may be as broad as classification of program versus data space, or as specific as data space no. 2 for user no. 3.

The advantage of bank or segment memory schemes is that they allow more memory to be accessed than the naked architecture will allow. This is done by appending the bank or segment address to the beginning of the address. This requires no real change in the existing processor architecture — only additions to the hardware and bank/segment control routines.

While banking and segmenting expand the addressing range of a processor, and implementation is easy in the architecture, actual use of a banked or segmented address space is quite involved. The problem with banking and segmentation has always been that of keeping track of which bank or segment the program or data needed is in. Not only is the programmer saddled with MMU loading instructions, but the overhead for loading segment registers (two to four microseconds for the iAPX 286) is always present every time the program executes.

As reasonable execution time is possible only by keeping a small number of bank or segment registers active, keeping those registers pointing at the most frequently used information is quite a chore. Then, every time the information needed by the processor cannot be found in a current bank or segment register, the appropriate bank or segment must be swapped with an existing register.

It doesn't take too many of these register swaps to turn into quite a lot of overhead. And, as we have seen before, the iAPX 286 has only two segment registers for data.

**iAPX 286 Segments Limit Compatibility with Today's Systems.** Just now being introduced are single-user computer systems which routinely handle data structures of several megabytes in size. New innovations such as the Smalltalk language require the manipulation of huge linear memory elements to offer the powerful concepts they provide.

The iAPX 286, with its 64K segment limit and lack of true virtual memory capability, is entirely unfit to be used as a host microprocessor for such systems.

For example, with such advanced systems a single item symbolized by an icon may represent a complete collection of disparate entities such as text files, spreadsheet data, graphic drawings, code package elements, and, most importantly, other icons which themselves may contain such items. Such a system treats that single icon as a fully addressable block which may be many megabytes in size. If an item is selected (activated) from that icon, then it is also addressed and accessed in like manner.

This is why a huge linear address range, as on M68000 processors, is so critical for efficient execution of such objects. Breaking such items down into artificial segments of 64K on the iAPX 286 is just not practical. The many items which, either as single entities or as groups are larger than 64K, would create havoc with segment loading and testing requirements, as the iAPX 286 at every access into such an item would be required to generate a complete subroutine call to guarantee that the proper segment is based in a segment register. The iAPX 286 program cannot simply access any address it wants, as can those on an M68000 processor, but instead must pass such an address to a subroutine to determine which segment is the proper one to base.

The Intel iAPX 286 Preliminary User's Guide gives an illustration of what must be done to access large data structures on the iAPX 286. The code is so lengthy (58 bytes) that it must be incorporated as a subroutine and usually will be passed a descriptor since that results in shorter code and due to the fact that there are not enough registers on the iAPX 286 to pass all the parameters required. The following sample code shows the access required to address a large array of four byte values with a 32-bit index on the iAPX 286, the MC68000 and the preliminary figures for the MC68020. The iAPX 286 routine is out of the Intel publication. Note that for the Intel code to work, the operating system must support the incrementing on segment identifications to address continuous segments (something not offered by Intel's own operating system).

**Large Array Access (iAPX 286, MC68000, MC68020)**

| MC68000/MC68010 | | Bytes | Cycles |
|---|---|---|---|
| MOVE.L | Index,Dn | 4 | 16 |
| ASL.L | #2,Dn | 2 | 12 |
| opcode | offset(Dn,Base) | 2 | 6 |
| TIME (12.5 MHz) = 2.72 us | | 8 | 34 |

| MC68020 | | Bytes | Cycles |
|---|---|---|---|
| MOVE.L | Index,Dn | 4 | 6 |
| opcode | offset(Dn,Base,Dn:2) | 4 | 4 |
| TIME (16 MHz) = 0.625 us | | 8 | 10 |

| iAPX286 | | Bytes | Cycles | | |
|---|---|---|---|---|---|
| MOV | BX,Descriptor | 3 | 2 | Descriptor | DW Structsize |
| MOV | AX,Index+2 | 4 | 5 | | DW Offset |
| MOV | CX,Index | 4 | 5 | | DW SegAsize |
| CALL | RESOLVE | 3 | 10 | | DW SegA |
| LDS | SI,X | 4 | 22 | X | DD *-* |
| opcode | [SI] | 0 | 0 | | |
| | | 49 | 119 | (RESOLVE) | |
| TIME (8 MHz) = 20.37 us | | 67 | 163 | | |

```
* RESOLVE IS ADAPTED FROM INTEL iAPX286 PRELIMINARY USER'S
* MANUAL (FIGURE 8-10)
*   INPUT:  DS:BX — DESCRIPTOR FOR ARRAY WHICH
                    CONTAINS
*               WORD SIZE OF STRUCTURE              +0
*               WORD 1ST SEGMENT OFFSET             +2
*               WORD SIZE OF SEGMENT                +4
*               WORD SEGMENT BASE ID                +6
*       AX = HIGH WORD OF 32-BIT INDEX
*       CX = LOW WORD OF 32-BIT INDEX
*   OUTPUT:  X + SEGMENT AND OFFSET POINTER TO ARRAY ELEMENT
```

| | | | Bytes | Cycles | |
|---|---|---|---|---|---|
| RESOLVE | MUL | AX,[BX] HIGH X SIZE | 3 | 24 | |
| | MOV | DI,AX | 2 | 2 | |
| | MOV | AX,CX | 2 | 2 | PATH1 = 125 CYCLES |
| | MUL | AX,[BX] LOW X SIZE | 3 | 24 | PATH2 = 135 CYCLES |
| | ADD | AX,[BX+2] +OFFSET | 3 | 7 | PATH3 = 99 CYCLES |
| | ADC | DX,DI | 2 | 2 | |
| | JNZ | NEW_SEG | 2 | 11/3 | AVG 119 CYCLES |
| | CMP | AX,[BX+4] | 3 | 7 | |
| | JAE | NEW_SEG | 2 | 11/3 | 49 BYTES |
| | MOV | CX,[BX+6] | 3 | 5 | |
| | MOV | X+2,CX | 4 | 3 | |
| | MOV | X,AX | 4 | 3 | |
| | RET | | 1 | 14 | |
| NEW_SEG | DIV | [BX+4] | 3 | 26 | |
| | ADD | AX,[BX+6] | 3 | 7 | |
| | MOV | X+2,AX | 4 | 3 | |
| | MOV | X,DX | 4 | 3 | |
| | RET | | 1 | 14 | |

As can be seen, the inordinate amount of segment handling adds an order of magnitude overhead for accessing large data structures such as the array.

In fact, the iAPX 286 is over 7 times slower than the MC68000 or MC68010, and a huge 32 times slower than the MC68020. Even worse for the iAPX 286 is the fact that only two data segments can be based at any one time (and even these are required for other dedicated operations), further aggravating the code and time overhead.

**Incrementing a Smalltalk or Graphics String Pointer.** To consider just a simple thing as scanning a string in a Smalltalk or large graphics environment requires unbelievable gymnastics on the iAPX 286, while the overhead of the autoincrement addressing mode on the M68000 processors is zero. In other words, any M68000 processor is so powerful that it increments address registers by the respective data type size in parallel with the instruction execution.

Contrast this with what is required on the iAPX 286. First, we will assume that the operating system is smart enough that it can assign consecutive segment identifications for contiguous large data structures. (This in itself requires that massive extensions be added to such things as language compilers, assemblers, linkage editors and dynamic memory allocation within the operating system to support such a capability.)

| | iAPX 286 | | M68000 Processors |
|---|---|---|---|
| | INC BX | 2 Cycles | (An)+ 0 Cycles |
| | JNE LBL | 16/4 Cycles | Autoincrement addressing mode |
| | MOV DS,AX | 2 Cycles | |
| | INC AX | 2 Cycles | |
| | MOV DS,AX | 20 Cycles | |
| LBL | | | |
| Result: | 10 Bytes 18 or 30 Cycles | | 0 Bytes 0 Cycles |
| | 1 Extra Register | | No Extra Registers |

The difficulty in implementing such a trivial thing on the iAPX 286 doesn't even begin to give an indication of the more complex manipulations required when a complete scalar item straddles a 64K boundary and must be accessed by a high-level language. Several pages of code are required to accomplish that feat on the iAPX 286, and they are omitted here.

**Artificial Intelligence Research Cannot Use the iAPX 286.** As reported in the Wall Street Journal on August 19, 1982, page 19, researchers say that the iAPX 286 cannot be used for artificial intelligence applications:

> ". . . The 68010 is an improved version of Motorola's 68000 microprocessor, which is forming the basis of dozens of new desktop computers.
>
> ". . . International Business Machines Corp. and Hewlett-Packard Co. recently began selling computers based on the 68000, too. At least four groups of scientists — at Yale, Massachusetts Institute of Technology, the University of Utah and the Rand Corp. — are at work on or say they have completed Lisp dialects for the 68000.
>
> ". . . An informal survey of the major centers of artificial intelligence research found no one trying to develop a Lisp dialect for the microprocessors with which the Intel Corp. competes against Motorola. Scientists surveyed say Intel's present circuit, the 8086, can't deal with enough information at one time to be useful in major artificial intelligence programs and they dislike the design of the improved version [iAPX 286] due from Intel soon."

Large data structures and efficient use of pointers are absolutely critical for proper operations in such environments. The fact that the iAPX 286 cannot handle virtual memory access faults, but forces segment loading checks and their associated overhead (shown earlier), restricts its use to the 8-bit world of 64K addressing. Each segment register load on the iAPX 286 requires 20 clock cycles. And the iAPX 286, with its small and special purpose register set, requires a massive amount of segment register manipulation as shown in the EDN Quicksort benchmark — where the entire MC68000 execution time is faster than just two of the segment loading instructions for the iAPX 286 native mode version.

**Dynamic Storage Areas and Sophisticated Software Systems.** Many benchmarks around are extremely simple and do not really tax the architecture of even the simplest 8-bit microprocessor. However, real world computer systems require the successful execution of large and sophisticated program modules with their associated linkage. As a result, many weaknesses are hidden and only found after a system is fully implemented.

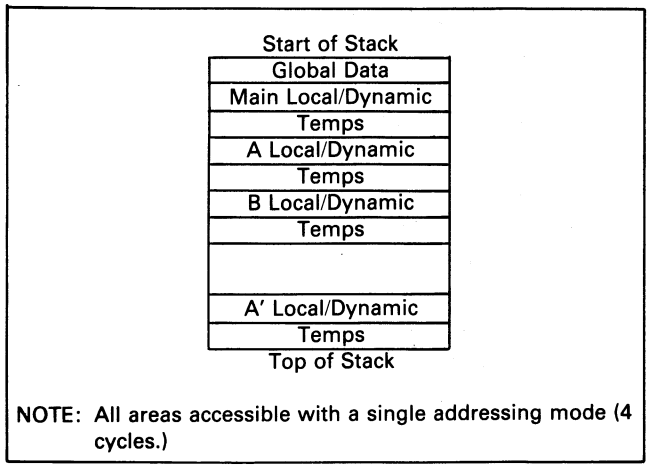
Intel's limited segment sizes have several such side effects which need to be brought to light. By exposing just a single example here, it will be seen that the problems caused are far from trivial, and it behooves anyone investigating the use of various microprocessors to ferret out just such anomalies prior to selection.

A sophisticated system requires the interaction of tens or maybe even hundreds of modules. Each module, when called, demands access to new and unique dynamic storage areas for such things as large local variable structures and stacks for temporary work areas, dynamic data allocation and subroutine return linkage. Such dynamic areas cannot be static (that is, allocated at linkedit or module load time) as the module may be called directly by itself, indirectly by routines called by itself, or by a completely different task. Dynamic areas are also required for recursive programmimg techniques. In fact, most modern programming languages—such as Ada, Pascal, and Lisp — have all modules recursive as a matter of language definition.

On any M68000 microprocessor, the matter is simply and automatically handled via its large linear address space.

As a routine is called, it merely allocates the dynamic memory it requires on the current stack and continues on its way. The powerful LINK instruction allocates the fixed stack re-

quirements for local dynamic data items and sets up any one of the address registers as a frame pointer. Since each and every executing module uses the stack dynamically, there is never any wasted space or management overhead (see Figure 13). The operating system can easily assign an initial stack and then increase it by a set amount whenever required as detected by virtual memory faults. Motorola Pascal, for example, has the main program allocate all global data initially on the stack. Then, as each function or subroutine gains control, it simply uses the LINK instruction to allocate its total requirements directly on the same stack. At termination, the UNLK instruction does the simple work of freeing the data.




NOTE: All areas accessible with a single addressing mode (4 cycles.)

**Figure 13. Dynamic Storage Allocation on M68000 Processors**

The iAPX 286 has no obvious way of handling dynamic areas, a problem to contend with in the iAPX 286 runtime enviroment.

When the iAPX 286 ENTER instruction attempts to extend the stack, it fails whenever the total dynamic area in use by all modules becomes greater than 64K. Thus, Intel's attempt to include a few M68000-like instructions in the iAPX 286 instruction set does nothing but point up the inadequacy of the iAPX 286 architecture to support today's sophisticated requirements. Even worse is the fact that the stack size limitation means that the local dynamic variables often will not fit into a single 64K segment, and thus some artificial means must be developed for allocating and freeing such storage. Complicating the problem for the iAPX 286 is the fact that segment values cannot be "played around with" as on the older 8086.

The end result of all this is that very cumbersome methods must be employed to support dynamic variables and stacks on Intel architectures. What takes a single instruction in a module to allocate or free the stack and variable space on any M68000 microprocessor, takes entire subroutines on the iAPX 286. And for dynamic memory segments to be efficiently reused, these subroutines must be called both before and after each module execution, since the dynamic order of module invocation cannot be predicted. Figure 14 gives an indication of the extra descriptors and overhead required for dynamic storage management with the iAPX 286. And remember: each descriptor load to reference any of these areas is over two microseconds (assuming 80 nanosecond memories).

As if this were not enough, the iAPX 286 high-level language runtime environment must often interface with the iAPX 286 operating system via call gates, since the descriptors pointing to blocks of memory segments can only be assigned and manipulated by higher priority protected routines. The total price exacted for such heavy encumbrances represents at least several orders of magnitude increase in processor time, just to manage the simple task of dynamic stack/variable allocation.

This is the very same problem which caused the Intel iAPX 432 to execute a recent set of benchmarks more than ten times slower than the MC68000.

The following examples illustrate the code required for all dynamic memory management.

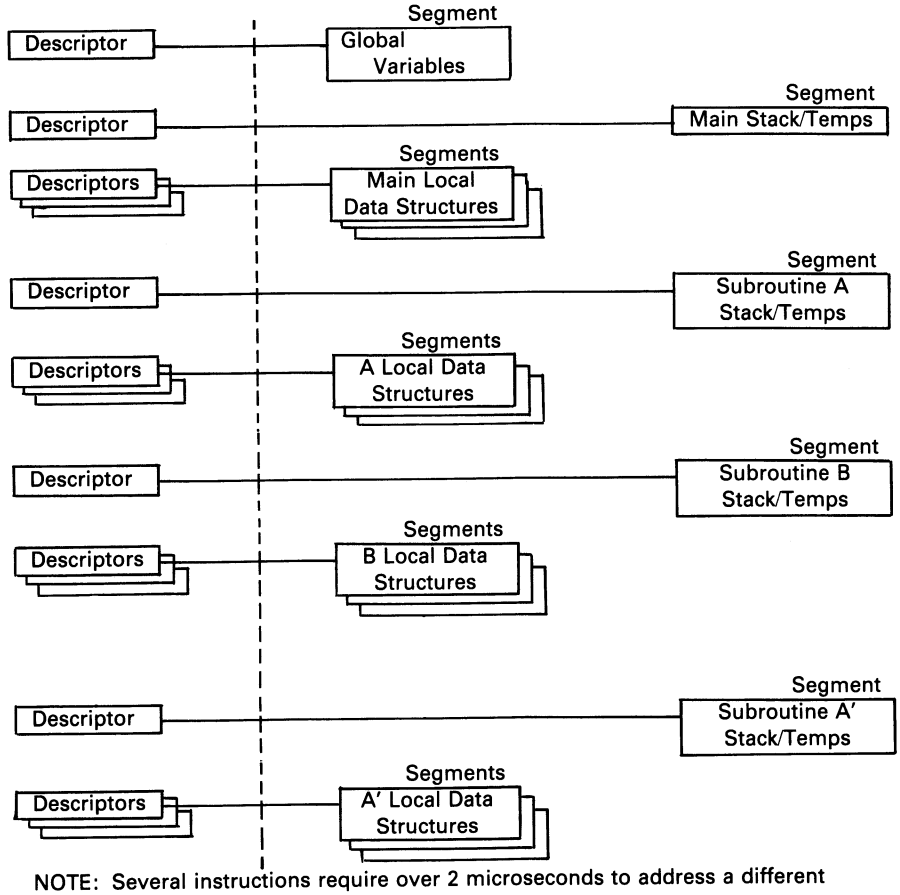
**MC68000 Method**

```
* ALLOCATE DYNAMIC AND LOCAL MEMORY
  LINK      #SIZE,An                    16 Cycles
    .
    .       PROGRAM BODY
    .
* FREE DYNAMIC AND LOCAL MEMORY
  UNLNK     An                          12 Cycles
```

**iAPX 286 Method**

```
; ALLOCATE OUR OWN STACK FOR TEMPORARIES AND SUBROUTINE CALLS
  CALL      NEWSTACK                    Hundreds to Thousands of Cycles
; ALLOCATE LOCAL DATA STRUCTURE SEGMENT
  MOV       AX,SIZE                     3 Cycles
  CALL      ALLOCSEG                    Hundreds to Thousands of Cycles
; REPEAT ABOVE FOR EACH LARGE STRUCTURE REQUIRED
    .
    .       PROGRAM BODY
    .
; FREE LARGE STRUCTURES
  MOV       BX,DESCRIPTORADDRESS        7 Cycles
  CALL      FREESEG                     Hundreds to Thousands of Cycles
; REPEAT ABOVE FOR EACH STRUCTURE OBTAINED

: FREE DYNAMIC STACK AND RETURN TO CALLER
  JMP       FREESTACK                   Hundreds to Thousands of Cycles
```




NOTE: Several instructions require over 2 microseconds to address a different segment.

**Figure 14.. Dynamic Storage on the iAPX286**

**Massive Descriptor Overhead for All iAPX 286 Native Mode Operating Systems.** There are many hidden pitfalls awaiting the operating system programmer who is forced to use the iAPX 286 method of MMU descriptor control, besides the fact that most of his current operating system will need substantial rewriting. Problems abound, as shown in Intel's own iAPX 286 Preliminary User's Manual.

One such example is that even before you build a descriptor you must first address it to build it. This entails the creation of yet another temporary descriptor reserved for just such activities to allow access to the new one to be built. Temporary descriptors are required even to just examine another descriptor. The added execution time overhead of such little nuances, and there are many, comes as a grim discovery for those who examine the iAPX 286 for their operating system base.

## I/O INTERRUPTS ON THE MC68000 AND THE iAPX 286

The following code shows the EDN benchmark requirements for the iAPX 286 benchmark A. The original code "as published" will not work on the iAPX 286 because the protected mode will not allow code space to be written to as was done on the 8086 (a decidedly uncomfortable incompatibility for Intel to acknowledge). The earlier EDN code would receive a protection exception 13 due to an access rights violation.

The protected iAPX 286 environment allows three ways to handle an interrupt. The first and "fastest" method is to have a single task system which runs all program segments at the same priority and allows the interrupt to be taken also at the same priority. The second method processes the interrupt at a higher priority level (which would ordinarily be the case) and also allows multiple tasks to be used. For small, single-purpose multitasking systems, this would be the norm. The third method is via a task switch. Since this third method is quite inefficient, it would only be used for very unique conditions where a complete high priority task is required to process a special case of infrequent interrupts.

**M68000 Version.** M68000 Family interrupt code is simple and fast. Intel presents the same benchmark in their "iAPX 186,286 BENCHMARK REPORT" of October 1982. In order to make it appear that the iAPX 286 is faster, Intel incorrectly claims that an interrupt handler on the MC68000 must change MMU segment registers during interrupt processing.

The truth is that the Motorola MC68451 MMU itself detects a change from user to supervisor state and uses the supervisor descriptors automatically. Since the MC68451 does not have 64K limits on its descriptors, as does the iAPX 286, operating systems using the MC68000 typically permanently map their entire resident operating system memory requirements with just a few permanent descriptors. Thus, the entire operating system runs with never once causing a descriptor fault and incurring the associated overhead. Contrast this with the constant descriptor loading required of an iAPX 286 operating system.

Another reason for simplicity of the MC68000 interrupt routine is its absolute address capability. Notice that the iAPX 286 must laboriously save registers and rebank its data segment to achieve pointer capability, something freely available with a simple addressing mode on the MC68000:

```
          Cycles

* ENTRY   44
           16   ADD  #1,COUNTER   INCREMENT COUNTER
           20   RTE                RETURN FROM
           80                      EXCEPTION
```

**Fastest iAPX 286 Version.** The fastest handler is the same priority handler, and can only be used in very primitive, single-task iAPX 286 systems. The handler must save at least one of the interrupt routine's segment and base

registers so that the interrupt counters can be properly based:

```
                    Cycles

* ENTRY              41
                      3     PUSH    SI          ;SAVE INDEX REGISTER
                      3     PUSH    DS          ; AND SEGMENT REGISTER
                     21     LDS     SI,CNTPTR   ;BASE COUNTER
                      7     INC     [SI]        ;INCREMENT IT
                     20     POP     DS          ;RESTORE SEGMENT
                      5     POP     SI          ; AND INDEX REGISTER
                     32     IRET
                    132
```

**Most Common iAPX 286 Version.** Normally, interrupts must be handled at a higher priority than problem program code, and thus different interrupt and return times come into play since a different stack is now forced to be based by the protection scheme. The code itself remains the same. However, the interrupt routine must only reference the global descriptor table (GDT), since any task may have been interrupted.

```
                    Cycles

* ENTRY              79
                      3     PUSH    SI          ;SAVE INDEX REGISTER
                      3     PUSH    DS          ; AND SEGMENT REGISTER
                     21     LDS     SI,CNTPTR   ;BASE COUNTER
                      7     INC     [SI]        ;INCREMENT IT
                     20     POP     DS          ;RESTORE SEGMENT
                      5     POP     SI          ; AND INDEX REGISTER
                     56     IRET
                    194
```

**Task Switch iAPX 286 Version.** The task changing would be rarely used. Only when a very special and infrequent exception needs to trigger a complete high priority task switch would this come into play. However, any iAPX 286 operating system allowing programs to supply direct interrupt handlers must use this method to guarantee system integrity.

```
                    Cycles

* ENTRY             167
                      3     PUSH    SI          ;SAVE INDEX REGISTER
                      3     PUSH    DS          ; AND SEGMENT REGISTER
                     21     LDS     SI,CNTPTR   ;BASE COUNTER
                      7     INC     [SI]        ;INCREMENT IT
                     20     POP     DS          ;RESTORE SEGMENT
                      5     POP     SI          ; AND INDEX REGISTER
                    170     IRET
                    397
```

**Summary.** Intel claims (in Advanced Information Book AFN-02060A) that cycle times will average 5% more due to instruction fetch wait from memory. Also, these times assume zero wait states, which require at least 80-nanosecond memory. The following chart shows the comparison of the two systems under a wide variety of conditions:

**Interrupt Increment and Return**

| | | | |
|---|---|---|---|
| MC68000 without MMU | | 12.5 MHz | 6.4 microseconds |
| MC68000 with MMU | | 10.0 MHz | 9.3 microseconds |
| 8086 | | 10.0 MHz | 11.6 microseconds |
| iAPX 286 80 ns memory | (same priority) | 8.0 MHz | 16.5 microseconds |
| iAPX 286 same memory speed | (same priority) | 8.0 MHz | 24.7 microseconds |
| iAPX 286 80 ns memory | (to other priority) | 8.0 MHz | 24.2 microseconds |
| iAPX 286 same memory speed | (to other priority) | 8.0 MHz | 36.4 microseconds |
| iAPX 286 80 ns memory | (task switch) | 8.0 MHz | 43.1 microseconds |
| iAPX 286 same memory speed | (task switch) | 8.0 MHz | 64.6 microseconds |

Notice that the 8086 beats the iAPX286.

Another important statistic is the longest interrupt latency possible, from the generation of an interrupt to the first instruction of the interrupt handler. The worst case for both systems is:

| MC68010 | Cycles | iAPX 286 | Cycles |
|---|---|---|---|
| DIVS.W d(An,m) | 136 | CALL task__gate | 188 |
| I/O INTERRUPT | 46 | I/O INTERRUPT (TASK GATE) | 170 |
| | 182 | | 358 |

Finally, assuming that the iAPX286 does not allow the use of task gate interrupts, worst case timing is:

| | Cycles |
|---|---|
| Call task__gate | 188 |
| I/O interrupt | 81 |
| | 269 |

**Intel Architecture Foils Intel's Own Programmers.** A prime example of the problems caused by the 8086/iAPX 286 architectural weaknesses is an examination of the code produced by Intel's top programmers for the EDN benchmark competition. The last benchmark of that series (benchmark K) inverts a square bit matrix of size seven by seven. That competition was originally published in the April 1981 edition of EDN.

Intel's time for that benchmark was 820 microseconds for the 8086 versus 368 microseconds for the MC68000. The Motorola version was a very simple one, where a virtual bit address was created by shifting the byte address over by three and appending the bit number thereto.

A second edition of the benchmarks was run in EDN's September 16th issue of the same year, and Intel submitted a new version for benchmark K. In it, they attempted to create a virtual bit address just as Motorola had done. However, they ran into one major problem: their new benchmark would not work with the vast majority of possible parameters since, with only 16-bit registers, when they shifted left by three they were throwing away the top three bits of the bit array address. The MC68000, with 32-bit registers, has no such loss of precision. For the Intel architecture, this meant that parameters located within 7/8ths of the address space would cause the routine to completely fail.

When Motorola contacted Intel about the flawed benchmark, Intel responded that bit matrixes located at such addresses were invalid to pass

to the subroutine. In other words, parameters which fail are simply not valid parameters. Such tunnel vision is a neat way to cover up the problem, but the simple fact still remains that the benchmark does not work.

Unfortunately, for users of the 8086/iAPX 286, in the real world we cannot build a software system and then loftily claim that anything which makes it fail is invalid.

And, by the way, you can guess which version Intel is now using to claim that the iAPX 286 speed compares with the MC68000. One wonders about benchmarks which don't work.

## PACKAGING

Since not all applications are able to take advantage of a single package type, it is important that components be available in packages that fit a wide range of enviromental constraints. The MC68000 is currently available in the following packages:

Dual-in-line ceramic 3.9 square inches — standard package ("L" Suffix)
Dual-in-line plastic 3.9 square inches — internal heat spreader ("G" Suffix)
Dual-in-line plastic 3.9 square inches — lower cost version ("P" Suffix)

JEDEC type B — LCC 1.0 inch square — socketable ("ZB" Suffix)

JEDEC type C — LCC 1.0 inch square — solderable ("ZC" Suffix)

Pin Grid Array Package — 1.0 inch square — both socketable and solderable ("R" Suffix)

**Power Considerations.** System designers, when choosing microprocessors, should include worst case power dissipation as an integral part of their performance rating procedure. High power dissipation requires effective and often costly thermal management to achieve system performance goals.

**"Worst Case" Calculations.** The first step in any determination of the impact of device power is a "worst case" calculation of the device juction temperature (Tj) using data sheet specifications. The specification for maximum power dissipation, along with maximum ambient temperature, and worst case thermal resistance (theta.ja), can be combined to derive the worst case junction temperature. The equation below can be used to calculate this temperature:

Tj = Ta + Pd * theta.ja

Additionally, some equation for derating power dissipated as a function of temperature such as:

Pd = K/(Tj + 273 ), where Tj is the temperature in degrees centigrade.

A "worst case" power dissipation analysis will be presented for the following devices:

MC68000ZB — Type B socketed chip carrier
MC68000G — 64 lead Plastic DIP
iAPX 286 — Type A socketed chip carrier

MC68000ZB Calculation:
max power = 1.5 W at Ta = 0°C
max temp = 70°C
theta.ja = 50°C/W

Pd = 1.28 W at Ta = 70°C

Tj = 134°C

MC68000G Calculation:
max power = 1.5 W at Ta = 0°C
max temp = 70°C
theta.ja = 30°C/W

Pd = 1.25 W at Ta = 70°C

Tj = 108°C

iAPX 286 Calculation:
max current = 600 ma at 25°C
max voltage = 5.5 v
max power = 3.3 W at Ta = 25°C
max temp = 70°C
theta.ja = 50°C/W

Pd = 3.1 W at Ta = 70°C

**Tj = 224°C**

Note the junction temperature for the iAPX 286.

**Conclusions.** Now that the "worst case" thermal analysis has been performed, how would a system designer use the results to determine the amount and cost of thermal management (if required to achieve a safe upper limit for Tj)? Traditionally, a safe upper limit for Tj has been subjective, and related heavily to reliability. A general guideline is that the junction temperature should be 125°C for the "worst case" thermal analysis. When the junction temperature is higher, the reliability of the device decreases correspondingly.

All devices discussed here, except the MC68000G,require some amount of thermal management, ranging from very little for the MC68000ZB to a great deal for the iAPX 286. The type of thermal management usually employed is the addition of heat sinks to the packages, and the assurance that there is a source of high-velocity air flow available to the heat sinks.

The junction-to-case thermal resistance characteristic (theta.jc) is useful in calculating the junction temperature, assuming that thermal management has effectively reduced the theta.ca to zero, and has made theta.ja equal to theta.jc. The JEDEC Type A socketed chip carrier used for the iAPX 286 has a theta.jc of 13°C/W. This results in a junction temperature of 108°C for the iAPX 286, which is the same as for the MC68000G with no thermal management. The same amount of thermal management for the MC68000ZB would achieve a theta.jc 17°C/W and a resulting junction temperature of 91°C.

## SUMMARY

The definition of a microprocessor's architecture involves many tradeoff decisions and impacts in very important ways the speed, efficiency and reliability of any computer system based on that architecture.

The Motorola M68000 Family of microprocessors implements a clean and powerful 32-bit architecture with general purpose registers and orthogonal addressing modes. A break with full compatibility to older 8-bit architectures was necessary so that advanced concepts could be introduced by the all-new 8/16/32-bit M68000 Family. The result was a new architecture which encompasses features required of today's system solutions and those for the rest of this decade and beyond, with complete user object-code compatibility.

In contrast to M68000 architecture, the iAPX 286 is weighted down with the instruction set of the much older 8086, with all of the cumbersome attributes associated with such ancient chip designs.

As a result, such crippling concepts as segmented addressing and special purpose registers rule the inflexible philosophy of iAPX architecture. These exact their harsh penalties in a multitude of ways, including excessive object code generation — which causes painful complexities in the integration of large, intricate software systems — and forcing at least seven times slower-than-M68000 execution in the accessing and addressing of large array and data structures.

The decision to implement direct 16-megabyte linear addressing was one of the greatest factors in the industry-wide acceptance of the M68000 Family. Contrast this to the mere 64K-byte segmented accesses of the iAPX 286. This forces extra manipulations on programmers, who must constantly attempt to contort their way around the severe limitations of restrictive registers and addressing modes.

The iAPX 286 does not even support virtual memory, but only a virtual segment restart capability. Contrast this with the virtual I/O, virtual machine and virtual window concepts available with Motorola's M68000 architecture.

The 32-bit architecture of M68000 Family microprocessors stands in stark contrast to the 16-bit segmented 8086/iAPX 286. Remarkable differences appear in performance, implementations, ease-of-use, code and execution efficiency between the two architectures when they are applied to actual programming environments.

In all important respects, the M68000 Family is seen to be far superior to the antiquated 8086/iAPX 286 architecture and fulfills the more sophisticated needs and requirements of today's demanding software systems and methodologies.

# APPENDIX A
## MC68000 Pascal is 45% Faster than iAPX 286 Pascal

Intel has recently claimed better performance for the iAPX 286 over the MC68000 in benchmark publications.

However, what they fail to tell customers is that the iAPX 286/8086 architecture is so inefficient that the Pascal benchmarks show a whopping 45% more bytes of code must be fetched to perform the exact same function.

The implications are obvious. Since the majority of bus cycles are fetching instructions (about 80% on the average), the iAPX 286/8086 must fetch 45% more program code to complete an equivalent function on the MC68000. Thus, for a given memory speed, the MC68000 executes these benchmarks significantly faster than the iAPX 286 possibly can because the iAPX 286 must fetch SIGNIFICANTLY MORE DATA (i.e., instructions.)

This analysis does not even begin to get into the overhead which will be required for the iAPX 286 native mode execution. The Pascal library code size was not included in the statistics gathered and therefore there is no such bias involved.

The much larger code size required is mandated by the inefficiencies of the Intel architecture, such as segment overhead, limited number of registers and forced dedicated register usage. These benchmarks will soon be rerun on the even faster MC68000 Pascal compiler and the results will be even more spectacular!

**Berkeley Benchmark Series Size in Bytes**

| Benchmark | MC68000 | iAPX 286/8086 | Ratio Larger |
|---|---|---|---|
| Search | 578 | 756 | 31% |
| Sieve | 120 | 348 | 290% |
| Puzzle | 1742 | 2301 | 32% |
| Acker | 124 | 311 | 250% |
| | | | 151% Average |

The results are quite clear. The MC68000's significantly reduced instruction needs allow it to spend more time completing a customer's program requests instead of wasting time decoding what to do.

# APPENDIX B
## Independent Benchmarks Show MC68000 Faster than iAPX 286

The following times for a wide variety of Motorola and Intel microprocessors are for an EDN benchmark study published in EDN magazine April 1, 1981, and September 16, 1981. The results show that the Motorola MC68000 without MMU beats the iAPX 286 in all cases and, with the MC68451 MMU, betters the iAPX 286 in the majority of cases. Note that Intel's new iAPX 186 turns out to be slower overall than the much older 8086 it is meant to replace, and that the new iAPX 286 is only 38% faster than the original 8086, even with significantly faster memories.

Also of interest is the fact that the MMU on board the iAPX 286 causes so much interrupt overhead that not only the old 8086 but the 8-bit MC68008 outperform it on the first benchmark.

**EDN BENCHMARKS**

| | (1) MC68000L12 | (2) MC68000+MMU | (3) MC68008 | (4) 8086-10 | (5) iAPX 186 | (6) iAPX 286 |
|---|---|---|---|---|---|---|
| A. | 25.6 μs | 39.2 μs | 57.6 μs | 43.2 μs | 50.0 μs | 96.8 μs |
| B. | 259.2 μs | 393.6 μs | 573.6 μs | 396.0 μs | 446.3 μs | 357.3 μs |
| E. | 127.0 μs | 177.4 μs | 372.6 μs | 201.0 μs | 249.8 μs | 128.4 μs |
| F. | 55.4 μs | 82.5 μs | 116.1 μs | 127.1 μs | 158.3 μs | 97.9 μs |
| H. | 116.8 μs | 180.0 μs | 281.6 μs | 269.0 μs | 259.2 μs | 199.8 μs |
| I. | 13.9 ms | 21.4 ms | 31.0 ms | 38.3 ms | 45.2 ms | 36.1 ms |
| K. | 289.1 μs | 418.7 μs | 555.6 μs | 938.5 μs | 724.7 μs | 508.8 μs |

(1) MC68000 at 12.5 megahertz, no wait states.
(2) MC68000 with MC68451 MMU at 10 megahertz, one wait state.
(3) MC68008 8-bit microprocessor at 10 megahertz, no wait states.
(4) 8086-10 at 10 megahertz, no wait states.
(5) iAPX 186 at 8 megahertz, no wait states.
(6) iAPX 286 protected mode (MMU turned on) at 8 megahertz, no wait states.

A. I/O Interrupt: four interrupts, increment and return
B. I/O Interrupt; queue interrupts
E. String search
F. Bit test/set/reset
H. Linked list benchmark
I. Quick sort
K. Bit matrix inversion

NOTE 1 All Motorola speeds available now as regular production. Intel 80186 (iAPX 186) and 80286 (iAPX 286) 8 megahertz parts will be available when regular production commences, according to the manufacturer.

NOTE 2 All MC68000 code the same. Some 8086 and 80186 code changed by Intel to allow iAPX 286 protected mode execution.

# APPENDIX C
## iAPX 286 Substring Benchmark

The following code shows that there are not enough registers on board the iAPX 286 microprocessor to accomplish even a simple substring character search. Lines 93, 95, 102, and 103 save and restore the overworked SI and DI registers.

```
                        53      ; THE BENCHMARK PROCEDURE
                        54
                        55
                        56      ;               EDN BENCHMARK FOR IAPX 86
                        57      ;               CHARACTER SEARCH
                        58      ;
                        59      ;
                        60      ;       ASSUMPTIONS:  1 VALID DATA FOR SRCHLNGTH
                        61      ;                       AND MINLNGTH (SRCHLNGTH>MINLNGTH)
                        62      ;                     2 PARAMETERS ARE PASSED IN REGISTERS AS FOLLOWS:
                        63      ;                                 ES:DI   TABLE_PTR
                        64      ;                                 DS:SI   STR_PTR
                        65      ;                                 CX      TABLE_LNGTH
                        66      ;                                 DX      STR_LNGTH
                        67      ;                     3 THE LOCATION OF THE STRING IS RETURNED IN DI
                        68      ;                     4 WORKING REGISTERS (AX & DX) ARE PUSHED AND POPPED
                        69
                        70      ;
                        71      ;
                        72
                        73      ;SAVE WORK REGISTERS
0022 50                 74              PUSH    AX
0023 52                 75              PUSH    DX
0024 51                 76              PUSH    CX                      ;SAVE LENGTH
0025 FC                 77              CLD
0026 8A04               78              MOV     AL,[SI]                 ;LOAD FIRST CHAR
0028 2BCB               79              SUB     CX,BX                   ;LENGTH DIFFERENCE
002A 41                 80              INC     CX                      ;POSSIBLE SEARCH COUNT
                        81      ;FIND MATCH TO FIRST CHARACTER
                        82
002B                    83      TRYNXT:
002B F2                 84        REPNE SCASB                           ;SCAN WHILE NOT EQUAL
002C AE
002D 7406               85              JE      MATCH1                  ;JUMP ON MATCH
002F BFFFFF             86      NOTFND: MOV     DI,-1                   ;SET NOT FOUND FLAG
0032 EB1790             87              JMP     DONE                    ;EXIT
                        88
                        89      ;FIRST CHARACTER MATCH HAS BEEN FOUND
                        90      ;COMPARE THE REST OF THE STRING
```

```
                                91
0035 8BD1                       92      MATCH1: MOV     DX,CX           ;SAVE POSSIBLE SEARCH COUNT
0037 57                         93              PUSH    DI              ;SAVE PLACE IN TABLE
0038 8BCB                       94              MOV     CX,BX           ;LOAD STR_LNGTH
003A 56                         95              PUSH    SI              ;SAVE STR_PTR
003B 4F                         96              DEC     DI              ;RETEST FIRST (IN CASE LEN=1)
003C F3                         97         REPE CMPSB                   ;COMPARE STRING
003D A6
                                98              ;
                                99      ;DROP THROUGH IF THE STRING DOES NOT MATCH OR WHEN
                               100      ; CX=0 (THE ENTIRE STRING MATCHES)
                               101
003E 5E                        102              POP     SI              ;RESTORE STR_PTR
003F 5F                        103              POP     DI              ;RESTORE PLACE IN TABLE
0040 8BCA                      104              MOV     CX,DX           ;RESTORE POSSIBLE SEARCH COUNT
0042 7404                      105              JE      FOUND           ;STRING MATCHES
0044 E3E9                      106              JCXZ    NOTFND          ;END OF STRING, NO MATCH
0046 EBE3                      107              JMP     TRYNXT          ;TRY NEXT BYTE IN TABLE
                               108
0048 5F                        109      FOUND:  POP     DI              ;GET TABLE_LENGTH
0049 2BFA                      110              SUB     DI,DX           ;COMPUTE PTR TO STR IN TABLE
                               111
                               112      ;RESTORE REGISTERS
                               113
004B 5A                        114      DONE:   POP     DX
004C 58                        115              POP     AX
                               116
004D C3                        117              RET                     ;RETURN TO CALLING ROUTINE
                               118      BNCHPROG ENDP
                               119
----                           120      CODE    ENDS
                               121
                               122              END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

# APPENDIX D
## Motorola MC68000 Quicksort

```
                                OPT      BRS
                        *
                        *               MC68000  EDN BENCHMARK I
                        *
                        *                     QUICKSORT
                        *
                        *  ATTRIBUTES:  * 16 MEGABYTES ADDRESS RANGE
                        *               * POSITION INDEPENDENT
                        *               * REENTRANT
                        *
                        *  INPUT:  D0 - "N" RECORD COUNT
                        *          D1 - "M" THRESHOLD FOR INSERTION SORT
                        *          A0 - "REC" ADDRESS OF THE SORT ARRAY
                        *
                        *  OUTPUT: THE SORT DATA ARRAY IS SORTED
                        *
                        *     ALL REGISTERS ARE TRANSPARENT OVER THIS ROUTINE
                        *
                        *                     LINES: 87
                        *                     BYTES: 266
                        *

                        * MISCELLANEOUS EQUATES
            00000010    ENTRYLEN EQU      16                  SORT ENTRY RECORD LENGTH
            00000003    KEY      EQU      3                   OFFSET TO KEY WITHIN RECORD
            00000007    KEYLEN   EQU      7                   SORT KEY LENGTH


                        *                    QUICKSORT SUBROUTINE
0 00000000 48E7FFFE     QUICK    MOVEM.L  D0-D7/A0-A6,-(SP)   SAVE ALL REGISTERS
0 00000004 2400                  MOVE.L   D0,D2               COPY NUMBER OF RECORDS OVER
0 00000006 E988                  LSL.L    #4,D0               CALCULATE POINTER TO LAST RECORD
0 00000008 43F008F0              LEA      -ENTRYLEN(A0,D0.L),A1 A1 <- POINTER TO LAST RECORD = R
0 0000000C E989                  LSL.L    #4,D1               FIND TOTAL SIZE OF M RECORDS
0 0000000E 2C41                  MOVE.L   D1,A6               KEEP VALUE IN A6 FOR LATER
0 00000010 48E7A040              MOVEM.L  D0/D2/A1,-(SP)      SAVE DUMMY, COUNT, AND TOP ON STACK
0 00000014 42A7                  CLR.L    -(SP)               MARK SORT STACK EMPTY


                        *                         QUICKSORT PHASE
                        * REGISTER USE:  A0 - FIRST RECORD OF SUBFILE    A1 - LAST RECORD OF SUBFILE
                        *                A2/A3 - KEY POINTERS            A4/A5 - WORK POINTERS
                        *                A6 - LENGTH OF "M" RECORDS      SP - RECURSIVE CALL ARGUMENTS

0 00000016 45E80003     SORT     LEA      KEY(A0),A2          A2 -> KEY(I) = REC(L)
0 0000001A 47E90013              LEA      ENTRYLEN+KEY(A1),A3 A3 -> KEY(J) = REC(R+1)
0 0000001E 49E80003     LOOP1    LEA      KEY(A0),A4          A4 -> V FOR CURRENT RECORD
0 00000022 45EA0010              LEA      ENTRYLEN(A2),A2     I <- I+1
0 00000026 2A4A                  MOVE.L   A2,A5               A5 TEMP FOR I
0 00000028 7006                  MOVE.L   #KEYLEN-1,D0        D0=LOOP COUNTER
0 0000002A B90D         CMP1     CMP.B    (A5)+,(A4)+         COMPARE V-REC(I)
0 0000002C 56C8FFFC              DBNE     D0,CMP1             LOOP WHILE EQUAL
0 00000030 62EC                  BHI      LOOP1               IF REC(I)<V CONTINUE COMPARING
```

```
 59 0 00000032 49E80003     LOOP2    LEA      KEY(A0),A4           A4 -> KEY(V) OF CURRENT RECORD
 60 0 00000036 47EBFFF0              LEA      -ENTRYLEN(A3),A3     J <- J-1
 61 0 0000003A 2A4B                  MOVE.L   A3,A5                A5 = TEMP FOR J
 62 0 0000003C 7006                  MOVE.L   #KEYLEN-1,D0         LOOP COUNTER
 63 0 0000003E BB0C         CMP2     CMP.B    (A4)+,(A5)+         COMPARE REC(J)-V
 64 0 00000040 56C8FFFC              DBNE     D0,CMP2              LOOP WHILE EQUAL
 65 0 00000044 62EC                  BHI      LOOP2                IF REC(J)>V CONTINUE COMPARING
 66 0 00000046 B5CB                  CMP.L    A3,A2                I >= J
 67 0 00000048 6436                  BCC      END1ST               BRANCH IF I >= J
 68 0 0000004A 4CEA000FFFFD          MOVEM.L  -KEY(A2),D0-D3       SWAP
 69 0 00000050 4CEB00F0FFFD          MOVEM.L  -KEY(A3),D4-D7       .     REC(J)
 70 0 00000056 48EB000FFFFD          MOVEM.L  D0-D3,-KEY(A3)       .           WITH
 71 0 0000005C 48EA00F0FFFD          MOVEM.L  D4-D7,-KEY(A2)       .                REC(I)
 72 0 00000062 60BA                  BRA      LOOP1               CONTINUE
 73
 74                         * DECIDE SUBFILE DIRECTION
 75 0 00000064 B28E         NEWLR0   CMP.L    A6,D1                (R-J) <= MSIZE? (R-J) SUBFILE SMALLER?
 76 0 00000066 6F08                  BLE      NEWR                 BRANCH IF SO
 77 0 00000068 B481                  CMP.L    D1,D2                DETERMINE SMALLER SUBFILE
 78 0 0000006A 650A                  BCS      STACK1               BRANCH IF (J-L) IS SMALLER
 79 0 0000006C 2F09                  MOVE.L   A1,-(SP)             STACK R
 80 0 0000006E 2F0B                  MOVE.L   A3,-(SP)             STACK J
 81                         *                                     (R-J) SUBFILE SMALLER, SET L & R TO
 82                         *                                    LARGER SUBFILE LIMITS
 83 0 00000070 43EBFFF0     NEWR     LEA      -ENTRYLEN(A3),A1     R <- J-1, L STAYS THE SAME
 84 0 00000074 60A0                  BRA      SORT                 CONTINUE SORT
 85                         *                                     (J-L) SUBFILE SMALLER, SET L & R TO
 86                         *                                    LARGER SUBFILE LIMITS
 87 0 00000076 48E70090     STACK1   MOVEM.L  A0/A3,-(SP)          PUSH L & J ONTO SORT STACK
 88 0 0000007A 41EB0010     NEWLR1   LEA      ENTRYLEN(A3),A0      L <- J+1, R STAYS THE SAME
 89 0 0000007E 6096                  BRA      SORT                 CONTINUE SORT
 90
 91
 92                         * NEW SUBFILE FOUND, NOW DETERMINE NEXT STAGE
 93 0 00000080 578B         END1ST   SUB.L    #KEY,A3              -KEY TO GET TO BEGINNING OF RECORD
 94 0 00000082 4CD0000F              MOVEM.L  (A0),D0-D3           SWAP
 95 0 00000086 4CD300F0              MOVEM.L  (A3),D4-D7           .     REC(L)
 96 0 0000008A 48D3000F              MOVEM.L  D0-D3,(A3)           .           AND
 97 0 0000008E 48D000F0              MOVEM.L  D4-D7,(A0)           .                REC(J)
 98 0 00000092 2209                  MOVE.L   A1,D1                D1 <- R
 99 0 00000094 240B                  MOVE.L   A3,D2                D2 <- J
100 0 00000096 9282                  SUB.L    D2,D1                D1 <- R-J
101 0 00000098 9488                  SUB.L    A0,D2                D2 <- J-L
102 0 0000009A B48E                  CMP.L    A6,D2                COMPARE (J-L) <= MSIZE
103 0 0000009C 62C6                  BHI      NEWLR0               BRANCH IF NO
104 0 0000009E B28E                  CMP.L    A6,D1                COMPARE (R-J) <= MSIZE
105 0 000000A0 62D8                  BHI      NEWLR1               BRANCH IF NO
106 0 000000A2 4CDF0300              MOVEM.L  (SP)+,A0/A1          POP NEXT L AND R FROM STACK
107 0 000000A6 2008                  MOVE.L   A0,D0                TEST IF STACK IS EMPTY
108 0 000000A8 6600FF6C              BNE      SORT                 CONTINUE SORT IF NOT EMPTY
109
110                         * FALL INTO INSERTION SORT AS ALL SUBFILES BELOW OR EQUAL M RECORDS
```

```
112                          *                    INSERTION SORT PHASE
113                          *
114                          *  REGISTER USE: D0 - LOOP CONTROL              A0 - REC(I)
115                          *                D1 - COUNTER AND SWAP REGISTER A1 - REC(J)
116                          *                D2/D4 - SWAP REGISTERS         A2/A3 - WORK REGISTERS
117                          *                D5/D7 - "V" SAVE REGISTERS     A4 - REC(J-1)
118                          *                                               A5 - "V" SAVE REGISTER
119                          *                                               A6 - FRAME POINTER
120                          *
121                          *       NOTE:  STACK SPACE IS RESERVED FOR "V" KEY COMPARE RECORD COPIES
122                          *
123
124 0 000000AC 4CDF0101              MOVEM.L  (SP)+,D0/A0          RELOAD RECORD COUNT AND TOP RECORD
125 0 000000B0 4E56FFF0              LINK     A6,#-ENTRYLEN        ALLOCATE "V" KEY COPY AREA ON STACK
126 0 000000B4 5540                  SUB      #2,D0                D0 RANGES FROM N-2 THROUGH 0
127
128 0 000000B6 41E8FFF0      LOOPOUT LEA      -ENTRYLEN(A0),A0     I <- I-1
129 0 000000BA 45E80003              LEA      KEY(A0),A2           A2 -> KEY(I)
130 0 000000BE 47E80013              LEA      ENTRYLEN+KEY(A0),A3  A3 -> KEY(I+1)
131 0 000000C2 7206                  MOVE.L   #KEYLEN-1,D1         LOOP COUNTER FOR COMPARE
132 0 000000C4 B50B          CMPII1  CMP.B    (A3)+,(A2)+          COMPARE KEY(I)-KEY(I+1)
133 0 000000C6 56C9FFFC              DBNE     D1,CMPII1            LOOP WHILE EQUAL
134 0 000000CA 6332                  BLS      ENDIF                BRANCH IF KEY(I) <= KEY(I+1)
135
136 0 000000CC 4CD020E0              MOVEM.L  (A0),D5-D7/A5        V  <-  REC(I)
137 0 000000D0 48D700E0              MOVEM.L  D5-D7,(SP)           AND ON STACK FOR KEY COMPARE
138 0 000000D4 43E80010              LEA      ENTRYLEN(A0),A1      A1 -> REC(J) = REC(I+1)
139 0 000000D8 2848                  MOVE.L   A0,A4                PRIME A4 -> REC(J-1)
140
141 0 000000DA 4CD1001E      LOOPIN  MOVEM.L  (A1),D1-D4           TEMP <- REC(J)
142 0 000000DE 48D4001E              MOVEM.L  D1-D4,(A4)           REC(J-1) <- TEMP
143 0 000000E2 2849                  MOVE.L   A1,A4                A4 -> NEXT REC(J-1)
144 0 000000E4 43E90010              LEA      ENTRYLEN(A1),A1      J = J+1
145 0 000000E8 45EF0003              LEA      KEY(SP),A2           A2 -> KEY(V)
146 0 000000EC 47E90003              LEA      KEY(A1),A3           A3 -> KEY(J)
147 0 000000F0 7206                  MOVE.L   #KEYLEN-1,D1         LOOP COUNTER IN D1
148 0 000000F2 B50B          CMPVJ   CMP.B    (A3)+,(A2)+          COMPARE KEY(V)-KEY(J)
149 0 000000F4 56C9FFFC              DBNE     D1,CMPVJ             LOOP WHILE EQUAL
150 0 000000F8 62E0                  BHI      LOOPIN               IF KEY(V) > KEY(J) CONTINUE LOOP
151
152 0 000000FA 48D420E0              MOVEM.L  D5-D7/A5,(A4)        REC(J-1) <- V
153
154 0 000000FE 51C8FFB6      ENDIF   DBRA     D0,LOOPOUT           CONTINUE LINEAR INSERT
155
156 0 00000102 4E5E                  UNLK     A6                   FREE AND RESTORE STACK
157 0 00000104 4CDF7FFF              MOVEM.L  (SP)+,D0-D7/A0-A6    RESTORE REGISTERS
158 0 00000108 4E75                  RTS                           RETURN TO CALLER
159
160
161                                  END

****** TOTAL ERRORS    0--    0
```

# APPENDIX E
## Intel 8086 Quicksort

```
ISIS-II 8086/8087/8088 MACRO ASSEMBLER V3.0 ASSEMBLY OF MODULE TEST_BENCHMARK_I
OBJECT MODULE PLACED IN :F1:MOTBI.OBJ
ASSEMBLER INVOKED BY:  asm86 :f1:motbi.a86 date(8/7/81) workfiles(:f2:,:f3:)


LOC  OBJ                  LINE     SOURCE

                             1                    name    test_benchmark_I
                             2     ;
                             3     ;              For the EDN algorithm and data:
                             4     ;
                             5     ;
                             6     ;
----                         7     code            segment public  'code'
                             8                     assume  cs:code
                             9     ;
                            10     ;         Define local values for records
                            11     ;
  0007                      12     key_length      equ     7
  0003                      13     key_offset      equ     3
  0010                      14     record_size     equ     16
                            15     ;
                            16     ;          Quicksort function
                            17     ;
                            18     ;          Four parameters are passed in the registers:
                            19     ;
                            20     ;        n       number of elements 0 < N <= 10000       cx
                            21     ;        rec     pointer to start of record series       ds
                            22     ;        m       transfer point to insertion sort        bx
                            23     ;
                            24     ;        All registers are transparent except for the flags.
                            25     ;
                            26     ;          Define abbreviations commonly used.
                            27     ;
  0008                      28     word_rec_size   equ     8
  00FF                      29     largest_key     equ     0ffh
                            30     ;
                            31     ;          The following local variables are initialized and restored
                            32     ;        with pushes and pops
                            33     ;
  000A[]                    34     rec_ptr         equ     word ptr [bp+10]        ; Save area for rec_ptr
  000C[]                    35     n               equ     rec_ptr+2               ; Save area for n
  000E[]                    36     m               equ     n+2                     ; Save area for m
 -0010[]                    37     temp_area       equ     byte ptr [bp-record_size]
                            38     ;
                            39     ;          Start of sort function
                            40     ;
0000                        41     quick_sort      proc    far
                            42
0000 55                     43             push    bp                      ; Establish stack addressibility
0001 53                     44             push    bx                      ; Set m in local area
0002 51                     45             push    cx                      ; Set n in local area
0003 1E                     46             push    ds                      ; Set rec_ptr in local area
                            47
0004 50                     48             push    ax                      ; Save working registers
0005 52                     49             push    dx
0006 56                     50             push    si
```

```
0007 57                 51              push    di
0008 06                 52              push    es
0009 8BEC               53              mov     bp,sp                   ; Identify base of save area
                        54
000B 8CDB               55              mov     bx,ds                   ; Form .rec(0)
000D 43                 56              inc     bx                      ; bx = .rec(1)
000E 8BD3               57              mov     dx,bx                   ; dx,1 = .rec(1)
0010 03D9               58              add     bx,cx                   ; bx = .rec(N+1)
0012 4B                 59              dec     bx                      ; r = .rec(N)
0013 33C0               60              xor     ax,ax                   ; Indicate empty r,l save stack
0015 50                 61              push    ax                      ; Will be popped before insertion sort
                        62      ;
                        63      ;         Outer loop of quicksort before the insertion sort.
                        64      ;
                        65      ;         Register usage:
                        66      ;
                        67      ;       l       dx      Note:   r >= l + m
                        68      ;       r       bx              rec(l-1) <= rec(i) <= rec(r+1) for l <= i <= r
                        69      ;
0016                    70      outer_loop:
                        71
0016 55                 72              push    bp                      ; Save stack base
0017 53                 73              push    bx                      ; Save r for later
0018 43                 74              inc     bx                      ; Form j = r+1 in bx
0019 8BEA               75              mov     bp,dx                   ; Set i = l
001B                    76      restart_i_loop:
001B 8B0600             77              mov     ax,key_length-1
001E 8EC2               78              mov     es,dx                   ; es = .rec(l)
                        79      ;
                        80      ;         Skip over left part of subfile.
                        81      ;
                        82      ;       Register usage:
                        83      ;               es = ss
                        84      ;               bp      .rec(i)
                        85      ;               bx      .rec(j)
                        86      ;               ax      length-1 of key
                        87      ;               dx      .rec(l)
                        88      ;               ij      is pushed on stack holding either i or j
                        89      ;
0020                    90      i_loop:
0020 45                 91              inc     bp                      ; i = i+1
0021 8EDD               92              mov     ds,bp                   ; ds = .rec(i)
0023 8E0300             93              mov     si,key_offset
0026 8BFE               94              mov     di,si                   ; Set offset for v
0028 A6                 95              cmpsb                           ; Look at first byte of key
0029 72F5               96              jb      i_loop
002B 7706               97              ja      j_loop
                        98
002D 8BC8               99              mov     cx,ax                   ; Else examine the rest of the key
002F F3                100      repe   cmpsb                           ; Compare rec(i) - rec(l)
0030 A6
0031 72ED              101              jb      i_loop                  ; Continue while rec(i) < rec(l)
                       102      ;
                       103      ;       Assert: i <= j          es = .rec(l)
                       104      ;
```

```
                        ;         Skip over right part of subfile.
                        ;       Registers same as before except ax,ds = .rec(j).
                        ;
0033                    j_loop:
0033 4B                         dec     bx                      ; j = j-1
0034 8EDB                       mov     ds,bx                   ; ds = .rec(j)
0036 BE0300                     mov     si,key_offset
0039 8BFE                       mov     di,si                   ; Set offset for v
003B A6                         cmpsb                           ; Look at the first byte of the key
003C 77F5                       ja      j_loop
003E 7206                       jb      compare_ij
                        ;
0040 8BC8                       mov     cx,ax                   ; Else look at the rest of the key
0042 F3                   repe  cmpsb                           ; Compare rec(j) - rec(1)
0043 A6
0044 77ED                       ja      j_loop                  ; Continue while rec(j) > rec(1)
                        ;
                        ;       Assert: j >= i-1        es = .rec(1)  ds=bx
                        ;
                        ;         Compare j and i for swap test.
                        ;
0046                    compare_ij:
0046 33F6                       xor     si,si                   ; Prepare for exchange of records
0048 8BFE                       mov     di,si
004A B90800                     mov     cx,word_rec_size        ; Get word size of a record
004D 3BEB                       cmp     bp,bx
004F 730E                       jae     swap_loop2              ; Jump if i >= j
                        ;
0051 8EC5                       mov     es,bp                   ; es = .rec(i)  ds = .rec(j)
                        ;
                        ;         Exchange rec(i) with rec(j).
                        ;
0053                    swap_loop1:
0053 268B05                     mov     ax,es:[di]
0056 8704                       xchg    [si],ax
0058 46                         inc     si                      ; Bump source pointer
0059 46                         inc     si
005A AB                         stosw                           ; Auto bump of destination pointer
005B E2F6                       loop    swap_loop1
                        ;
005D EBBC                       jmp     restart_i_loop          ; Continue preorder loop
                        ;
                        ;         Save subfile definitions on the stack area.
                        ;       But first exchange rec(1) with rec(j).
                        ;
005F                    swap_loop2:
005F 268B05                     mov     ax,es:[di]
0062 8704                       xchg    [si],ax
0064 46                         inc     si                      ; Bump source pointer
0065 46                         inc     si
0066 AB                         stosw                           ; Auto bump of destination pointer
0067 E2F6                       loop    swap_loop2
                        ;
                        ;       Assert: ds,bx = .rec(j)
                        ;               dx,es = .rec(1)
```

```
                        159     ;
                        160     ;       Compare the subfile sizes of j-1 and r-j to determine the
                        161     ;     the next subfile to sort.  Note: Subfiles do not include j!
                        162     ;
0069 8BC3               163             mov     ax,bx                   ; Save j
006B 5B                 164             pop     bx                      ; Get bx=r
006C 8BF3               165             mov     si,bx                   ; si,bx = r
006E 2BF0               166             sub     si,ax                   ; si = r-j
0070 8BF8               167             mov     di,ax                   ; di = j
0072 2BFA               168             sub     di,dx                   ; di = j-1
0074 5D                 169             pop     bp                      ; Restore stack addressing
0075 8B4E0E             170             mov     cx,m                    ; cx = m
                        171     ;
                        172     ;       Register usage:
                        173     ;             dx,es   .rec(l)
                        174     ;             bx      .rec(r)
                        175     ;             ax      .rec(j)
                        176     ;             cx      m
                        177     ;             si      subfile size of r-j
                        178     ;             di      subfile size of j-1
                        179     ;
0078 3BF9               180             cmp     di,cx
007A 770C               181             ja      jl_bt_m                 ; Jump if j-1 > m
                        182
007C 3BF1               183             cmp     si,cx
007E 771A               184             ja      set_new_l               ; Jump if r-j > m
                        185     ;
                        186     ;       Both subfile sizes <= m.
                        187     ;     Get next r,l pair from the stack if not empty
                        188     ;
0080 5B                 189             pop     bx                      ; See if the stack is empty
0081 0BDB               190             or      bx,bx                   ; Test r value
0083 741B               191             jz      end_outer               ; Do insertion sort if empty
                        192     ;
                        193     ;       Set l and r from the stack.
                        194     ;     Stack ordering is r,l,r,l,....
                        195     ;
0085 5A                 196             pop     dx                      ; Get l from stack
0086 EB8E               197             jmp     outer_loop
                        198     ;
                        199     ;       j-1 > m. See if r-j > m
                        200     ;
0088                    201     jl_bt_m:
0088 3BF1               202             cmp     si,cx                   ; Compare r-j with m
008A 7606               203             jbe     set_new_r               ; Jump if r-j <= m
                        204     ;
                        205     ;       See which subfile is largest, both are larger than m.
                        206     ;
008C 3BFE               207             cmp     di,si                   ; Compare j-1 with r-j
008E 7208               208             jb      stack_lj                ; Jump if j-1 < r-j
                        209     ;
                        210     ;       Push lower and upper limits of j-1, (the larger subfile).
                        211     ;     Stack ordering is: r,l,r,l,.....
                        212     ;
0090 52                 213             push    dx                      ; Stack l = l
```

```
0091 50                 214             push    ax                      ; Stack r = j
0092                    215     set_new_r:
0092 8BD8               216             mov     bx,ax                   ; Set new r = j-1
0094 4B                 217             dec     bx                      ; l remains same
0095 E97EFF             218             jmp     outer_loop
                        219     ;
                        220     ;         Push lower and upper limits of r-j, (the larger subfile).
                        221     ;
0098                    222     stack_lj:
0098 52                 223             push    dx                      ; Stack l = l
0099 50                 224             push    ax                      ; Stack r = j
009A                    225     set_new_l:
009A 8BD0               226             mov     dx,ax                   ; Set l as j+1
009C 42                 227             inc     dx                      ; r remains same
009D E976FF             228             jmp     outer_loop
                        229     ;
                        230     ;         Perform an insertion sort on the entire record array.
                        231     ;
00A0                    232     end_outer:
00A0 83EC10             233             sub     sp,record_size          ; Allocate space for temp
00A3 8B5E0C             234             mov     bx,n                    ; Get original array length
00A6 035E0A             235             add     bx,rec_ptr              ; Form i = .rec(n)
00A9                    236     enter_last_loop:
00A9 B80600             237             mov     ax,key_length-1         ; Set key length
00AC BA0300             238             mov     dx,key_offset           ; Set key offset
                        239     ;
                        240     ;         Note: bx = i+1
                        241     ;
00AF                    242     last_loop:
00AF 4B                 243             dec     bx                      ; Form i = i - 1
00B0 3B5E0A             244             cmp     bx,rec_ptr              ; Test for .rec(0) implying i = 0
00B3 7653               245             jbe     done
                        246
00B5 8EDB               247             mov     ds,bx                   ; ds = .rec(i)
00B7 43                 248             inc     bx
00B8 8EC3               249             mov     es,bx                   ; es = .rec(i+1)
00BA 4B                 250             dec     bx                      ; Restore i
00BB 8BF2               251             mov     si,dx                   ; Set source and destination offset
00BD 8BFA               252             mov     di,dx
00BF A6                 253             cmpsb                           ; Compare 1st bytes of keys
00C0 72ED               254             jb      last_loop
00C2 7706               255             ja      move_down
                        256
00C4 8BC8               257             mov     cx,ax                   ; Examine rest of key
00C6 F3                 258       repe  cmpsb                           ; Compare rec(i) -  rec(i+1)
00C7 A6
00C8 72E5               259             jb      last_loop               ; Loop while rec(i) < rec(i+1)
                        260     ;
                        261     ;         Move the entire subfile down one element until a record >= rec(i)
                        262     ;       is found.
                        263     ;
00CA                    264     move_down:
00CA B80800             265             mov     ax,word_rec_size        ; Get word count of a record
00CD 53                 266             push    bx                      ; Save i
00CE 8CD2               267             mov     dx,ss                   ; Save rec(i)
```

```
00D0 8EC2                268             mov     es,dx
00D2 8D7EF0              269             lea     di,temp_area            ; Set address of temp area
00D5 33F6                270             xor     si,si
00D7 8BC8                271             mov     cx,ax                   ; Set move count
00D9 F3                  272       rep   movsw                           ; v = rec(i)
00DA A5
00DB 8EC3                273             mov     es,bx                   ; j-1 becomes i in bx
                         274     ;
                         275     ;         Use es = ds = .rec(j-1).
                         276     ;       Assert: dx=ss
                         277     ;
00DD                     278     move_down_loop:
                         279
00DD BE1000              280             mov     si,record_size          ; si is offset for rec(j)
00E0 33FF                281             xor     di,di                   ; di is offset for rec(j-1)
00E2 8BC8                282             mov     cx,ax                   ; Set move length
00E4 F3                  283       rep   movsw                           ; rec(j-1) = rec(j)
00E5 A5
                         284     ;
                         285     ;         Note: si is offset for rec(j+1) now.
                         286     ;
00E6 43                  287             inc     bx                      ; j = j + 1
00E7 8EC2                288             mov     es,dx                   ; es:di = .v
00E9 8D7EF3              289             lea     di,temp_area+key_offset ; Set address of temp area
00EC 83C603              290             add     si,key_offset           ; si has offset for rec(j) key
00EF B90700              291             mov     cx,key_length           ; Set size of compare string
00F2 F3                  292       repe  cmpsb                           ; Compare rec(j) with v
00F3 A6
00F4 8EC3                293             mov     es,bx                   ; Set new base address
00F6 8ED8                294             mov     ds,bx
00F8 72E3                295             jb      move_down_loop          ; Continue while rec(j) < v
                         296     ;
                         297     ;         Copy v to rec(old j).
                         298     ;
00FA 33FF                299             xor     di,di                   ; Set offset to rec(j-1)
00FC 8EDA                300             mov     ds,dx                   ; Get address of v in ds:si
00FE 8D76F0              301             lea     si,temp_area            ; Set address of temporary area
0101 8BC8                302             mov     cx,ax                   ; Set move length
0103 F3                  303       rep   movsw                           ; rec(j-1) = v
0104 A5
0105 5B                  304             pop     bx                      ; Restore i
0106 EBA1                305             jmp     enter_last_loop
                         306     ;
                         307     ;         All done, restore registers.
                         308     ;
0108                     309     done:
0108 8BE5                310             mov     sp,bp                   ; Clear stack area
010A 07                  311             pop     es
010B 5F                  312             pop     di
010C 5E                  313             pop     si
010D 5A                  314             pop     dx
010E 58                  315             pop     ax
                         316
010F 1F                  317             pop     ds                      ; Restore rec_ptr
0110 59                  318             pop     cx                      ; Restore n
```

```
0111 5B                 319             pop     bx                      ; Restore m
0112 5D                 320             pop     bp
0113 CB                 321             ret
                        322
                        323     quick_sort      endp
                        324 +1  $eject
```

# NOTES

# MOTOROLA SEMICONDUCTOR SALES OFFICES

### MOTOROLA SEMICONDUCTOR AMERICAS DISTRICT OFFICES

**ALABAMA**, Huntsville ..... (205)830-1050
**ARIZONA**, Phoenix ..... (602)244-7100
**ARIZONA**, Phoenix (General Motors Group) ..... (602)244-7125
**CALIFORNIA**, Encino/ Sherman Oaks ..... (213)986-6850 (213)872-1505
**CALIFORNIA**, Orange (Orange Exch.) ..... (714)634-2844 (L.A. Exch.) ..... (213)445-5585
**CALIFORNIA**, San Diego ..... (714)560-4644
**CALIFORNIA**, San Jose ..... (408)985-0510
**COLORADO**, Colorado Springs .. (303)599-7404
**COLORADO**, Denver ..... (303)773-6800
**CONNECTICUT**, New Haven/ Hamden ..... (203)281-0771
**FLORIDA**, Pompano Beach/ Ft. Lauderdale ..... (305)491-8141
**FLORIDA**, Casselberry/Maitland (305)831-3422
**FLORIDA**, St. Petersburg ..... (813)576-6030
**GEORGIA**, Atlanta ..... (404) 256-0222
**ILLINOIS**, Chicago/Schaumburg (312)576-7800
**INDIANA**, Fort Wayne ..... (219)484-0436
**INDIANA**, Indianapolis ..... (317) 849-7060
**INDIANA**, Kokomo ..... (317)457-6634
**IOWA**, Cedar Rapids ..... (319)373-1328
**KANSAS**, Kansas City/Mission ... (913)384-3050
**MASSACHUSETTS**, Berlin ..... (617)562-3856
**MASSACHUSETTS**, Burlington .. (617)273-5020
**MICHIGAN**, Detroit/Westland .... (313)261-6200
**MINNESOTA**, Minneapolis ..... (612)545-0251
**MISSOURI**, St. Louis ..... (314)872-7681
**NEW JERSEY**, River Edge ..... (201)488-1200
**NEW YORK**, Poughkeepsie/ Fishkill ..... (914)473-8102
**NEW YORK**, Long Island/ Hauppauge ..... (516)231-9000
**NEW YORK**, Pittsford ..... (716)248-5494
**NEW YORK**, Syracuse ..... (315)455-7373
**NORTH CAROLINA**, Raleigh ..... (919)876-6025
**OHIO**, Cleveland ..... (216)461-3160
**OHIO**, Dayton ..... (513)294-2231
**OHIO**, Columbus/ Worthington ..... (614)846-9460
**OKLAHOMA**, Tulsa ..... (918)664-5227
**OREGON**, Portland ..... (503)641-3681
**PENNSYLVANIA**, Philadelphia/ Horsham ..... (215)443-9400
**TENNESSEE**, Knoxville ..... (615)690-5592
**TEXAS**, Austin ..... (512)452-7673
**TEXAS**, Dallas ..... (214)931-9222
**TEXAS**, Houston ..... (713)783-6400
**UTAH**, Salt Lake City ..... (801)539-1190
**VIRGINIA**, Charlottesville ..... (804)977-3691
**WASHINGTON**, Bellevue ..... (206)454-4160 Seattle Access ..... (206)622-9960
**WASHINGTON, DC/MARYLAND**, Hyattsville ..... (301)577-2600
**WISCONSIN**, Milwaukee/ Wauwatosa ..... (414)476-5554

Field Applications Engineering Available Through All Sales Offices

### MOTOROLA SEMICONDUCTOR—CANADA

**MANITOBA**, Winnipeg ..... (204)889-0693
**ONTARIO**, Downsview/Toronto .. (416)661-6400
**ONTARIO**, Ottawa ..... (613)235-4388
**QUEBEC**, Montreal ..... (514)731-6881

### MOTOROLA SEMICONDUCTOR INTRA-COMPANY OFFICES

**ARIZONA**, Scottsdale ..... (602)949-3811
**FLORIDA**, Ft. Lauderdale ..... (305)475-6120
**ILLINOIS**, Franklin Park/ Schaumburg ..... (312)576-2788
**ILLINOIS**, Schaumburg ..... (312)576-5518
**ILLINOIS**, Schaumburg/ Automotive ..... (312)576-7800
**TEXAS**, Ft. Worth ..... (817)232-6255

### MOTOROLA SEMICONDUCTOR INTERNATIONAL SALES OFFICES

**AUSTRALIA**, Melbourne ..... (03)561-3555
**AUSTRALIA**, Sydney ..... 438-1955 439-2242
**AUSTRIA**, Vienna ..... (0222)65 01 26
**BRAZIL**, Sao Paulo ..... 707-286
**DENMARK**, Gladsaxevej ..... (01)67 44 22
**ENGLAND**, Wembley, Middlesex ... 01-902-8836
**FRANCE**, Grenoble ..... (076)90 22 81
**FRANCE**, Paris ..... (01)555-91-01
**FRANCE**, Toulouse ..... (061)41 11 88
**GERMANY**, Langenhagen/ Hannover ..... (0511)78-20-37
**GERMANY**, Munich ..... (089)92 481
**GERMANY**, Nurenberg ..... (0911)65761
**GERMANY**, Sindelfingen ..... (07031)83074
**GERMANY**, Wiesbaden ..... (06121)76-1921
**HONG KONG**, Hung Hom, Kowloon ..... 3-632201-8 3-336211-22
**ISRAEL**, Tel Aviv ..... 338973
**ITALY**, Bologna ..... (051)533 446
**ITALY**, Milan ..... (02)824 2021 824-2046
**ITALY**, Rome ..... (03)831 4746
**JAPAN**, Osaka ..... (06)305 1421
**JAPAN**, Tokyo ..... 03-440-3311
**KOREA**, Seoul ..... 261-7137
**MEXICO**, D.F. ..... (525)524-0706
**NETHERLANDS** Utrecht ..... (030)443 808
**NORWAY**, Oslo ..... (02)671467
**SCOTLAND**, East Kilbride ..... (03552)39 101
**SINGAPORE** ..... 2945438
**SOUTH AFRICA**, Bramley ..... 786 1184
**SPAIN**, Madrid ..... (01)279 0802
**SWEDEN**, Solna ..... 08/82 02 95
**SWITZERLAND**, Geneva ..... (022)991 111
**SWITZERLAND**, Zurich ..... (01)730 40 74
**TAIWAN**, Taipei ..... 7528944-9

**MOTOROLA** *Semiconductor Products Inc.*

P.O. BOX 20912 • PHOENIX, ARIZONA 85036 • A SUBSIDIARY OF MOTOROLA INC.

16718-7 PRINTED IN USA 12-83 IMPERIAL LITHO C18479 60,000