

intel

FAE

NEWSLETTER

Volume 6

Issue 2B

Feb 1984

SPECIAL ARCHITECTURE EDITION

# Microprocessor Operation

SPECIAL ARCHITECTURE EDITION

SPECIAL ARCHITECTURE EDITION

SPECIAL ARCHITECTURE EDITION

SPECIAL ARCHITECTURE EDITION

SPECIAL ARCHITECTURE EDITION

SPECIAL ARCHITECTURE EDITION

SPECIAL ARCHITECTURE EDITION

SPECIAL ARCHITECTURE EDITION

# CONTENTS

<u>Introduction</u>	1		
<u>In General</u>	1		
<u>Format of the Contents</u>	2		
<u>Intel Point-to-Point Discussion</u>		<u>Reference Section of Mot Paper</u>	
SUMMARY	3	SUMMARY .....	i.
FORWARD		FORWARD	
The M68000 Family:		The M68000 Family:	
Some Design Philosophies	5	Some Design Philosophies	
ARCHITECTURE	6	ARCHITECTURE .....	ii
Application	6	Application .....	ii
Instruction	6	Instructions .....	ii
Bus Structure	8	Bus Structure .....	ii
COMPATIBILITY	11	COMPATIBILITY .....	iii
Existing Software	11	Existing Software .....	iii
Compromise	11	Compromise .....	iii
Future	11	Future .....	iii
PERFORMANCE	15	PERFORMANCE .....	iii
Data Types	19	Data Types .....	iii
Register Store Size	22	Register Store Size .....	iii
Address Space Size	23	Address Space Size .....	iii
THE CREATION OF AN ADVANCED		THE CREATION OF AN ADVANCED	
ARCHITECTURE	24	ARCHITECTURE .....	iv
Architectural Size	24	Architectural Size .....	iv
Linear Addressing	25	Linear Addressing .....	iv
Data Register Size	27	Data Register Size .....	iv
General Purpose Registers	27	General Purpose Registers .....	iv
Register Set Size	27	Register Set Size .....	v
THE BEGINNING OF A FAMILY		THE BEGINNING OF A FAMILY .....	v
Family Members at Present	31	Family Members At Present .....	v
Extensions	31	Extensions .....	v
AN ARCHITECTURAL CONTRAST:		AN ARCHITECTURAL CONTRAST:	
The MC68000 Microprocessor Family		The M68000 Microprocessor Family	
and the		and the	
8086/iAPX 286	33	8086/iAPX 286	
GENERAL FEATURES	33	GENERAL FEATURES .....	1
Register Set Alternatives	33	Register Set Alternatives .....	1
Register Scheme - General		Register Scheme - General	
Purpose versus Dedicated	35	Purpose versus Dedicated .....	2
Source of the iAPX 286		Source of the iAPX 286 Register Set	
Register Set? - 8086	35	Problem? - 8086 .....	4
VIRTUAL MEMORY	37	VIRTUAL MEMORY .....	5
iAPX 286 Virtual Claims	37	iAPX 286's Virtual Claims .....	5
True Virtual Memory Facilities	40	True Virtual Memory Facilities .....	7
More Advantages with Mot's		More Advantages with Motorola's	
True Virtual Memory	40	True Virtual Memory .....	7

## Intel Point-to-Point Discussion

DATATYPES AND OPERATIONS	41
32 Bits Since the Beginning	41
Bit Manipulation	42
Register and Addressing Mode	
Flexibility	43
iAPX 286 Just Doesn't Stack Up	44
Indexed and Absolute Addressing	46
iAPX 286 does not Offer Absolute	
Addressing	47
Focus: Misleading Move	
Comparison	48
Program Counter Variances	50
INSTRUCTION SET	51
Stack Operation	52
String Manipulation	53
Branching	54
Memory-to-memory arithmetic	56
The iAPX 286 Single Accumulator	57
CODE COMPATIBILITY	
ACROSS A FAMILY	58
M68000 COMPATIBILITY	58
MC68000-MC68008 Compatibility	58
MC68000-MC68010 Compatibility	58
MC68000-MC68020 Compatibility	58
Application Level Object Code	
Compatibility	58
MC68010 Change 1	58
MC68010 Change 2	58
iAPX 286 COMPATIBILITY	59
Operational Modes	59
Segment Wrap	59
Bus Locking	59
Protected Mode Operation	59
8086 Application Program	
Compatibility	59
Segment Base Address	59
Carnegie-Mellon Benchmark Code	59
Guide for an 8086 Code Writer	59
Operating System Compatibility	59
Future Operating System	
Compatibility	59
Summary	59
PRIVILEGE LEVEL PROTECTION	60
MEMORY MANAGEMENT	63
iAPX 286 Segments	
Limit Compatibility	
with Today's Systems	65
Incrementing a Smalltalk or	
Graphics String Pointer	67

## Reference Section of Mot Paper

DATA TYPES AND OPERATIONS	8
32 Bits Since the Beginning	8
Bit Manipulation	8
Register and Addressing Mode	
Flexibility	9
iAPX 286 Just Doesn't Stack Up	10
Indexed and Absolute Addressing	11
iAPX 286 Does Not Offer Absolute	
Addressing	11
Program Counter Variances	13
INSTRUCTION SET	13
Stack Operation	14
String Manipulation	15
Branching	16
Memory-to-Memory Arithmetic	17
The iAPX 286 Single Accumulator	17
CODE COMPATIBILITY	
ACROSS A FAMILY	18
M68000 FAMILY COMPATIBILITY	18
MC68000 to MC68008 Compatibility	19
MC68000 to MC68010 Compatibility	19
MC68000 to MC68020 Compatibility	19
Application Level Object Code	
Compatibility	19
MC68010 Change 1	20
MC68010 Change 2	20
iAPX 286 COMPATIBILITY	20
Operational Modes	20
Segment Wrap	20
Bus Locking	20
Protected Mode Operation	21
8086 Application Program	
Compatibility	21
Segment Base Address	21
Carnegie-Mellon Benchmark Code	21
Guide for an 8086 Code Writer	22
Operating System Compatibility	22
Future Operating System	
Compatibility	23
Summary	23
PRIVILEGE LEVEL PROTECTION	23
MEMORY MANAGEMENT	24
iAPX 286 Segments	
Limit Compatibility	
with Today's Systems	25
Incrementing a Smalltalk or	
Graphics String Pointer	27

## Intel Point-to-Point Discussion

Artificial Intelligence Research Cannot Use iAPX 286	68
Dynamic Storage and Sophisticated Software Systems	69
Massive Descriptor Overhead for all iAPX 286 Native Mode Operating Systems	71
I/O INTERRUPTS ON THE MC68000 AND THE iAPX 286	72
MC68000 Version	72
Fastest iAPX 286 Version	72
Most Common iAPX 286 Version	72
Task Switch iAPX 286 Version	72
Focus:	
Fair Interrupt Comparison	74
Focus:	
Misleading iAPX 286 Access Time Claims	75
Summary	75
Intel's Architecture Foils Intel's Own Programmers	75
PACKAGING	76
Power Considerations	76
"Worst Case Calculations"	76
Conclusions	79
SUMMARY	81
COMMENT ON MOT APPENDIX A: MC68000 Pascal is 45% Faster than iAPX 286 Pascal	82
COMMENT ON MOT APPENDIX B: Independent Benchmarks Show MC68000 Faster than iAPX 286	84
COMMENT ON MOT APPENDIX C: iAPX 286 Substring Benchmark	88
COMMENT ON MOT APPENDIX D: Motorola MC68000 Quicksort	88
COMMENT ON MOT APPENDIX E: Intel 8086 Quicksort	88

## Reference Section of Mud Paper

Artificial Intelligence Research Cannot Use the iAPX 286	28
Dynamic Storage Areas and Sophisticated Software Systems ...	
Massive Descriptor Overhead for All iAPX 286 Native Mode Operating Systems	31
I/O INTERRUPTS ON THE MC68000 AND THE iAPX 286	31
MC68000 Version	31
Fastest iAPX 286 Version	31
Most Common iAPX 286 Version	32
Task Switch iAPX 286 Version	32
Summary	33
Intel Architecture Foils Intel's Own Programmers	33
PACKAGING	34
Power Considerations	34
"Worst Case" Calculations	34
Conclusions	35
SUMMARY	35
APPENDIX A MC68000 Pascal is 45% Faster Than iAPX 286 Pascal	36
APPENDIX B Independent Benchmarks Show MC68000 Faster Than iAPX 286	36
APPENDIX C iAPX 286 Substring Benchmark	37
APPENDIX D Motorola MC68000 Quicksort	39
APPENDIX E Intel 8086 Quicksort	42

## Intel Point-to-Point Discussion

Artificial Intelligence Research Cannot Use iAPX 286	68
Dynamic Storage and Sophisticated Software Systems	69
Massive Descriptor Overhead for all iAPX 286 Native Mode Operating Systems	71
I/O INTERRUPTS ON THE MC68000 AND THE iAPX 286	72
MC68000 Version	72
Fastest iAPX 286 Version	72
Most Common iAPX 286 Version	72
Task Switch iAPX 286 Version	72
Focus:	
Fair Interrupt Comparison	74
Focus:	
Misleading iAPX 286 Access Time Claims	75
Summary	75
Intel's Architecture Foils Intel's Own Programmers	75
PACKAGING	76
Power Considerations	76
"Worst Case Calculations"	76
Conclusions	79
SUMMARY	81
COMMENT ON MOT APPENDIX A: MC68000 Pascal is 45% Faster than iAPX 286 Pascal	82
COMMENT ON MOT APPENDIX B: Independent Benchmarks Show MC68000 Faster than iAPX 286	84
COMMENT ON MOT APPENDIX C: iAPX 286 Substring Benchmark	88
COMMENT ON MOT APPENDIX D: Motorola MC68000 Quicksort	88
COMMENT ON MOT APPENDIX E: Intel 8086 Quicksort	88

## Reference Section of Mud Paper

Artificial Intelligence Research Cannot Use the iAPX 286	28
Dynamic Storage Areas and Sophisticated Software Systems ...	
Massive Descriptor Overhead for All iAPX 286 Native Mode Operating Systems	31
I/O INTERRUPTS ON THE MC68000 AND THE iAPX 286	31
MC68000 Version	31
Fastest iAPX 286 Version	31
Most Common iAPX 286 Version	32
Task Switch iAPX 286 Version	32
Summary	33
Intel Architecture Foils Intel's Own Programmers	33
PACKAGING	34
Power Considerations	34
"Worst Case" Calculations	34
Conclusions	35
SUMMARY	35
APPENDIX A MC68000 Pascal is 45% Faster Than iAPX 286 Pascal	36
APPENDIX B Independent Benchmarks Show MC68000 Faster Than iAPX 286	36
APPENDIX C iAPX 286 Substring Benchmark	37
APPENDIX D Motorola MC68000 Quicksort	39
APPENDIX E Intel 8086 Quicksort	42

<u>Appendix A</u>	
80286 32-bit Multiply/Divide	89
<u>Appendix B</u>	
MC68000 32-bit Divide	90
<u>Appendix C</u>	
32-Bit Divide for iAPX 286	92
<u>Appendix D</u>	
iAPX 86/286 EDN Benchmark K: Bit Matrix Transpose	94
<u>Appendix E</u>	
MC68000 + MC68451 Task Switch Code	96

## Introduction

A paper published by Motorola titled "An Architectural Contrast: The MC68000 Microprocessor Family and the 8086/iAPX 286" has been distributed to many people considering a choice between Intel's iAPX 286 microprocessor and Motorola's MC68000 or MC68010 microprocessor. By reading its cover, one presumes the Motorola paper is positioned as a work of great revelation. Yet its major revelation is the absence of architectural growth in the Motorola MC68000 line to solve the more difficult problems facing designers today.

Additionally it presents many misleading comparisons, incorrect statements and incorrect conclusions regarding the iAPX 286. The writers of the comparison document apparently do not understand the iAPX 286 and seem confused with regard to comparing processors.

This paper, published by Intel to our Field Applications and Sales Engineers, corrects the inaccuracies of the Motorola document and accurately compares processors. Many customers are in need of more accurate, in-depth analysis which the Motorola document does not provide. Use or distribute accordingly.

## In General

The Motorola document appears to attempt to discredit Intel Microprocessors. The document is poorly written with a number of incorrect claims and arguments. From the Motorola document, the reader might draw the conclusion that Intel is a dishonest vendor actively seeking to mislead the world about its products.

Most of the comparisons made by Motorola are inaccurate or incomplete. For example, the EDN benchmark results quoted throughout are not the same as published by EDN. In another example, Motorola uses some PASCAL benchmarks to claim a 45% speed advantage for the 68000 based on byte count, while in reality those same benchmarks actually show a 2-3 times performance advantage for the iAPX 286. This paper also includes analysis of 32-bit arithmetic performance, which the iAPX 286 performs faster as well.

Motorola ignores other more important comparisons. CPU-to-CPU comparisons of code fragments are not very useful. Real comparisons require system-level benchmarks between complete systems running real programs. At minimum, CPU comparisons should be based upon an equally fast memory system available to each. But when Motorola compares an 12.5Mhz 0 wait-state 68000 against an 8Mhz 0 wait-state 80286, the 68000 consumes a memory resource 40% faster<sup>1</sup> than required by the iAPX 286... and leaves that untold to the reader.

<sup>1</sup> 68000 provides 150ns access time; iAPX 286 provides 242ns access time.

Furthermore, Motorola avoids discussing the performance impact of managing the 68451 MMU which in turn does memory management for virtual memory 68010 systems. Nor do they consider the wait-states it adds to every memory reference, whereas the integrated memory management of the iAPX 286 operates in parallel with other CPU units, adding no wait states. Nor do they consider advanced system-level features like floating point arithmetic, for which Intel offers specialized coprocessors (8087 and 80287).

Performance comparisons with real programs on real products show the iAPX 286/287 has a significant performance advantage over the MC68000. Recent comparisons of Xenix-286 against other 68000-based UNIX systems substantiate this ("The Intel iSBC 286/10 Single Board Computer as a Xenix<sup>1</sup> Engine" Intel order number 230676 and "Applications on Xenix 286 = Performance" Intel order number 230805). On such performance issues Intel receives unsolicited letters from real customers such as this written November 3, 1983:

"The benchmarks clearly show the benefit of the [286 and the] 287. Despite the fact that the 286 is running at 3.3MHz [in our system], it is still twice as fast as a 10MHz no wait state 68000 in our benchmark which solved a system of first-order differential equations. In fact, the 286/10 board system performed at VAX levels."

Motorola does not offer program-level comparisons of the iAPX 286 and 68000 on real products with real programs that accomplish useful work.

Benchmark measurements of microprocessors can show many different kinds of results. Each vendor attempts to show benchmarks that demonstrate their product is superior. With such apparently conflicting information, the customer should look a little deeper into the numbers. As a start, one can develop their own "feel" for the performance advantage of the iAPX 286 over competitive offerings simply by reviewing the instruction times given in the data sheets of the respective products.

#### Format of the Comments

The remaining topics of this document are arranged identically to the Motorola comparison. To provide balanced, accurate information and to simplify cross-referencing, the Motorola statements are extracted, and analysis presented, beginning ahead.

Unfortunately, the Motorola paper is melodramatic and contains numerous incorrect or rhetorical statements... yet it presents for Intel an opportunity to highlight to our customers iAPX 286 performance, and its features which even its future competitor, the future 68020 will not have.

No person or product is perfect, yet the iAPX 286 stands tallest in the marketplace!

<sup>1</sup> Xenix is a trademark of Microsoft Corporation



Motorola statements:

The MC68000 is presented as a family positioned against the 8086 chip. The 8086 is presumed ancient. The iAPX 286 is claimed to not support virtual memory (absolutely a FALSE statement).

Intel analysis:

Any microprocessor family development involves tradeoffs. Compatibility, performance, and efficient implementation are very important. The success and capabilities of the iAPX 86 family are directly attributable to our decision favoring those. The M68000 has been used by industry, often in low-volume applications. However, for good reason the iAPX 86 is the world-standard microprocessor architecture as measured by volume shipments and amount of software written for it. This paper explores those reasons.

The comments regarding ancient chip designs and inflexible philosophies of the iAPX 86 are unfounded. The capable iAPX 86 and iAPX 286 allow very efficient object code generation. The iAPX 286 demonstrates the flexibility of the iAPX 86 family in adding new functions to the architecture, in silicon, in a way that can be used by existing programs.

The iAPX 286 supports larger data areas than the 68000/10/20. Each task on an iAPX 286 possesses virtual memory space of 1 gigabyte ( $2^{30}$ ), compared to 16 megabyte total virtual for the 68010 and estimated  $2^{32}$  total virtual for the future 68020. In fact the virtual address space of the 68010 is not larger than its physical space, since only 24 bits of logical address information is passed to the MMU. Even with the 286's larger virtual area, the iAPX 286 has comparable performance to either of those products in accessing very large arrays. A comparison is shown later (see page ...). In addition, with Intel Program Management Tools like MAKE and SVCS, Intel development tools support development of large software systems much better than competitive offerings.

SVCS stands for Software Version Control System and it functions as a data base manager that automatically logs who makes changes, when and why. Then our MAKE facility automatically finds the correct versions of each module, automatically recompiles those modules that need, and produces the complete, correct system. Automatically of course.

Motorola's stating the iAPX 286 does not support virtual memory is decidedly misleading given the inherent support of virtual memory and protection in the iAPX 286. The iAPX 286 architecture provides a virtual memory and protection system much more comprehensive and effective than that supplied by Motorola. No significant architectural enhancements are forthcoming in the future 68020 to bring it within range of the supermicro architectural capabilities available today in the iAPX 286. This paper has more to convey about those capabilities.

The iAPX 86/286 family is certainly different from the M68000, however we demonstrate the advantages in performance, coprocessors, ease-of-use, code and

execution efficiency. The claim that the M68000 is superior to the iAPX 286 is not supported by anything published by Motorola.

The iAPX 286 brings microprocessors truly into the realm of multitasking when the CPU is put in its Protected Virtual Address Mode, where tasks are recognized entities handled efficiently by the hardware itself. By virtue of the iAPX 286 architecture itself, each task the user cares to install (the minimum is only one) possesses a separate state, separate stacks, and separate memory space up to 1Gbyte per task.

The complete task switch on the iAPX 286 is a single hardware operation which:

- o saves 22 registers
- o loads new values
- o verifies and loads new descriptors
- o transfers control

Total time = 22usec. Complete processor state change and local memory space change. Automatically invoked by JMP, CALL or Interrupt, via Task Switch Gate. The architecture provides high integrity, high performance multitask environments.

Motorola statement:

This particular set of "areas of concern" is declared.

1. Performance
2. Need for Product
3. Capability
4. Effort to Design In
5. Ease of Use
6. Reliability
7. Future

Intel discussion:

Examine Intel's track record on supporting these design needs years before Motorola. The 8086 had an 18 to 24 month lead over the 68000. The 8087 has a 2 to 3 year lead over the 68881, which has yet to become a reality. The 8089 has 2 to 3 years lead over the 68440. The 80286 has 18 to 24 month lead over the future 68020 still in development. The capabilities of Intel products actually exceed those of Motorola years ahead of Motorola's comparable product offerings.

The 8086 has been built into more products than Motorola microprocessors. Many different applications like robotic systems, graphic systems, Xenix workstations, industrial controllers, personal computers, and communication controllers have proven the capability and success of Intel processors.

Our product line breadth, documentation, development tools, field applications support, training and second-sourcing are not matched by any other microprocessor vendor. Therefore, Intel products are supported more completely than Motorola's.

Intel products are easier to design in. Our product family is supported with a fully-integrated development environment. Our support for operating systems, network development systems, development workstations, high level language compilers, Program Management Tools, software debuggers, hardware debuggers like I<sup>2</sup>C, and system products is not matched by Motorola or any other development system vendor.

Our component families are much easier to design with than Motorola's since we offer completely supported systems of components (80186, 82586, 82730, 8087, 80287, 80286, 8207, 8206, 82258, 8288, 8289, 82288, 82289) with all products designed to work together. Motorola offers only incomplete systems often requiring custom solutions for memory controllers, floating point, local area networks, system bus interface, high-performance memory management, etc.,. Thus competitive offerings burden the customer with MSI design for standard functions, at the cost of time, chance of error, consumption of power and board area.

## Motorola Section: ARCHITECTURE

Subsections: Applications, Instructions

Motorola statements:

Several statements are made concerning general purpose architectures being able to do reasonably well in most usages.

Intel discussion:

As Motorola states in the first sentence, there are many different philosophies for computer architecture.

There are also applications where special purpose architectures are better for certain applications than a general purpose architecture could be. Intel's implementation of functional partitioning, providing special-purpose components (80186, 8087, 80287, 8089, 82258, 82288, 82289, 80130, etc.) has thus been copied by Motorola to a limited degree in the 68000 product line. Motorola, however, does not have comparable products to the 80186 with its integrated peripherals, the 8087, 80287, 8089, or 80130.

Yet in many microprocessor applications one part must do several jobs to be provide effective computing power. A general purpose architecture is needed to efficiently perform different types of work. The 8086 is a general purpose microprocessor optimized for high level languages and compact program representation. The powerful 8086 architecture efficiently supports many different languages. The iAPX 286 extends this architecture in further support of high-level language, for example with ENTER and LEAVE instructions to setup and remove stack frames at any lexical nesting level. The iAPX 286 is also uniquely capable among all microprocessors, including competitive 32-bit offerings, of creating protected, multitask environments. Thus its name which is here to stay, the Supermicro.

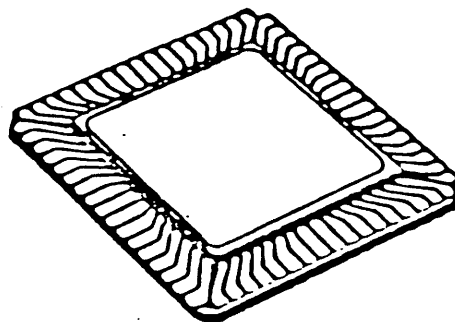


Figure 1: JEDEC Type A 68-Lead Chip Carrier becomes a Supermicro when 80286 is inside.

Studies like "Empirical Evaluation of Some Features of Instruction Set Processor Architectures"<sup>1</sup>, Lunde, Communications of the ACM, March 1977, pp. 143-153 show most compilers are unable to effectively use many registers. A general purpose architecture will be more effective in general if it allows general compilers to produce dense, efficient code. The key to high-performance compiled code is proper use of the CPU instructions and a fast CPU.

The combination of general purpose functions and specialized hardware has been very effective in the 80286.

The addressing modes of the 68000 were very expensive to implement. The 68000 could not support virtual memory systems because its auto-increment and auto-decrement address modes were not restartable. The 68010 required 30% more silicon than 68000 to fix the instruction restart problems of the 68000 and add minor performance enhancements. The 68010 came out over two years after the 68000.

For some arithmetic and addressing operations the 8086 registers are characterized to improve the encoding density of instructions. Short instructions reduce memory usage and get the most instructions executed with a limited bus bandwidth. For example, each 8086 memory reference instruction requires only 5 bits in the instruction to encode register usage, segment usage, and addressing modes (including index and base registers and three displacement sizes). The 68000 requires 6 bits to encode only one level of indexing with only two displacement sizes (0 and 16 bits). The 68000 and future 68020 require extra words to encode two-level indexing.

The 68000 claims an orthogonal instruction set. Orthogonality is best if it is uniformly applied. The 8086 offers more orthogonality in addressing modes and operand types than the 68000. 68000 programmers must also pay attention to special cases and write extra instructions to deal with them.

The 68000 addressing modes lack 32-bit offsets. To use a 32-bit offset with a 32-bit index register (only A0-A7, not D0-D7), the programmer must perform the addressing arithmetic with separate instructions. Using two index registers for addressing an operand is limited to an 8-bit offset. To index off a data register, the programmer must first zero an address register, then use the limited double index addressing form.

Many 68000 arithmetic instructions cannot use the address registers (TST, ASL, CLR, AND, etc.) as destinations. Some instructions can not use all the addressing modes, PEA (A0)+. Some instructions are limited to register operations (i.e. EOR and LSL).

By contrast, all of the addressing modes can be used in any memory reference instruction on the iAPX 286. All the iAPX 86 16-bit arithmetic instructions can use memory or registers interchangeably. iAPX 286 multiply and divide instructions do require that one operand and the result be in the accumulator, but perform with 290% and 564% less clocks than 68000, not including the 68000's effective address calculation time.

<sup>1</sup>Amund Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures", Communications of the ACM, 20, No. 3 (1977), pp. 143-153.

## Subsection: Bus Structure

---

### Motorola statement:

Bus structure should facilitate system design.

### Intel analysis:

Intel microprocessors simplify the hardware designer's job, not make it more complex. Intel microprocessors have the most efficient, highest performance, and general purpose microprocessor busses. The multiplexed bus of the 8086/8088 and 80186/80188 fits the most functions into the fewest pins than the 68000 or 680008. This allows lower cost components in 40-pin packages than large 64-pin DIPs.

The Intel busses support the same memory and peripheral configurations as a Motorola system and then some. Cases in point:

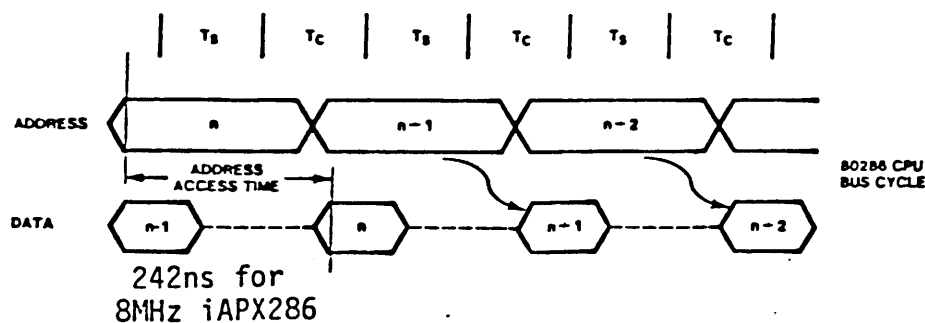
- 1) The Intel systems allow more cost-effective systems that use high-performance peripherals like the 82586 or 82730.
- 2) Neither does Motorola offer any product similar to the Dual-Port DRAM Controller 8207 for cost-effective dynamic memory systems up to 2Mbyte without external drivers. Dual-port support in the 8207 is vital for high-performance applications having multiple processing elements accessing common memory resources. Even for single-port designs, the 8207 still provides refreshing, 4-bank interleaving, and accepts status information directly from 86/88/186/188/286 processor for efficient interface.

Intel busses allow more address access time, as measured at the pins of the processor, than Motorola busses. The 8MHz 80186 allows 315ns of address access time in four 8MHz clocks as compared to only 290ns of address access time in the 8MHz 68000. The 8MHz 80286 allows 242ns of address access time in two 8MHz clocks. The 8MHz 68000 requires two more 8MHz clocks (250ns) to provide only 50ns more time than the 8MHz 80286.

The demultiplexed bus of the 80286 provides better utilization of memories in high-performance systems. The 12.5MHz 68000 uses 4 clocks per memory and has only 170ns of address access time out of 320ns total bus time. Only 53% of the bus cycle is usable! The 8MHz 80286 uses two clocks<sup>1</sup> per bus cycle and allows 242ns of its 250ns bus cycle to be usable, 97% efficient!

Throughout the paper, Motorola claims 80ns memories are required for the iAPX 286 8MHz, 0 wait state operation, which is incorrect. In fact, 242ns of address access time is available to the rest of the system, including buffers and DRAM chips themselves.

<sup>1</sup> The term "clock" used throughout this paper and in 80286 data sheet where instruction execution times are listed, refers to the internal processor clock (which is half the frequency of the CLK signal generated by the 82284 support chip).



Pipelined Bus Operations

Figure 2: 8MHz iAPX 286 provides 242ns access time for code or data

The 80ns figure used by Motorola is not accurate and applied out-of-context. The number is possibly taken from the iAPX 286 Hardware Reference Manual (order number 210760) page 4-12, from the table below.

Memory Interface Technique	Address Access Time Provided by Single-Buffered System with 0 Wait States		Usage Guidelines
	8 MHz	6 MHz	
ALE-Controlled Address Latches	87.5 ns	139.9 ns	Useful for implementing small bootstrap EPROMs; 2 waits at 8 MHz suitable for 2732A-3 or 2764A-3.
Special Address Strobe Logic	155 ns	228.3 ns	Useful for small static RAM systems; memories require separate latches; insufficient address hold time for I/O devices.
Interleaved Memories	180 ns	254 ns	Useful for large EPROM or static RAM arrays; 2 sets of address latches required, with 6 TTL DIPs to control interleaving.
8207 DRAM Controller (uses internal interleaving)	120 ns RAS 58 ns CAS	183 ns 89.6 ns	Useful for dynamic RAM arrays.

Figure 3: Comparing Several Memory Interface Techniques

Note the Usage Guideline for the "ALE-controlled Address Latches" technique above. It is not intended for DRAM access. Yet Motorola uses an 80ns number, even less than the ALE method provides, perhaps to paint a sour picture. But the sour picture is incorrect!.

When looking at the correct techniques for DRAM interfacing, the last two in the table, the access time available to the DRAM chips themselves is excellent.

Throughout the Motorola document, Motorola uses the incorrect 80ns access time, to further damage the 80286 INCORRECTLY on pages 13 and 17 of their paper: the document says that with "equal speed memories 200ns" the iAPX 286 would have to slow down by as much as a factor of two. This is doubly misleading!!

- 1) Firstly, iAPX 286 memory speed required is not 80ns, but rather 150,
- 2) In reality, the memory resource consumed by the 68000/68010 must be faster than that of iAPX 286.

When access times available at the processor pins are compared accurately, one sees that the Motorola configurations are the ones which provide less access time than the iAPX 286!!! The Intel access times are not less than the Motorola times.

Table 1: Total Access Times at the processor pins  
for the situations Motorola's document compares

(see for example on page 37 of their document):

<u>Processor Configuration/Mode</u>	<u>Speed</u>	<u>Wait States</u>	<u>Access Time</u>
68000, no MMU	12.5MHz	0	170
68000, with fastest 68451 MMU	10MHz	1	165
iAPX 286, protected virtual	8MHZ	0	242ns *

\* Motorola's paper incorrectly states that at least 80ns memories are required.

The Motorola disparity obviously favors the 68000 by requiring a faster, more expensive memory resource.

Intel's approach, to perform a more fair comparison, with equivalent memory resources available to each processor, adjusts the wait states required for memory access as shown below, yet still allows the fastest speed selection of each processor to be used for the comparisons.

Table 2: Total Access Times at the processor pins  
for the situations this Intel document compares:

<u>Processor Configuration/Mode</u>	<u>Speed</u>	<u>Wait States</u>	<u>Access Time</u>
68000, no MMU	12MHz	1	250
68000, with fastest 68451 MMU	10MHz	2	265
iAPX 286, protected virtual	8MHZ	0	242ns *

The Table 2 configurations to be compared in this paper place the microprocessors on equal footing from a memory resource standpoint.

Therefore to compare with equivalent memory resources, we will compare the following processor speeds and wait states:

- 68000/10 no MMU: 12.5MHz, 1 wait states,
- 68000/10 with 68451 MMU: 10.0MHz, 2 wait states,
- 80286 in either Real or Virtual address mode: 8.0MHz, 0 wait state.



## Motorola's Section: COMPATIBILITY

---

Subsections: Existing Software, Compromise, Future

Motorola's statements:

Existing software investment is an important compatibility consideration. The architecture should, however, be forward extensible by design.

Intel analysis:

No disagreement with the idealistic statements of this section. Yet in practice, no significant Motorola architectural extensions are alluded to anywhere in the Motorola document. Granted, the future 68020 will eventually have an optional coprocessor hanging off to the side and an MMU peripheral in front, to perform address translation, but the future 68020 will not be capable of addressing the multitasking market as does the iAPX 286 and its numeric coprocessor do today.

The reason is architectural. As Motorola states, the MC68000 is a flat architecture, very flat. The MC68000 recognizes only one type of "space", an undifferentiated linear memory space. The iAPX 286, on the other hand, recognizes several spaces, first a memory space (segmented), additionally an I/O space, and when in Protected Virtual Address Mode a task space. Task space permits multitasking (sometimes called windowing). Cognizance of a "task space" permits the iAPX 286 to keep Bob Smith's task separate from Mary Jones's task, and keep them both separate from another, real-time, interrupt-driven task that is, for example, pulling information off a communication network. The memory addresses are virtual, with the associated physical address depending on the task as well as the logical address itself. Thus each task has a separate virtual memory space.

The MC68000 architecture will be making NO moves into this important dimension and will continue to exist in a flat, one-space environment. Don't expect many new things to happen in a long, straight MC68000 tunnel. We can expect to hear much noise about 32-bits of this and that in the future, but the 68000 architecture will not be addressing the multitasking market of the 80's with any reasonable degree of integrity at the hardware level.

The forward extension of the Intel architecture has been accomplished with a very high degree of compatibility in the iAPX 286. And the performance of the 80286 is offered also to the existing single task applications with complete compatibility via the Real Address Mode of the 80286.

It should be of great concern to Motorola's customers, that all the sections devoted to the future (page iii 'Future', and page v, second column) are devoid of any substantial architectural development. Only the most cursory type of lip service is given to what ought to be an area of substantial evolution, an area of intense, long-term planning, an area yielding tangible new capabilities in the architecture.

Widening a data bus, or adding several new instructions (even if they happen to be implemented by a coprocessor) barely qualify as innovation. Of more concern to the customer, they provide no new capabilities for cost-effectively dealing with 1980's environments, where individual and secure tasks need to be hardware recognized, not software emulated.

In the personal computer market, for example, programmers have already begun emulating a multitasked (windowed) computer. Packages such as VisiOn, or Microsoft Windows, or Lisa, partially emulate a multitask environment, but without the hardware separation of stacks and address spaces provided by the iAPX 286. Because they run on single-tasking microprocessors these environments are not secure, since any procedure (as opposed to a true task) may reach across into another's code or data. And because the entire system may be brought down by a bug in any one procedure, the single-tasking microprocessor such as the MC68000/10/20 is not nearly as suitable for reliability-critical applications as the iAPX 286.

On the issue of compatibility, discussion by Motorola totally misses the mark: the desire for compatibility does not justifiably end. Compatibility is an issue gaining more importance over time as the investment in software grows. Doing something different for the sake of being different doesn't make sense when \$3 billion\* of personal computer software alone expects registers, instructions, and hardware to work a certain way.

Given the current and projected shortage of programmers, who possess an expensive talent, who can afford to let a programmer rewrite a program with new bugs and different operation if the existing one works fine?

Intel recognized the importance of compatibility early on. Our 16-bit microprocessor family is much more compatible with our 8-bit family than Motorola's. One reason for the success of the 8086 was the relative ease of moving software from the 8-bit world to the 16-bit world. Intel will continue to support compatibility by making it even easier to move from the 16-bit to 32-bit world in the iAPX 386.

\* Future Computing Estimates total cumulative market for iAPX 86 based personal computer software is \$3 billion.

Motorola's view of rejecting software compatibility from the 8-bit world has hurt the 68000. Data interchange between the 68000 and rest of the 8- 16- and 32-bit world will always be a problem. The 68000 does not use the same byte ordering for integers as does the Intel 8086, 80186, 80286, 8080, 8085, or VAX, or NS16032, NS32032, Z80, or even 6809(!). As local area networks become more important, 68000 systems will have to suffer converting integer formats to talk with the networks based on the 16-bit standard.

The iAPX 86 family is designed with more future expansion potential than the M68000 family. Segmentation allows the iAPX 86 architecture to be expanded in such a way that old software can use new features.

The 68000 requires cumbersome programming practices be followed by applications programs to be compatible with future 68020 . The iAPX 386 imposes no such constraints on iAPX 286 programs. Three examples of required 68000 programming rules are listed here.

- a. All 68000 programs must correctly leave 0's in the upper 8-bits of addresses to work on future 68020 . Since the 68000 does not check the upper address bits, programs that violate this rule will still work on a 68000 but fail on a future 68020 .
- b. 68000 programs must correctly define the double indexed address mode with 0s in bits 10-8 to be sure the future 68020 will correctly interpret this field the same as the 68000. If done incorrectly, the 68000 program will work, but the future 68020 will execute something different.
- c. Since no floating point hardware exists for the 68000, software emulation is used. Software emulation of the 68881 is very slow. Many systems use special hardware boards or different software floating point to be fast. Direct calls to floating point hardware or software routines is not compatible with future 68020 and 68881. Such programs must be recompiled, or rewritten if in assembler, to run on the future 68020 .

An operating system for the 68000 or 68010 will probably have to be rewritten to deal with the larger address space of the future 68020 and different memory management hardware. The 68451 does not support 32 address bits. Current custom memory managers for the 68000/68010 probably do not support 32-bit address either (it costs more money and board space to map 8 extra bits that aren't used).

Only the iAPX 86 family has segmentation which allows expansion of the processor architecture. The key to long-range performance in a processor architecture is integrating high-level functions into the silicon in a manner that lets old software use the new features.

The iAPX 286 added new operations, i.e. task switches and call gates, to the iAPX 86 family without requiring new instructions. Older iAPX 86 software can use the new features without any changes. The smooth transition is performed with segmentation. These new high-level system functions are performed by hardware instead of slow external instructions.

The 68000 architecture is self limiting and running out of capabilities. It allows only 32-bit pointers. Intel's iAPX 386 with segmentation allows 48-bit pointers. The expanded address space of the iAPX 386 will become very important in the future when large files are directly mapped into virtual memory and available over networks. Video disks are expected to contain over 10 gigabytes of information on inexpensive media. The single 4 gigabyte 68020 address space will not be capable of containing several very large files.

## Motorola's Section: PERFORMANCE

### Motorola statement:

Data types, register store size, and address size are facets of microprocessor performance. The more numerous or larger they are, the better.

### Intel analysis:

The Motorola performance section talks in generalities with no specific examples. When looking at examples, the iAPX 286 is faster. The reason the 80286 is so fast is that it has twice the transistors of the 68000 (133,000 vs. 68,000) to do the job. The transistors are used in a pipelined design of

- a) instruction unit,
- b) execution unit,
- c) address unit, and
- d) bus unit that very efficiently uses memory bandwidth.

The fully protected 80286 memory read instruction executes in 5 clocks. The 8MHz 80286 can sustain a 1.6 MIP rate executing load instructions with 242ns of address access time. With 250ns of address access time, the unprotected 12.5MHz 68000 runs at .833 MIPs (3 bus cycles at one wait state each to provide 250ns of access time). The protected 10MHz 68000 with 265ns of address access time runs at .555 MIPs (3 bus cycles at two wait states each: one to permit address translation in the MMU and one to provide access time as the efficient iAPX 286 bus provides without wait states).

The 80286 uses 32-bit data types faster than the 68000. The following program examples show the inherent speed of the 80286.

### So let's plunge right in to 32-bit math performance with 32-bit arithmetic comparisons!

The MC68000 has instructions for 32-bit add and subtract. But subroutines must be used for 68000 32-bit multiply and divide where both operands are 32-bit values. The iAPX 286 requires multiple instructions for all these functions.

To illustrate the basic speed of the iAPX 286 in 32-bit arithmetic, several examples are examined. In these examples, unsigned binary numbers are assumed. Except for divide, the calculation is straightforward with easily predictable execution times. Both processors use the same algorithm for the multiply and divide subroutines. Appendix A lists the 32-bit multiply code for both the 80286 and 68000. Appendix B and C shows the 32-bit divide code.

Because the divide algorithm depends upon the value of the operands, the divide subroutine has three sections, with various operand values. All three sections were measured and their results averaged. For two of them the 32/16 bit unsigned divide was used to calculate the result. However, when a 32-bit divisor is used, the quotient is calculated one bit at a time. The execution time depends on how many zeros appear in the result. A quotient of 64<sub>16</sub> was assumed for this section of code.

The following tables quantify the execution times for these 32-bit operations. As explained in the section on Bus Structure, to equalize system cost a customer must pay for this performance, these figures assume equivalent address access times, as measured at the processor pins (or MMU address pins), for all the processors.

- 1) The protected, virtual memory 80286 is running at 8MHz in protected mode with no wait states. The 8MHz 80286 at 0 wait states provides 242ns of address access time.
- 2) The virtual memory 68000 is assumed to use the 68451 MMU at 10MHz (fastest speed available) with two wait states per memory cycle, providing 265ns of address access time.
- 3) The real memory 68000 runs at 12.5MHz with one wait state per memory cycle, providing 250ns of address access time.

The following table shows the 68000 instructions used for the comparison of 32-bit arithmetic speeds. The multiply and divide subroutine source is in appendix A and B. The wait state counts are the number of bus cycles performed by these instructions. The wait states equalize the address access times of the processors as mentioned. It is also costly to build a microprocessor system with significantly less than 240ns of total access time since 100ns DRAM's become necessary. DRAMs of such speed are expensive first in terms of unit cost, and in terms of PC board area, since only 16K DRAMs are available with 100ns access time.

Table 3: MC68000 32-bit arithmetic code

32-bit add/sub in registers

ADD.L Dx,Dy      6 clocks + 1 clock per wait state

32-bit add/sub from memory using static addressing

ADD.L mem,Dx      22 clocks + 5 clocks per wait state

32-bit multiply (subroutine code shown in Appendix A)

JSR      dmul      278 clocks + 19 clocks per wait state

32-bit divide for cases of: 10000/100, 10<sup>8</sup>/100, and 10<sup>8</sup>/10<sup>6</sup> (z=29)  
(subroutine code shown in Appendix B)

JSR      ddiv      236/392/1953 clocks + 22/28/232 clocks per wait state

Table 4: 32-bit arithmetic times for MC68000

Function	Average clocks	Real memory 12.5MHz 68000 1 wait time	Virt. memory 10MHz 68000 2 wait time
Add/Subtract	14 + 3/wait	1.36 usec	2.0 usec
Multiply	278 + 19/wait	23.76 usec	31.6 usec
Divide	860 + 94/wait	76.32 usec	104.8 usec

The 32-bit iAPX 286 code is shown next. The multiply subroutine source is in Appendix A, while Appendix C has the divide source. No wait states are required for the 8MHz 80286 because it allows 242ns of address access time without wait states. — — — —

Table 5: 80286 32-bit arithmetic code

32-bit add/sub in registers

ADD	AX,CX	2 clocks + 1 per wait state
ADDC	DX,SI	2 clocks + 1 per wait state

32-bit add/sub from memory using static addressing

ADD	AX,memloc	7 clocks + 3 per wait state
ADDC	DX,memloc+2	7 clocks + 3 per wait state

32-bit multiply (subroutine code shown in Appendix A)

CALL	dmul	131 clocks + 13 per wait state
------	------	--------------------------------

32-bit divide for cases of: 10000/100 and 10<sup>8</sup>/100 and 10<sup>8</sup>/10<sup>6</sup> (z=29)  
(subroutine code shown in Appendix C)

CALL	ddiv	92/128/1133 + 18/24/404 per wait state
------	------	--

Table 6: 32-bit arithmetic times for Protected, Virtual Memory 80286

Function	Average	8MHz 0-wait time
add/sub	9 + 4/wait	1.125 usec
multiply	131 + 13/wait	16.375 usec
divide	451 + 148/wait	56.375 usec

The following table summarizes the results. To average them, the execution times are converted into performance, shown in parenthesis, using the iAPX 286 as the standard (1.0). A lower performance number means less processor performance for the same job.

Table 7: Comparison of 32-Bit Arithmetic Times

Function	Prot, Virt. iAPX 286 8MHz 0-wait	Real mem. 68000 10MHz 2-wait	Virt. mem. 68000 12.5MHz 1-wait
Add/Subtract	1.125 usec (1.0)	2.0 usec (.562)	1.36 usec (.827)
Multiply	16.375 usec (1.0)	31.6 usec (.518)	23.76 usec (.689)
Divide	56.375 usec (1.0)	104.8 usec (.538)	76.32 usec (.739)
Average	----- (1.0)	----- (.539)	----- (.752)

The iAPX 286 is faster than the 68000 for all types of four-function arithmetic. This includes cases when the iAPX 286 is in Protected Virtual memory mode. Because the iAPX 286 takes the protection and memory management overhead when loading the segment register (and the overhead is only 15 clocks per segment register loaded), the protected, virtual memory iAPX 286 has no performance penalty for subsequent memory accesses, such as for math operations, and handles 32-bit arithmetic faster than even the unprotected, real address 68000 (fully unencumbered from the 68451 MMU)!



## Subsection: Data Types

Intel supports more datatypes now than the MC68000 architecture will. The following illustrates the numeric types supported by the iAPX 286:

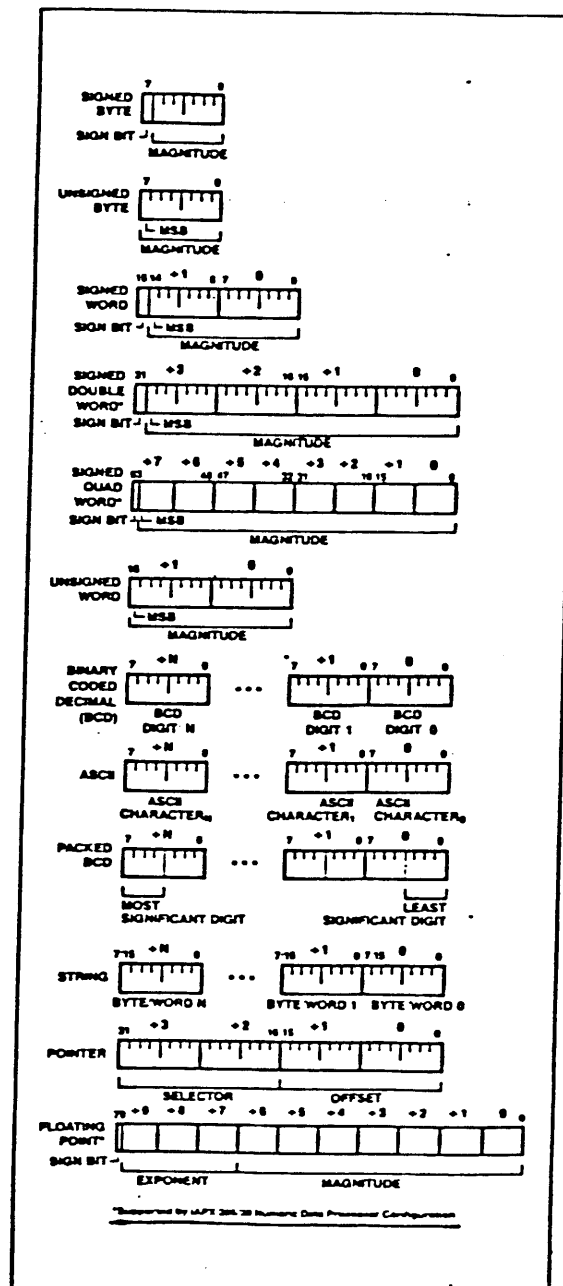


Figure 4: Numeric Types Supported by the 80286

The MC68000 does not support Packed BCD, and only in the future will the most expensive member of the family, the MCfuture 68020, provide an automatic interface to its planned coprocessor, the 68881.

The Intel 80287, coprocessor of the iAPX 286 family, adds automatic support for these additional numeric types. They include 16, 32, and 64 bit integers, 32, 64, and 80-bit floating point types, 19-digit packed BCD strings with type conversions between types supported automatically by the coprocessor. Functions with these types execute up to 100 times faster than their functional equivalent running in software.

Table 2. 80287 Datatype Representation in Memory

Data Formats	Range	Precision	Most Significant Byte				HIGHEST ADDRESSED BYTE											
			7	0	7	0	7	0	7	0	7	0	7	0	7	0		
Word Integer	$10^4$	16 Bits	<div><div>S</div><div>MAGNITUDE</div><div>TWO'S COMPLEMENT</div></div> <div><div>15</div><div>0</div></div>															
Short Integer	$10^9$	32 Bits	<div><div>S</div><div>MAGNITUDE</div><div>TWO'S COMPLEMENT</div></div> <div><div>31</div><div>0</div></div>															
Long Integer	$10^{19}$	64 Bits	<div><div>S</div><div>MAGNITUDE</div><div>TWO'S COMPLEMENT</div></div> <div><div>63</div><div>0</div></div>															
Packed BCD	$10^{18}$	18 Digits	<div><div>S</div><div>X</div><div>MAGNITUDE</div></div> <div><div>79</div><div>72</div></div>															
Short Real	$10^{\pm 38}$	24 Bits	<div><div>S</div><div>BIASED EXPONENT</div><div>SIGNIFICAND</div></div> <div><div>31</div><div>23</div><div>0</div></div> <div><div>1.6</div></div>															
Long Real	$10^{\pm 308}$	53 Bits	<div><div>S</div><div>BIASED EXPONENT</div><div>SIGNIFICAND</div></div> <div><div>63</div><div>52</div><div>0</div></div> <div><div>1.6</div></div>															
Temporary Real	$10^{\pm 4932}$	64 Bits	<div><div>S</div><div>BIASED EXPONENT</div><div>I</div><div>SIGNIFICAND</div></div> <div><div>79</div><div>64</div><div>63</div><div>0</div></div>															

NOTES:

- (1) S = Sign bit (0 = positive, 1 = negative)

(2) d<sub>n</sub> = Decimal digit (two per byte)

(3) X = Bits have no significance: 8087 ignores when loading, zeros when storing.

(4) Δ = Position of implicit binary point

(5) I = Integer bit of significand; stored in temporary real, implicit in short and long real

(6) Exponent Bias (normalized values):  
 Short Real: 127 (7FH)  
 Long Real: 1023 (3FFH)  
 Temporary Real: 16383 (3FFFH)

(7) Packed BCD:  $(-1)^S (D_{17} \dots D_0)$

(8) Real:  $(-1)^S (2^{E-BIAS}) (F_0 F_1 \dots)$

Figure 5: Additional Numeric Types Supported by the 80287 Coprocessor

The 68000 offers no comparable numeric types at present.

The special system types of the iAPX 286 have been ignored by the Motorola paper. Motorola offers no counterpart to these supermicro system types! Hardware recognition of these system types are the basis of the iAPX 286's multitasking capabilities. These include a special state segment for each task, called the Task State Segment. There are descriptors for these Task State Segments, for code and data segments, and descriptors which act as gates to higher privilege resources.

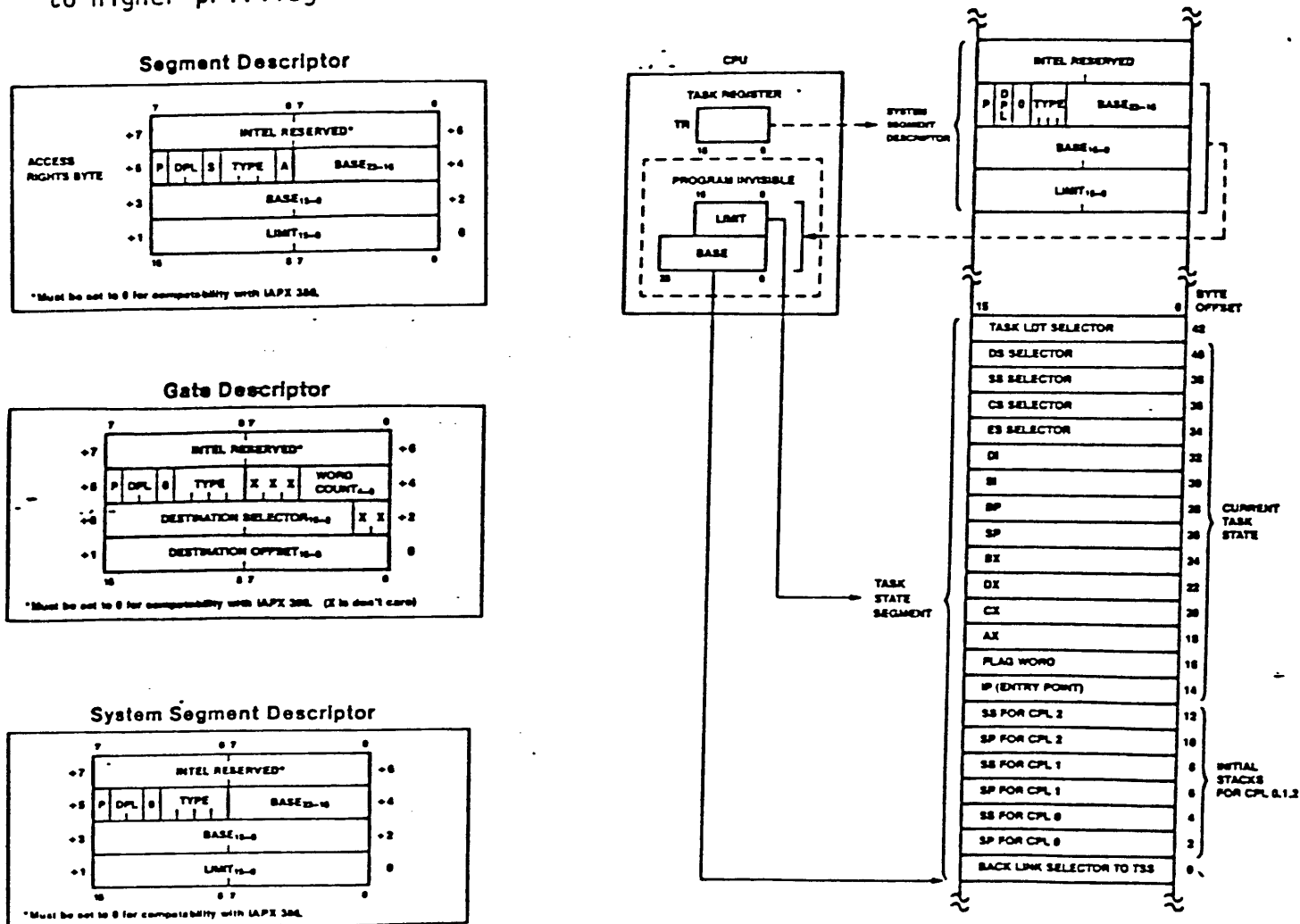


Figure 6: iAPX 286 System Types

Task State Segment and TSS Registers

## Subsection: Register Store Size

The onboard registers are a means of cacheing data in the processor. The working set of the iAPX 286 consists of eight 16-bit registers of which four can have uses as 16-bit base or pointer registers. There are also four 16-bit user-addressable segment registers that point into the memory-resident descriptor tables containing the base, limit, type, and access control for each segment. That information is cached into the 48-bit cache for each segment register. In addition to an instruction pointer and a flag register, is the machine status word which provides control for emulation of the coprocessor and other functions. The task register for the currently-executing task, and three descriptor table registers (one each for the GDT, IDT, and LDT) complete the register set.

The best tradeoff is to have sufficient registers for efficient compiler use but no more than necessary. Independent studies have shown that the number of general registers on the iAPX 286 is sufficient for maximum performance of optimizing compilers. The Lunde study below, for instance, reveals the average number of live registers required for a variety of functions each written in several languages.

Studies such as Lunde's show the eight working registers of the iAPX 286 set do provide adequate working space. The registers are efficiently encoded by the instruction set as well.

Table 8: Average Number of Live Registers<sup>a</sup>

Language:	ALGOL	BASIC	BLISS	FORTFOR	FORTEN	Mean
Bairstow	4.4	3.6	3.8	3.8	4.0	3.9
Crout	6.0	3.7	4.7	6.4	6.0	5.4
Treesort	2.5	3.5	3.1	1.8	2.7	2.7
PERT	4.2	3.6	3.6	2.0	3.0	3.3
Havie	6.0	3.5	3.7	3.6	4.5	4.3
Ising	6.5	-	3.6	1.9	3.2	3.8
Secant	-	-	-	3.0	3.4	3.2
Mean	4.9	3.6	3.8	3.2	3.8	3.8

**Notes:**

Bairstow - Polynomial roots by Bairstow's method

Crout - Linear equations by Bairstows method

Treesort

PERT

Havie - Numerical Integration by Havies method

Ising - Generation on Ising configurations

<sup>a</sup>Amund Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," Communications of the ACM, 20, No. 3 (1977) pp 143-153.

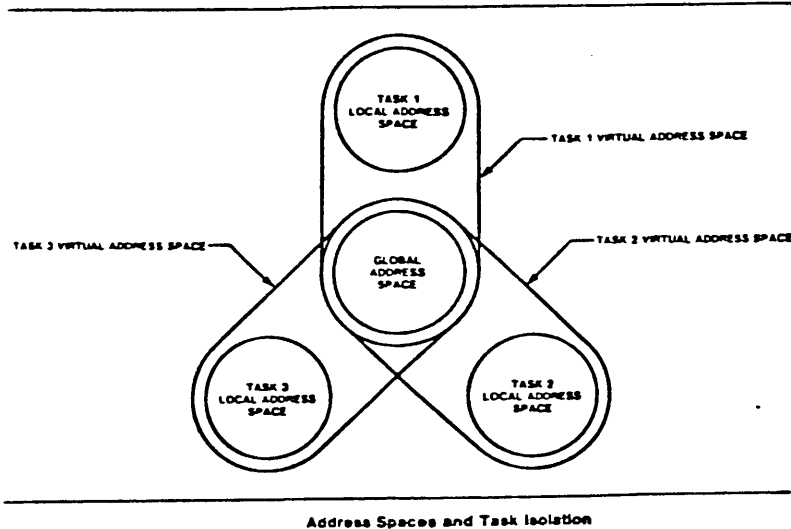
## Subsection: Address Space Size

The address size of the iAPX 286 is the LARGEST found on any microprocessor. Compare:

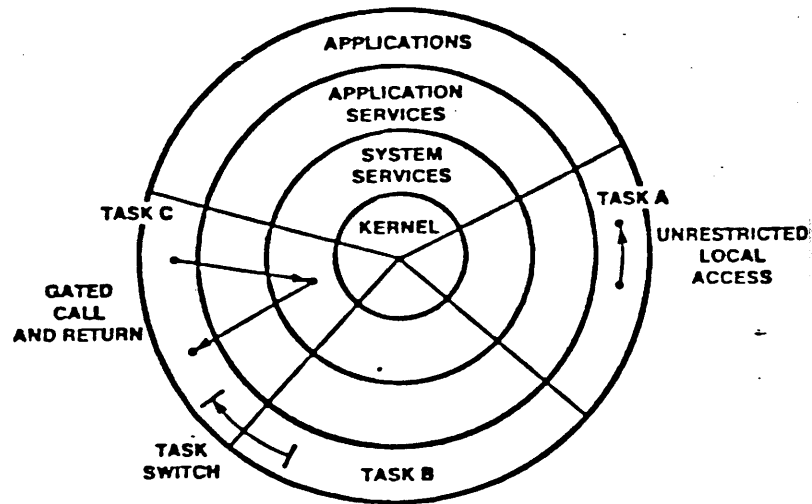
- 1) The iAPX 286 has a virtual address space of 1 Gbyte ( $2^{30} = 1,073,741,824$ ) per task.
- 2) The 68000 and 68010 each have total virtual address space = physical address space = 16 Mbyte ( $2^{24} = 16,777,216$ ).
- 3) The future 68020 will probably have a total address space of  $2^{32}$  bytes to be shared by the entire system, four times what the iAPX 286 has available per single task, yet since the iAPX 286 can easily accommodate as many as 2000-4000 tasks, only a fraction of total maximum iAPX 286 capability.

Half the 1 Gbyte available to each task belongs solely to that task alone. The remainder is within a global address space, but may be protected from undesired access through the mechanisms of privilege levels or gates.

The protected, virtual iAPX 286 address space may be conceptualized as follows.



Emphasizing each task having  
1/2 Gbyte private and  
1/2 Gbyte global address space.



Emphasizing protection levels  
assigned to portions of each  
task's virtual address space.  
Each task shown as one slice  
of "the pie."

Figure 7: Two Conceptualizations of iAPX 286 Virtual Address Space

## Motorola's section: THE CREATION OF AN ADVANCED ARCHITECTURE

### Subsection: Architectural Size

#### Motorola statements:

Motorola invents the term "architectural size." The address lines were extended to 24 bits in the 68000/10 and will be to 32 bits in the future.

#### Intel analysis:

By "architectural size" Motorola apparently wishes to say "memory size," which Intel has discussed to show the iAPX 286 has a larger virtual address space (the address space available to the programmer) than any other microprocessor including its competitor the future 68020 .

## Subsection: Linear Addressing

---

### Motorola statement:

When designing the MC68000, Motorola chose a flat, linear memory model.

### Intel analysis:

The 68000 flat memory model has advantages and disadvantages. The advantage being that the largest one-component address can be as large as the virtual address space. The disadvantages being that program modularity, protection and sharing are not provided for in the architecture. Even with a memory management unit (peripheral), the flat architecture is not growable to the reliability, protection, and efficiency features required for multitasking applications as was the architecture of the 8086 to the 80286.

Programmers normally require the ability to group their information into content-related or function-related blocks, and the ability to refer to these blocks by name. Four goals of modern computer systems, such as defined by Peter J. Denning<sup>1</sup>, each force the system to provide the programmer with means of handling the named blocks of a task's address space.

- Goals - a) Program modularity - each program constitutes a named block which is subject to recompilation and change at any time.
- b) Varying data structures - the size of certain data structures may vary during use, and it may be necessary to assign each structure to its own, variable size block.
- c) Protection - program modules must be protected from unauthorized access.
- d) Sharing - programmer A may wish to share module S with programmer B, even though S occupies addresses which A has already reserved for other purposes.

The segmented address space achieves these objectives. Address space is regarded as a collection of named segments each being a linear array of addresses. In a segmented address space, the programmer references an information item by a two-component address (s, v) in which s is a segment name and v the variable name within s.

By allocating each program module to its own segment, a module's name and internal addresses are unaffected by changes in other modules; thus, the first two objectives are satisfied. By associating with each module certain access privileges (e.g. read, write, instruction fetch), protection may be enforced. By enabling the same segment to be known in different tasks under different names, the fourth objective may be satisfied.

The size of an iAPX 286 segment is arbitrary up to a maximum of 64 Kbytes (65,536). In the majority of cases, 65,536 bytes is more than adequate for any given module which exists in the system. There are some instances where homogenous data structures are larger than 64K bytes. This paper addresses all points of the iAPX 286 architecture as they stand. These cases, where

<sup>1</sup> Peter J. Denning, "Virtual Memory," Computing Surveys, 2, No 3 (1970), pp. 153-189.

several segments are used to hold one logical module are handled automatically by high-level language compilers such as Pascal-286 and Fortran-286. This paper lists the sequence of a few instructions which map a single-component address into the two-component (s, v) address used by iAPX 286.

Motorola appears focused on the single issue of maximum segment size. This single architectural point is revisited over and over again at least fourteen times. Our competition is groping to find any issues at all which may be less-than-optimum in certain applications. Against a sound AND UNIQUELY CAPABLE architecture, Motorola picks over the same one issue, and without ever presenting the small scope of its impact. Maximum segment size is not likely to be an issue in most applications.



Subsections: Data Register Size, General Purpose Registers, Register Set Size

Motorola statements:

The MC68000 contains 32-bit registers. They are general purpose registers.

Intel analysis:

Because the 68000 and 68010 have actual datapaths 16 bits wide (half the logical register width), the iAPX 286 is able to perform 32-bit arithmetic operations faster than either. The simple 32-bit operations the 68000 and 68010 perform occur as a result of chaining two 16-bit operations together with microcode. The 68000 and 68010 do not perform 32-bit multiplication or division because it is not possible to "chain" together two 16-bit multipliers or dividers. As shown pages 17-20, the 80286 is faster than the 68000 or 68010 when performing 32 bit arithmetic: add, subtract, multiply and divide.

For some arithmetic and addressing operations the 8086 registers are characterized to improve the encoding density of instructions. The register set of the 80286 is general purpose for many operations as shown in Table 9. The register usage and addressing modes for operations using the eight 80286 registers in a general purpose contain an addressing-mode byte called the mode register/memory byte, Figure 8. This byte specifies the effective address calculation mode. Included among the effective address choices are the eight general registers themselves.

		MOD			
		00	01	10	11
R/M	000	BX + SI	BX + SI + d8	BX + SI + d16	AX/AX
	001	BX + DI	BX + DI + d8	BX + DI + d16	CL/CX
	010	BP + SI	BP + SI + d8	BP + SI + d16	DL/DX
	011	BP + DI	BP + DI + d8	BP + DI + d16	BL/BX
	100	SI	SI + d8	SI + d16	AX/SP
	101	DI	DI + d8	DI + d16	CH/BP
	110	d16	BP + d8	BP + d16	DH/SI
	111	BX	BX + d8	BX + d16	BH/DI

Figure 8: Mode Register/Memory Byte provides symmetrical register usage and addressing modes.  
Most 80286 instruction encodings include it.

Specifically, all 80286 instructions below contain the mode register/memory byte to use the registers generally:

Table 9: Symmetrical iAPX 286 Instructions, Containing the Mod/RM Byte

MOV	register to register/memory
MOV	segment register to register/memory
MOV	register/memory to segment register
PUSH	memory/register to stack
POP	stack to memory/register
XCHG	register/memory with register
LEA	load effective address to register
LDS	load pointer into DS and register
LES	load pointer into ES and register
ADD	register/memory with register to either
ADD	immediate to register/memory
ADC	register/memory with register to either
ADC	immediate to register/memory
SUB	register/memory with register to either
SUB	immediate to register/memory
INC	register/memory
DEC	register/memory
SBB	register/memory with register to either
SBB	immediate to register/memory
CMP	register/memory with register
CMP	register with register/memory
MUL	register/memory with AL or AX to AX or DX:AX
IMUL	register/memory with AL or AX to AX or DX:AX
IMUL	immediate with register/memory to any register
DIV	AX or DX:AX by register/memory to AX or DX:AX to register/memory
IDIV	AX or DX:AX by register/memory to AX or DX:AX to register/memory
SHIFT/ROTATE	register/memory by count
SHIFT/ROTATE	register/memory by 1 (short form)
SHIFT/ROTATE	Register/memory by count in CL (short form)
AND	register/memory and register to either
AND	immediate to register/memory
AND	immediate to AL or AX (short form)

Table 9: Symmetrical iAPX 286 Instructions, (continued)

TEST	register/memory and register
TEST	immediate and register/memory
TEST	immediate and AL or AX ( <u>short form</u> )
OR	register/memory and register to either
OR	immediate to register/memory
OR	immediate to AL or AX ( <u>short form</u> )
XOR	register/memory and register to either
XOR	immediate to register/memory
XOR	immediate to AL or AX ( <u>short form</u> )
NOT	register/memory and register to either
NOT	immediate to register/memory
NOT	immediate to AL or AX ( <u>short form</u> )
JMP	indirect within segment, destination = (register/memory)
JMP	indirect intersegment, destination = (register/memory)

Even among the symmetrical instructions, short encoding forms of the instructions often exist when use of the AL or AX is specified. The short encoding is chosen automatically by the Intel macro assembler or high level language compiler.

The iAPX 286 is designed to execute programs with memory resident data very fast as well. The memory bus of the 80286 (mentioned page 11 of this document) is much more efficient in using memory access time than that of the 68000. The efficient 80286 bus interface complements its fast execution unit.

The performance of Intel microprocessors has consistently exceeded that of Motorola's. The 8MHz and 10MHz iAPX 86 were available before or at the same time as the 6MHz and 8MHz 68000 and exceeded their performance. The 8087 and 8089 added more system-level performance than the 68000 has achieved so far. The iAPX 286 provides twice the performance of the 68000 in protected applications. The iAPX 286 with 8207, 80287, and 82258 will provide more system-level performance than future 68020 with its minuscule cache, TTL-based memory management, floating point, custom memory control, and DMA support.

Without the same product line breadth, and development support, a Motorola-based design will be later into production and have less system-level performance than a design based on Intel products.

The reliability of Intel products is well proven. Our customers demand and get high reliability in Intel products.

The future of Intel products is brighter than Motorola's. Intel products are based on widely accepted standards. To take advantage of new technology in

building large, complex, systems we must use standards for software, computation, and interconnect. Standards allow future technology to work with and enhance today's technology.

Intel has the world-standard 16-bit architecture with the largest software base. Intel supports standards for arithmetic (IEEE P754 floating point standard with 8087/80287) and interconnect (IEEE 796 Multibus-I standard with 8288/8289/82288/82289) in hardware. Motorola uses either separate standards, or provides less support.

## Motorola Section: THE BEGINNING OF A FAMILY

---

Subsections: Family Members at Present, Extensions

Motorola statements:

The MC68000 is positioned as "the beginning of a family." A pro forma statement is made that "space is available to allow additional enhancements." "Because of this foresight," (which Motorola leaves unspecified) a variety of "orientations" are available.

Intel analysis:

Partly because little is to be made of a flat architecture or of architectural growth that can be based upon it, Motorola elevates data bus adaptations such as the 68008 to major status. The future 68020, future competitor of the 80286, is not a present member of the family, although Motorola discusses it here. It is estimated to have an automatic coprocessor interface as the 8086 has supported since 1978.

The Motorola section on 68000 extensions is vacuous! It must be pointed out that Intel is continuing to lead the competition into new architectural capabilities. Intel introduced the 8088, first 8-bit bus version of a 16-bit microprocessor 2 years before the 68008. Expansion of the iAPX 86 family into cost-sensitive high-integration markets with the 80186, containing peripherals with the processor, has nothing comparable from Motorola. Motorola offers no high-performance peripherals like the 82586 Ethernet Controller or 82730 Text Coprocessor, which provide many opportunities for end-product differentiation.

The integrated memory management, multitasking, and protection capabilities of the 80286 are not being addressed in the Motorola product line. Those Intel innovations, while integrated on-chip, are flexible since

- 1) The memory management technique is based upon standard descriptor concepts and provides for segment usage profiling, data sharing, aliasing and synchronization.
- 2) task management need not be used if the user desires all programs (operating system and applications) to execute as a single task,
- 3) the hierarchical protection model may be used to implement only a two-level user/supervisor model, or
- 4) the protection model may be dispensed with simply by building the system with all segments at the same privilege level.

In all cases the computational horsepower of the 80286 is being applied to the application.

At this point, Motorola is 2 to 3 years behind Intel in architectural development.

- 1) Most customers find that Intel product families are the earliest ones to meet application requirements. The functional partitioning of the instruction set into coprocessors is an earlier example, the supermicro capabilities (memory management, multitasking and protection) and new efficient bus of the 80286 are the latest.

## Motorola section: AN ARCHITECTURAL CONTRAST

---

Subsections: General Features, Register Set Alternatives

Motorola statements:

The 68000 is said to glitter, but its general features aren't discussed. Then what begins as a discussion on register set alternatives never really discusses any. Instead, the iAPX 286 multiply and divide instructions are discussed in detail since they use the AX register for one operand and the result. Motorola makes the incorrect statement that another register must be loaded to hold the other operand of unsigned multiply or divide; in fact the operand may be in memory and accessed by any iAPX 286 addressing mode.

Intel analysis:

On the topic of general issues, the first thing to point out is that the performance of the iAPX 286 is comparable to the future future 68020, not the 8086, 68000 or 68010.

Also the iAPX 286 provides superior memory management flexibility, superior protection and multi-task capability. The 16Mbyte address space of the 68010 may only support 32 separate address spaces versus 8000 on the iAPX 286. Since half the 68451 descriptors represent the system state only 16 descriptors are left for use by application programs. This allows a maximum of 16 tasks to exist at any one time if and only if each task consumes only one segment. This does not provide any intra-task protection let alone inter-task protection.

The segments defined by the Motorola 68451 MMU must begin on a physical address that is a power of two. This mandates an inflexible system.

At least 26 iAPX instruction types use the registers generally, as covered earlier. They were listed in Table 9 of this document. The register set of the 80286 is general purpose for many operations. Instructions using the eight 80286 registers in general purpose, contain an addressing-mode byte called the mode register/memory byte. This byte specifies the effective address calculation mode. Included among the effective address choices are the eight general registers themselves. Only for some arithmetic and addressing operations are the 8086 registers characterized to improve the encoding density of instructions.

The working register set of the iAPX 286 consists of eight 16-bit registers of which four can have uses as 16-bit base or pointer registers. There are also four 16-bit user-addressable segment registers that point into the memory-resident descriptor tables containing the base, limit, type, and access control that is cached into the 48-bit cache for each. In addition to an instruction pointer and a flag register, is the machine status word which provides control for emulation of the coprocessor and other functions. The task register for the currently-executing task, and three descriptor table registers (one each for the GDT, IDT, and LDT) complete the register set.

For encoding density, the best tradeoff is to have sufficient registers for efficient compiler use but no more than necessary. The Lunde study, shown on pp. 21-22 reveals the average number of live registers required for a variety of functions is less than four. Studies like these show the registers on the iAPX 286, described above are for maximum performance of optimizing compilers.

Thus, the iAPX 286 eight-register working set does provide adequate work space and is still efficiently encoded within the instruction set.

The register set of the 8086 reflects the orientation towards high level languages. Assembly language programmers also use the registers in a similar manner. The eight registers allow an efficient implementation and very dense encoding. Most iAPX 86 instructions will be 2 or 3 bytes in length. Only direct addressing within a segment or 16-bit literals extend instructions beyond 3 bytes.

The 68000 instructions must be multiples of 2 bytes in length. Most memory reference instructions are 4 bytes or 6 bytes in length when the stack displacement or absolute address is included. Therefore, 68000 encoding is not as dense as that of the 80286.

The 80286 instruction encoding density allows for fast program execution since less memory bandwidth is tied up fetching the instructions. Until recently, memory systems have been slow, requiring several wait states to use them. The wait states slowed down each instruction fetch. Programs with longer instructions require more instruction fetches and more wait states to execute with slow memories.

The 68000 does provide more registers than the 8086. Yet only rarely can compilers can use more registers than those provided in the iAPX 86. The characterization of register for certain operations such as string operations rarely is a hindrance since the instructions for general purpose operations use any register of the set, as shown on pp. 21-23 of this document. Characterizing the multiplication and division operations for the AX register has helped accelerate the performance of those instructions, where performance is often critical.

Table 10: Clock Counts for Signed Multiplication and Division

(16-bit operation, one operand in memory)

<u>Processor</u>	<u>Multiplication</u>	<u>Division</u>
iAPX 286	24	28
68000	70 to 82 *	158 to 170 *
68010	42 to 54 *	122 to 134 *

\* exact value depends on 68000 or 68010 addressing method used.

The iAPX 286 has fantastic computational performance! None of these processors offers 32-bit multiplication or division with a single instruction. Not surprisingly though, the iAPX 286 is also faster than competitive offerings for 32-bit arithmetic as shown on pp. 18-20 of this document.

## Subsection: Register Scheme

### Motorola statements:

Motorola stands corrected when claiming the 80286 register presents a problem. In fact, the register set and CPU are extremely capable. The eight general-purpose iAPX 286 register are displayed but erroneous statements have been made by Motorola.

### Intel analysis:

Firstoff, the Motorola claim that no addressing mode to the iAPX 286 stack pointer or the stack is incorrect! Another correction: the EDN benchmark in Motorola Appendix C contains only 28 lines of code, not 41 as claimed in the Mot paper!

Most computers have exceptional cases with which the assembly language programmer or compiler writer must be concerned. Motorola fails to mention the special cases which also bother the writer of a compiler for the 68000. All 68000 compiler writers must watch for the special cases like: not using address registers for index arithmetic, no CLR A0 instruction, CLR mem first reads the memory location before writing zero, not having 32-bit offsets for indexed addressing, and limitations on addressing modes like no PEA (A0)+ or JSR (A0)+.

The mainly general-purpose availability of the iAPX 286 register set was described on pp. 21-23 of this document. When memory is referenced, the iAPX 86 family establishes segment register usage rules that closely match the way memory addressing is performed. Reference the table below for the implied segment register usage such as, "the Code Segment register is referenced automatically when the iAPX 286 fetches instructions."

Table 11: iAPX 86/286 Segment Register Usage

Segment Register Selection Rules		
Memory Reference Needed	Segment Register Used	Implicit Segment Selection Rule
Instructions	Code (CS)	Automatic with instruction prefetch
Stack	Stack (SS)	All stack pushes and pops. Any memory reference which uses BP as a base register.
Local Data	Data (DS)	All data references except when relative to stack or string destination
External (Global) Data	Extra (ES)	Alternate data segment and destination of string operation

For special cases requiring segment overrides, iAPX 286 development tools will automatically insert the required segment overrides. Motorola fails to recognize the advantages of a strongly typed assembler which recognizes in which segment an operand is located. The assembler is informed of operand locations and segment register contents via declarations similar to those used in the IBM 360/370 assembler.



Finally, Motorola claims that register usage in the iAPX 86 EDN Benchmark E, called string search, required 41 lines of code. Only 28 lines are listed on pages 37-38 of their document! The 68000 required 18 lines of source. The iAPX 86 required multiple push/pop instructions to save/restore the registers while the 68000 used the multiregister load/store instruction. Motorola apparently forgot to mention that the 68000 required the same number of code bytes as the iAPX 286, since the Motorola encoding format is less dense than the 80286!

## Subsection: Virtual Memory, iAPX 286's Virtual Claims

### Motorola statements:

Motorola, on their page 5 present four attributes of virtual memory, yet only attribute 4 is relevant to virtual memory systems:

"4. When a memory access faults, the operating system can completely recover and continue the program."

Item 1 on the list, segment size, has not a thing to do with virtual memory.

Neither does item 2, protection mechanisms, for protection mechanisms exist separately from memory management. In the MC68010, which lacks hardware protection barriers between address spaces, Motorola appears to imply that having separate descriptors in the MMU somehow prevents undesired access to the various regions. In fact they don't.

Item 3 of the list, demand paging, is not required of memory management systems. Demand paging is only one possible implementation of memory management.

Given what appears a very confused Motorola understanding of virtual memory, Motorola continues to make the absolutely incorrect statement that the iAPX 286 does not support virtual memory. In fact, the iAPX 286 totally supports virtual memory.

Finally Motorola seems to misunderstand bus faults issues, on its page 7.

### Intel analysis:

In brief, Intel's iAPX 286 supports virtual memory more effectively than the MC68010. In the document, Motorola twists the meaning of virtual memory to exclude the iAPX 286. Simply changing the term "demand paging" to "demand segmentation" refers to the iAPX 286 implementation of virtual memory.

Without doubt, the iAPX 286 is a virtual memory machine; a program can run on the iAPX 286 without direct knowledge of whether a segment is in memory or not. iAPX 86 development tools will automatically split large data arrays into multiple segments which may be independently located anywhere in memory. The programmer need not be concerned with how the processor indicates segments which have been moved to secondary storage. The iAPX 286 supports demand segmentation in the same way other processors support demand paging. The iAPX 286 provides all the hooks to allow recovery from not-present exceptions.

The 68000 can not support virtual memory; some of its addressing modes are not restartable. The 68010 added 30% to the 68000 die size to fix all the instruction restart problems of the 68000. Any use of virtual memory requires the 68010.

Motorola presents the iAPX 86 segment size as a major problem in virtual memory systems. Segments need only be as large as the data or code contained therein, so only as much information is loaded into memory as is generally needed. Segments help virtual memory systems immensely by characterizing the address space. The working set of a program is clearly defined by the 2-4

segments identified by the segment registers. The linear address space of the 68000 gives no hints to the OS about what the current working set is; the OS must guess.

The protected mode 8MHz iAPX 286 executes the EDN I Benchmark, called quicksort, in 20.5ms as shown in the iAPX 186, 286 Benchmark Report, pg. 7 (order number 210826), not the 30.1ms Motorola claims. The iAPX 86 segment size requires the quicksort program to look for end conditions in segments. It adds code size, but not any appreciable execution time in comparison to a quicksort within a single segment.

The iAPX 286 protected mode implementation of quicksort allows some very useful structuring of the sort data. The data to be sorted can be split into several variable sized segments. This allows easy insertion and deletion of records since only a small part of the array need be moved to do so. Insertion of new records is easier because the new memory area for the new record need not be contiguous with the old memory area.

The discussion of loading segment registers on the iAPX 286 loses sight of how they work and of locality concepts. After loading the segment register, it is used by many instructions that reference data in the segment. The overhead of loading the register and fetching descriptor information is spread across many instructions.

Protecting a 68000/68010 penalizes every memory cycle. The 68451 MMU requires two wait states be added to every bus cycle to check and translate the address and allow realistic total access time of about 250ns. The iAPX 286 has no address translation penalty in accessing memory for code or data cycles.

Demand segmentation on the iAPX 286 provides all characteristics of virtual memory. The iAPX 286 automatically examines each segment descriptor used to be sure the segment is in memory. The segment descriptor check occurs when the program loads the segment descriptor. The compiler or assembly language programmer generates the segment load instruction whenever a new segment is required by a program.

#### Analysis focus: Bus error misconceptions

The Motorola discussion of bus errors misses the point. The writers of the Motorola document do not appear to understand the requirements of fault-tolerant systems. Repeatable hardware errors require redundant hardware to recover from them. Software cannot copy bad memory to good memory or somehow hope that a hard bus error means that software will fix it. The Intel iAPX 432 was carefully designed to allow fault-tolerant systems which can gracefully recover from hard errors.

The iAPX 286 easily supports retry of transient bus errors. Bus cycle retry is easily performed by the 82288 bus controller. When a bus error is detected, the hardware simply disables the bus controller via CENL/AEN#, after waiting for the bus to settle, it reenables the 82288 via CENL/AEN#. The 82288 will reissue the bus command to restart the bus cycle. If the bus transfer fails again, a hard bus error has been detected.

Attempting to restart the program at a later time doesn't make sense since no software can make the hard error go away. Restarting the program requires redundant hardware resources like duplicate busses and memory controllers. For systems which must be fault tolerant, the iAPX 432 microprocessor provides fault tolerance support in the CPU and support components. The bus cycle retry of the 68010 is not adequate for fault tolerant systems.

The claim for write hard-fail recovery is far fetched. They had to make it a write hard-fail since a read hard-fail would make copying the data useless since the data will be read wrong. How can a write hard-fail allow correct reading of data? If the write was a partial byte write and the other unchanged byte had bad parity or bad ECC then it's still bad and copying bad data won't help.

The read hard-fail case is also far fetched. The only possible case where the correct value of the location can be determined is when the location corresponds to the code area of a program. This is only one possibility in many cases where a hard read failure can not be recovered if no redundant memory copy exists. This won't work if the read is from program-modified data. Refreshing the program doesn't solve the problem of recovering from hard memory failures without redundant memory copies.

Subsections: True Virtual Memory Facilities

More Advantages with Motorola's True Virtual Memory

---

Motorola statements:

The apparent underlying message of Motorola's titles are again false. In fact the iAPX 286 completely supports virtual memory. Unfortunately for the credibility of the Motorola paper, its "bottom line," which essentially repeats the implicit message of the title is just as false.

Intel analysis:

Bus cycle retry is confused with virtual I/O in the 68010. Since the 68000 does not have I/O instructions, memory mapped I/O is required to perform I/O. Performing virtual I/O on the 68000 requires aborting memory accesses to special locations identified as memory mapped I/O area.

The iAPX 286 uses a much simpler approach to memory mapped I/O. Special I/O instructions are provided which can be set up to cause a trap by applications and operating system code if they attempt to perform I/O. The trap is fully restartable and does not issue any I/O bus cycle. The trap handler can easily simulate the I/O operation and restart the program.

If iAPX 286 programs want to use memory mapped I/O yet still protect the system from undesirable I/O operations, the operating system should allocate one segment to the I/O area. The segment can be write-protected to detect attempted writes. Reads from the segment can simply read status information placed there by the OS. The write-protect fault is restartable to allow emulation of the write operation by the O.S. Another possibility is to mark the I/O segment on the iAPX 286 "not present" so that both I/O reads and I/O writes are trapped to the operating system. This approach is also fully restartable.

Shared memory is implemented by the iAPX 286 much more efficiently than the 68451. The 68451 requires segments to be a power of two in size with a minimum segment size of 256 bytes. For a program to support odd-sized shared memory areas requires multiple segments in the 68451.

In contrast the iAPX 286 allows variable sized segments from 1 to 65536 bytes. The shared memory area can be flagged with the not-present bit traps to an O.S. service when a program attempts to access shared memory.

## Subsection: Data Types and Operations

### Motorola statement:

The Motorola discussion does not differentiate between types recognized by hardware and types recognized by software. This section therefore becomes somewhat ambiguous.

### Intel analysis:

As this topic has been mentioned earlier in the Motorola document, full coverage of data types should be referred to pp. 16-20 of this document. The iAPX 86 family supports more datatypes than the 68000. The Motorola datatype discussion ignores the iAPX 86 floating point data types supported by the 8087 and 80287 and system types of the iAPX 286. The discussion also confuses datatypes supported by single instructions and those requiring multiple instructions. No mention is made of the iAPX 86 BCD multiply and divide operations which have no equivalent on the 68000.

Supermicro datatypes for operating system support of tasks, segment attributes, and gates is also ignored by the Motorola discussion. All of these very important operating system datatypes must be simulated with many instructions in a 68000 systems.

For example the task switch operation of the iAPX 286, based on the task datatype. The ability of the iAPX 286 CPU to perform task switches using a simple CALL or JMP instruction when desired (or through an INTR when appropriate) while maintaining a back link task pointer in memory permits single-threaded multitasking to be performed with a simple scheduler or kernel. A complex operating system design need not be undertaken.

The discussion of 32-bit integers ignores the basic speed advantage of the iAPX 286. The 32-bit add, subtract, multiply, and divide operations of the iAPX 286 are faster than those of the 68000 as shown pp. 14-18 of this paper.

## Subsection: Bit Manipulation

### Motorola statements:

Motorola claims easier bit manipulation than the iAPX 286. However the immediate forms of the iAPX 286 boolean instructions provide very easy bit test, set, reset and invert operations, and with any iAPX 286 addressing mode.

### Intel analysis:

Primarily, the supermicro system capabilities of the iAPX 286 automatically perform more bit checking on-the-fly than the MC68010 could efficiently perform for privilege-level checking, not-present checking and the like. These are checked automatically so that operating system code need not inspect many flags.

When needed, the iAPX 86 family easily performs all bit manipulation operations. Immediate forms of the iAPX 86 boolean instructions make the setting, resetting, complementing and testing of bits very easy, and any addressing mode can be used to reach the effective address. The added function of 68000 bit-manipulation instructions is not much. The Motorola set of operations is limited to single bits within a single 8-bit word of memory or 32-bit data register.

Both processors require extra instructions to convert a general bit address to a 8-bit memory byte address then to a bit in the word. The iAPX 86 requires two extra instructions to identify the bit in the word.

Table 12: 68000 Bit-Test Time of Bit Array in Memory at A0

12	MOV	index,D0	; Get bit index
4	MOV	D0,D1	; Save low bit address
12	LSR	#3,D0	; Form byte address
14	BTST.B	D1,(A0+D0)	; Test bit in memory
42 clocks + 8/wait			

4 usec	unprotected at 12.5MHz 1 wait
5.8 usec	protected at 10MHz 2 wait

Table 13: iAPX 286 Bit-Test Time of Bit Array in Memory at DS:SI

5	mov	bx,index	; Get bit index
3	mov	cx,107H	; Form byte address
2	and	cl,bl	; Mask out bit address
8	shr	bx,3	; Form byte address
5-12	shl	ch,cl	; Move mask bit into position
5	test	ch,[bx+si]	; Test bit
28-35 clocks			

3.5 - 4.4 usec at 8MHz 0-wait

## Subsection: Register and Addressing Mode Flexibility

### Motorola statement:

The iAPX 286 only offers 2394 ways to do an effective address calculation for the ADD instruction.

### Intel analysis:

For shame! The 68000 provides more addressing modes than the iAPX 286. The problem is that they are used so infrequently but cost alot. Studies on VAX 11/780<sup>1</sup> show that other than register direct, register indirect, direct address, stack operations, literals, and register indexed, any other address mode is used an order of magnitude less. This means that each 68000 instruction is larger than necessary to allow the specification of addressing modes that are not used.

Motorola has again failed to present a complete enough picture to show the cost of these modes. 68000 and 68010 effective address calculation cost up to twelve clocks of TIME. Even the simpler ones cost, and the simple Absolute Long is the worst case. The effective address calculation time is additive to the following groups of instructions:

- o Standard Instructions (e.g. arithmetic and logicals)
- o Immediate Instructions
- o Single Operand (Unary) Instructions
- o Shift/Rotate Instructions
- o Bit Manipulation Instructions

Details on 68000 and 68010 effective address calculation times are listed in their respective data sheets.

By contrast, effective address calculation of the iAPX 286 does not cost any clocks, as the calculation proceeds in parallel with other functions. This is true even when the effective address has four components: segment base, base register (such as BX or BP), index register (such as SI or DI), and immediate displacement. This is also true when one or both of the index registers are incremented/decremented following the instruction (as with string operations).

The address unit of the iAPX 286, an efficient machine in itself, performs effective address calculations and limit checking in parallel with functions of the instruction unit, execution unit, and bus unit. The iAPX 286 puts its 130,000 transistor budget TO WORK!

<sup>1</sup> Cheryl A. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," in Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems, ACM order number 556811, (March 1982), pp. 177-184



## Subsection: iAPX 286 Just Doesn't Stack Up

### Motorola statements:

According to Motorola here: Intel, Zilog, and National have all forgotten to implement an "extremely important" addressing mode. On this issue, as in the byte-ordering convention, Motorola finds itself across the fence from all these companies.

### Intel analysis:

Real clever pun in the title! Seriously, though, would it be faster to use the Motorola Address Register Indirect with Postincrement, or use an index register of the iAPX 286 and then increment the index register twice? The Motorola method takes one line of code and 8 clocks. The Intel method requires three lines of code and only 7 clocks. That stacks up real well.

The 68000 postincrement and predecrement addressing modes are used mainly for stack pushes and pops. The VAX study clearly shows the limited use of these modes, called "extremely important" by Motorola. Here's what the 1982 VAX study showed:

Table 14: Compiler Addressing Mode Frequency (percent)

<u>Addressing mode</u>	<u>BASIC</u>	<u>BLISS</u>	<u>COBOL</u>	<u>FORTRAN</u>	<u>PASCAL</u>	<u>PL/1</u>
Register	34.9	38.1	37.9	41.0	45.1	42.2
Literal	9.8	18.5	18.1	18.2	15.5	10.9
Byte Displacement	16.3	16.7	10.8	13.6	16.6	11.2
Register Deferred	9.0	8.6	15.3	9.1	2.0	2.5
Autoincrement	17.4	3.3	4.8	4.1	2.6	4.4
Index	4.7	3.0	2.1	5.6	7.6	8.8
Long Displacement	6.1	6.5	7.3	5.5	3.0	3.5
Word Displacement	.8	2.0	.9	1.5	6.5	13.9
Byte Displ. Defer.	.5	1.5	1.3	.6	.4	2.1
<u>PREDECREMENT</u>	.5	1.0	1.0	.5	.7	.3
<u>POSTINCREMENT</u>	.1	.7	.2	.3	.0	.0
Long Displ. Defer.	-	.1	.3	.0	-	.1
Word Displ. Defer.	-	-	-	.0	-	.1

Based on the VAX study, it appears some people at Motorola are still learning what is "extremely important," and are not afraid to publish their inaccuracies in the meantime. At Intel the approach is get the facts straight first, then work on little puns.

The iAPX 86/286 has all the stack operations needed by programs. It's clear that the discussion of data scanning by Motorola is an infrequent application in comparison to all the simple data movement performed by programs. It is faster not to have these modes and use separate increment or decrement instructions for the rare cases that change pointers upon use. Furthermore, the increment/decrement instructions are as easily used for memory-resident pointers as for register-resident ones.

The table from the VAX study is also relevant to the 64K segment size issue. It indicates that displacements greater than 16 bits, called long displacements, occur an "average" of only 5.3% of the time, even on a large machine, the VAX. This supports Intel's own experience with the iAPX 86 family as well. By far, most of the time the 64K segment size is not an issue. Nevertheless, the iAPX 386 will handle segments up to 4 Mbyte in size. With the iAPX 386, that issue will be cleared away entirely with capability of truly HUGE segments.

## Subsection: Indexed Addressing

---

### Motorola statements:

Motorola Figure 8 is presented in the vicinity of this section and has some glaring errors. Motorola claims an order of magnitude (10x presumably) speed advantage in accessing data structures larger than 64Kbytes, which is incorrect as shown pp. 64.

### Intel analysis:

Corrections to Motorola's misrepresentation of the iAPX 286 in its Figure 8 are required.

- 1) The iAPX 286 does support predecrement and postincrement directly on the stack pointer register. It also does support autoincrement and autodecrement of the index registers for all string operations. For ~~other~~ other operations needing incrementing or decrementing, it has been shown on pg 45 that incrementing the pointer register on the iAPX 286 is faster than using the Predecrement or Postincrement modes of the 68000/10 regardless.
- 2) The iAPX 286 certainly does supports IP relative addressing. All conditional and unconditional jumps within a code segment are IP-relative as a matter of fact. The iAPX 286 Programmer's Reference Manual clearly describes the JMP operand as a "relative offset added modulo 65536 to the offset of the instruction that follows the JMP. The result is the new value of IP. A byte offset is sign-extended before it is added." Code segments may also contain data at addresses which are CS-relative. The iAPX 286 does not require any IP-relative addressing modes of the type used by the 68000. The segmentation of the iAPX 86 provides an advantage by eliminating the need for such an addressing mode. This is no deficiency of the iAPX 286 architecture. Programs simply use the segment offset directly to address anything in the segment. The offset within a segment is always independent of the segment's base location.

Motorola fails to give Intel credit for more dense instructions with 8-bit displacement encoded as a single byte rather than the 2 bytes required on the 68000. The iAPX 86 also allows double indexing without any extra code size increase.

The register choice in the iAPX 86 matches the usage of registers by high-level languages. The BP register is for stack addressing, BX for base address of data structures, and SI/DI for indexing within the structure.

The iAPX 86 is much more efficient in accessing local stack-based variables than the 68000. Studies show most modules require fewer than 128 bytes of stack space. The iAPX 86/286 allow an 8-bit offset be used to reference such values. The 68000 requires a 16-bit offset for all such references. 68000 stack reference instructions will almost always be 1 byte longer than iAPX 86 stack reference instructions.

## Subsection: iAPX 286 Does Not Offer Absolute Addressing

### Motorola statements:

In the heading, Motorola falsely states that the iAPX 286 doesn't offer absolute addressing. That is a false statement. Motorola never attempts to prove its false statement within the paragraph which followed. Motorola presents a misleading string move example in this section, which is more accurately described in the following analysis.

### Intel analysis:

The numerous false statements are becoming routine at this point of the Motorola paper. Refute false statements we will.

In fact, the iAPX 286 provides direct addressing to any address in the logical address space. It provides such with the type "Pointer" shown here on page 18. The iAPX 86 provides more efficient absolute addressing than the 68000. The addressing on the iAPX 86 was designed to match the way programs need addressing.

The common usage of absolute addressing is to access static data structures or I/O devices. Absolute addressing is called direct addressing on the iAPX 86. Direct addressing on the iAPX 86 provides very efficient access to local static variables. The iAPX 86 segmented structure and development tools place local variables for each routine in the same segment. A 16-bit offset, relative to DS, can access all local static variables for a single compilation module and is sufficient a very high percentage of the time, as supported by our experience and for example the VAX study, to quote additional external data. In the rare cases that local static variables exceed 64K, the compilers will allocate multiple data segments.

The DS segment register is loaded on entry to the subroutine by the standard subroutine entry prologue code to gain addressability to the local static variables for that module. Within that module, each reference to the variable requires a 16-bit offset in the instruction. The shorter local static variable reference instructions will execute faster than the 68000 that requires 32-bit addresses.

Another use of absolute addressing is for memory mapped I/O. The iAPX 86 I/O instructions provide very efficient I/O access without the need for 32-bit absolute addresses. For memory mapped I/O, the LDS, LES instructions can be used for gaining access to the I/O device address. The iAPX 286 will automatically check access to such an I/O device with the LDS, LES instructions. Subsequent accesses to the I/O device using the base address will run at the full speed of the iAPX 286 without any protection checks.

### Analysis focus: Misleading Move Comparisons

The comparison of a move from memory-to-memory on page 12 of the Motorola document is completely wrong. The iAPX 286 instruction clock counts are wrong and memory access times are completely wrong.

The context of the move must be correctly understood as well. A move from one random external address to another random external address is not a frequent operation. Programs access local variables on the stack or in static areas much more frequently than randomly-placed external variables. The iAPX 286 has much better access to local stack-based and static variables than the 68000 due to its allowing 8-bit offsets, adequate for up to 256 bytes of data.

For the relatively rare times that access to random external data is required, the following is the correct comparison.

#### 68000 Move Memory-to-Memory Unprotected 12.5MHz 1-wait

MOVE.W here,there	35 clocks
	2.8 usec

#### iAPX 286 Move Memory-to-Memory Unprotected 8MHz 0-wait

LDS SI,hereptr	7 clocks	
LDS DI,thereptr	7 clocks	
MOVSW	5 clocks	
	<u>19</u> clocks	2.375 usec

Note that the protected and unprotected cases are correctly compared.

#### 68000 Move Memory-to-Memory Virtual Memory 10MHz 2-wait

MOVE.W here,there	42 clocks	4.2 usec
-------------------	-----------	----------

#### iAPX 286 Move Memory-to-Memory Protected, virtual memory 8MHz 0-wait

LDS SI,hereptr	21 clocks	
LES DI,thereptr	21 clocks	
MOVSW	5 clocks	
	<u>47</u> clocks	5.9 usec

### Analysis focus: Misleading Move Comparisons

The iAPX 286 in protected mode is slower in setting up the addressing because we are permitting Motorola to continue its assumption that the 68451 MMU happens to contain the descriptors for the random locations HERE and THERE! But nevertheless, once the iAPX 286 memory protection overhead is taken during the LDS and LES instructions (while the CPU automatically loads its descriptor caches for DS and ES), subsequent accesses are much faster. For example, the comparison for a 32-bit move is shown below. Each extra 16-bit transfer costs only 5 more clocks for the iAPX 286 but costs 12-10 more clocks for 68000, depending on whether the 68451 MMU (or other memory management unit) is used for virtual memory or not.

	Unprotected 8MHz iAPX 286 0-wait	Unprotected 12.5MHz 68000 1-wait
32-bit move	3.0 usec	3.6 usec
	Protected, Virtual mem 8MHz iAPX 286 0-wait	Virtual memory 10MHz 68000 2-wait
32-bit move	6.5 usec	5.2 usec

The future 68020 clock counts shown on page 12 do not include all the times required to execute the instruction. Just two 4-clock bus cycles, one to read the word and one to write it, are shown in the clock count. Extra bus cycles are required to fetch the instruction. Each bus cycle will have wait states for MMU operation and to allow use of reasonable speed memories.

The future 68020 move instruction is 10 bytes long and requires three 32-bit bus cycles to fetch. Motorola assumes the instruction is already in the instruction buffer. This assumption will not be true most of the time. The inner loop of programs will not reasonably contain a move operation to a fixed address as shown. The future 68020 clock counts should show the count when the instruction is not in the code buffer and also show the size and hit-rate for the instruction buffer.

## Subsection: Program Counter Variances

---

### Motorola statements:

Motorola falsely states that "the iAPX 286 expressly forbids the moving of ROMS between systems" and "segment ID's have meaning only to one individual operating system on just a single hardware system." Both statements are false.

### Intel analysis:

Motorola is completely wrong regarding iAPX 286 ROM-based code. The iAPX 286 MMU allows a segment to be located anywhere in physical memory. The numeric value of a segment ID (called selector in Intel literature) is unrelated to the physical address of the segment. An iAPX 286 program in ROM can contain segment selectors and be able to execute at any physical address range.

The iAPX 86 does not require any program-counter-relative addressing modes of the type used by the 68000. The segmentation of the iAPX 86 provides an advantage by eliminating the need for such an addressing mode. This is no deficiency of the iAPX 286 architecture. Programs simply use the segment offset directly to address anything in the segment. The offset within a segment is always independent of the segment's base location.

## Subsection: Instruction Set

### Motorola statements:

Motorola positions its instruction set as completely new based upon unspecified "in-depth studies." The iAPX 286 is presumably just "an 8086 with a memory manager on-chip."

### Intel analysis:

The iAPX 286 instruction set includes more functions than that of the 68000. A portion of the iAPX 286 is also functionally partitioned into the numerics coprocessor, the 80287. The iAPX 286 supports more data types in more ways than the 68000. The 68000/68010/future 68020 have nothing comparable to the task, gate, protection control operations of the iAPX 286. These operations and floating point instructions are what system applications need to run fast.

The 68000 instructions are not as orthrogonal as Motorola claims. For example, the multibit shift instructions cannot apply to memory-based operands. The logical test instructions cannot apply to address registers. The CLR instruction cannot apply to an address register. The iAPX 86 has no such restrictions.

Motorola figure 9 exaggerates the capabilities of the 68000 LINK instruction. It does not compare to the more capable ENTER instruction of the iAPX 286. On the 286 ENTER creates an entire stack frame dynamically by allocating the appropriate amount of memory, complete with back links to previous lexical nesting levels. The 68000 LINK instruction is but a small part of that capability.



## Subsection: Stack Operation

---

### Motorola statements:

Motorola makes another false charge on page 15 of its document that no addressing mode which address the stack. Motorola devotes another page to the predecrement and postincrement addressing modes and attempts to create an iAPX 286 deficiency where none exists.

### Intel analysis:

Motorola states on its page 15 that the iAPX 286 has no addressing modes that address the stack. This is incorrect. The BP-relative addressing mode of the iAPX 86 automatically references the stack segment. This addressing mode is very frequently used and is much better than the 68000 method of addressing the stack. Most iAPX 286 stack reference instructions require only 3 bytes. The 68000 requires 4 bytes for most stack reference instructions.

The rare usefulness of the predecrement and postincrement addressing modes and the method of handling pointers for data structures such as stacks have been thoroughly discussed, with supporting research on pp. 43-44 of this document.

As we know from the VAX study, such addressing modes have frequency of 1% or less, and a repeat discussion is not worthwhile.

## Subsection: String Manipulation

---

### Motorola statement:

Based on the false statement Motorola made on page 11 of its document, that the iAPX286 contains no autoincrement/autodecrement capability, the 68000 string instructions are incorrectly positioned as a competitive advantage over the iAPX 286.

### Intel analysis:

The iAPX 86 has string instructions built into the instruction set for faster string support than that of the 68000. The iAPX 86 string move instructions support the real needs for high-performance string operations that devote the entire databus bandwidth to data transfer (i.e. opcode fetching is unnecessary while the string operation executes). The other less-frequent string operations are easily built up from the primitives and the LOOP instruction.

The protected, virtual memory iAPX 286 move-string instruction MOVS runs at the full bus bandwidth of the iAPX 286, 4 million bytes/second moved from the source to the destination, faster than many dedicated DMA controllers! The virtual memory 10MHz 68000 can move data at the rate of only 1 million bytes/second (40 clocks to execute MOV.L (A0)+, (A1)+ DBRA D0, loop at two wait states per memory reference). The 68010 at 10MHz improves this rate to only 1.33 million bytes/second.

## Subsection: Branching

---

### Motorola statements:

Motorola correctly states that conditional branching is an all-important concept. Motorola incorrectly states that the iAPX 286 allows conditional destinations to be only within +127/-128 bytes of the conditional instruction.

### Intel analysis:

Motorola is confused about how compilers generate code for jump instructions which having short forms and long forms. In the iAPX 86, it is no more difficult than the 68000 to generate code for this case. The iAPX 86 conditional jumps constructs have a 2-byte form and a 5-byte form. The iAPX 86 five byte conditional jump uses the opposite-sense of the conditional jump to skip around a 3-byte unconditional jump to the destination. This is a completely valid construct. Similarly, the 68000 jump instructions have a 2-byte form and a 4-byte form. The compiler writer can either always generate the fixed length long form or else add the extra code to allow use of the smaller short form.

Table 15: Conditional branch execution times

	Cond Branch Taken	Not Taken
iAPX 286 2-byte form	$7 + m^*$	3
iAPX 286 5-byte form	$10 + m^*$	9
68000 2-byte form	$10 + 2/\text{wait-state}$	$8 + 1/\text{wait-state}$
68000 4-byte form	$10 + 2/\text{wait-state}$	$12 + 2/\text{wait-state}$
68010 2-byte form	$10 + 2/\text{wait-state}$	$6 + 1/\text{wait-state}$
68010 4-byte form	$10 + 2/\text{wait-state}$	$10 + 2/\text{wait-state}$

$m$  = Number of bytes of code in next instruction.

The 68010 does not show how much time is required to refill its instruction buffer after the jump although additional time is probably required.

The iAPX 86 two-byte jump instruction is more often usable than the two-byte 68000 jump instruction. Since iAPX 86 instructions are shorter, the -128 to +127 byte range of the two-byte jump instructions will cover more instructions than on a 68000. Covering more instructions allows the iAPX 86 two-byte form to be used more often than the 68000 can use the two-byte form.

The iAPX 86 full-range unconditional jump is either 3 bytes for jumps within a segment or 5 byte to jump out of a segment. This is one byte less than the 68000 unconditional jump instruction sizes.

### Analysis focus: Another Motorola Inaccuracy

Motorola is again wrong about position independent code on the iAPX 86. On page 17 Motorola claims JMPs allow only absolute addresses on the iAPX 286. The 16-bit displacements used by the iAPX 86 jump or call instructions are always relative to the program counter. The iAPX 286 inter-segment jump instructions use a full 32-bit logical address which is position independent.

Note that the 68000 does not have any position-independent jump or call instruction which uses a 32-bit offset. Either separate instructions must calculate the destination address from a 32-bit relative constant or else they must use an absolute destination (and not have position-independent code!).

## Subsection: Memory-to-Memory Arithmetic

### Motorola statements:

The memory-to-memory arithmetic discussion is confused since the 68000 has no memory-to-memory arithmetic operations for arrays. The code example used by Motorola on page 17 is wrong. The code shown by Motorola adds two very-large integers to each other. It does not add two arrays of individual numbers to each other.

### Intel analysis:

The Motorola memory-to-memory arithmetic discussion is confused. The 68000 has no memory-to-memory arithmetic operations for arrays. The 68000 requires such values first be loaded into a register, then the arithmetic performed on the source. The ADDX.L instruction of the 68000 is for integer operands larger than 32-bits or odd-sized operands like a 24-bit integer.

The 68000 code example used by Motorola on page 17 is wrong. The code shown by Motorola adds two very-large integers to each other. It does not add two arrays of numbers to each other. Adding the X bit (essentially the carry bit) in each operation prevents ADDX.L from being used in array-to-array additions. Besides, array additions usually require three operands, ruling out use of the primitive ADDX.L instruction. The loop instruction for the 286 requires 4 or 8 clocks depending upon if the loop is taken, not the 10 clocks as Motorola claimed.

All the Motorola processor configurations used in the results at the bottom of page 17 require faster memories than the iAPX 286! Therefore Motorola is incorrect that the iAPX 286 must encounter some slow-down to use "equal speed memories" is false, as explained on pages 8-9 of this document. The misleading nature of Motorola's story is exposed when one realizes that the "equal speed memories 200ns" would cause the Motorola system to fail, since the configurations they use only give 170ns and 165ns access time. The analysis Motorola presents is quite low-grade.

The power of the Intel 80287 is ignored in this example. With a single opcode, a packed BCD number may be loaded into 80-bit wide registers of the 80287, and 19 decimal digits may all be added with one opcode (FADD), eliminating the looping entirely. Furthermore, the 80287 FADD is performed to the IEEE 754 standard adopted by most mini and mainframe computers. The IEEE 754 standard provides for well-defined gradual underflow or overflow as well as a complete set of exception condition traps, all detected by the 80287 in hardware. The 68000 does not enjoy this capability or a similar device. The 68881 planned for the future 68020 will not operate as a coprocessor for the 68000 or 68010. The 68000 or 68010 will have to address the device as a slave peripheral.

## Subsection: The iAPX 286 Single Accumulator

---

### Motorola statements:

Motorola claims that having [rather infrequently-used] instructions such as multiplication, division, string operations, BCD and ASCII, use the accumulator can result in "massive congestion."

### Intel analysis:

Motorola misses the point of the iAPX 86 design in this section. The iAPX 86 register usage is optimized for the way programs are commonly written. Additionally, use of the AX register allows a concentration of iAPX 286 hardware within the execution unit, so programs using these instructions execute faster. Refer to the Table 10 on pg 30 of this document.

The instruction set provides short instruction forms for several operations using the AX. Most programs use a temporary value several times in the period of a few short instructions. If these temporary values are kept in the AX, then the short instruction forms can be used thereby shortening the program. The AX register is not required by most arithmetic or logical instructions.

Several other instructions, like multiply and divide, use the AX as the destination register. These operations are multiprecision operations that require special handling to be sure the result is used correctly. Requiring that the destination be in AX or DX is no great problem for these special case operations.

## Motorola section: CODE COMPATIBILITY ACROSS A FAMILY

### Motorola statements:

Full code compatibility is important to make use of existing application programs. Although operating system code needs to be somewhat compatible, they are also typically subject to and candidates for change that makes the operating system run more efficiently on the new processor.

### Intel analysis:

Intel provides more compatibility across a broader range microprocessors than Motorola. The iAPX 86 family, consisting of 8088, 8086, 80186, 80188, and real mode 80286, provides a wider performance range and feature range in a software-compatible family than Motorola provides in the 68000 family. The iAPX 386 will continue to execute object code from any of Intel's iAPX 86 family.

Motorola loses sight of what compatibility is. Motorola confuses protected mode iAPX286 operation with real mode operation of the iAPX 286. The "Guide for the 8086 Code Writer" Motorola mentions is to allow compatibility between iAPX 86 code and code for protected mode of the iAPX 286. The real mode of the iAPX 286 executes iAPX 86 family object code unchanged. The major requirement being the iAPX 286 system configuration must match that of the iAPX 86, same memory locations and same I/O configuration.

The protected mode iAPX 286 extends the family architecture with more address space, protection, and built-in support for multitasking operating systems. The Protected Virtual Address mode iAPX 286 changed the way segment register contents are interpreted, to extend the family architecture. As a result, iAPX 86 programs which inherently relied on 8086 segment addressing may not work in protected mode on the iAPX 286.

Any changes made to iAPX 86 programs to let them run in protected mode remain backward compatible to the iAPX 86. As long as the new iAPX 286 instructions are not used, 1 megabyte of memory is sufficient, and segment addresses are relocated, the iAPX 86 can execute iAPX 286 protected mode programs.

The new features of the iAPX 286 require changes in an iAPX 86 operating system that wants to use them. A protected, virtual mode operating system does require changes to an iAPX 86 operating system to support the expanded address space and descriptor tables. If the protection features of the iAPX 286 are used, further changes are required to support OS and application privilege levels. To use the hardware-supported tasking features of the iAPX 286 also requires changes.

#### Subsection: M68000 Compatibility - all flavors

The 68000 family also requires changes to use new processor features. A 68000 operating system must change to deal with the different interrupt stack format from 68000. A 68000 operating system must also change to support emulation of read from SR. Virtual memory requires extensive changes to 68000 operating system software.

Protected 68000 systems must also deal with the question: What to do about programs that perform privileged operations? Since some 68000 systems are not protected, the applications software may perform protected operations.

The future 68020 will require operating system changes for the expanded physical memory space, new paging hardware, and potential problems with programs not zeroing the upper 8 address bits or address mode bits correctly.

All 68000 applications code may not be compatible at all with future 68020 due to the extra address lines and address modes.

#### Subsection: iAPX 286 Compatibility

Segment wraps on the iAPX 286 are different. In the very rare case that an iAPX 86 program attempts to wrap around a segment, the iAPX 286 could simulate the operation.

Motorola does not understand bus locking. Their processors provide inadequate support for multiprocessor systems without a bus lock facility. Bus lock is not a compatibility issue in real mode iAPX 286 systems. In protected systems, bus lock falls into the category of "what to do about programs that used to be able to do privileged operations, e.g. HLT, I/O, bus lock".

The 1981 EDN code was written to maximize the usage of 8086 registers/instructions.

The comments on future operating system compatibility made by Motorola are unfounded. The iAPX 386 fully supports the iAPX 286 method of managing memory. An iAPX 286 operating system and applications software will run unchanged on the iAPX 386. New features of the iAPX 386 are invisible to iAPX 86 and 286 applications programs.



## Motorola section: PRIVILEGE LEVEL PROTECTION

### Motorola statements:

Motorola devises a protection model that protects the operating system from applications programs.

### Intel analysis:

Motorola does not understand the ring structure of the protected mode iAPX 286. The protection capabilities in no way imply rigidity or limitation. The iAPX 286 ring structure allows flexibility in system design. Protected mode iAPX 286 programs are not ever forced to use the ring structure. For example, an operating system for the protected mode iAPX 286 could simply run all programs at privilege level 0.

The simplest protected system need only use two privilege levels of the protected mode iAPX 286. Level 0 should be used for the operating system and level 3 for applications programs. This is very analogous to the user-supervisor states of older systems.

But to allow security, even if that security is only two levels, the ring structure of the iAPX 286 is designed into its architecture. Attempts at implementing reliable internal security control without hardware support have resulted in operating systems and utility software which are very large and complex. However, models such as the "security kernel" introduced by Schell have helped refine the hardware and software requirements for a secure system<sup>1</sup>. The secure kernel approach consists of providing the required support for a specific set of security functions at the kernel level in either hardware or software. Ames has identified four architectural properties necessary to support security kernel-based protected systems<sup>2</sup>.

- 1) Support for multiple processes,
- 2) Protection of large segmented virtual memory,
- 3) Minimum of three execution domains,
- 4) Control of access to I/O devices.

Intel's iAPX 286 microprocessor with multitasking support, segmented memory management, four privilege levels (execution domains) and I/O protection mechanism corresponds directly to each of the support criteria.

Neither the 68000, 68010 or future 68020 have yet demonstrated a protection structure designed into the architecture. There appear to be two techniques of adding rings outside the 68000 architecture: use upper address bits as ring numbers or put different rings in different address spaces not directly accessible.

<sup>1</sup> R. R. Schell, "A Security Kernel for a Multiprocessor Microcomputer," Computer, 16, no. 7 (1983), pp 47-53.

<sup>2</sup> S. R. Ames, Jr., G. Gasser, and R. R. Schell, "Security Kernel Design and Implementation: an Introduction," Computer, 16, no. 7, pp. 14-24.

Using upper address bits as ring numbers is very expensive. It requires custom logic external to the CPU to match the ring number used by the program against the current ring number managed by the OS. All controls of the ring number must be done by privileged operating system software. It is not clear that any standard MMU component for the 68000 would be able to work with the ring hardware.

Using privileged instructions to access separate address spaces is a poor implementation of a ring structure. Operating system software must emulate all accesses to other rings. This is required because the multiple address space access instructions must be privileged. About 100-200 instructions would have to be executed to emulate every such access. The resulting slow-down from any access to these other address spaces would be unacceptable.

Traditionally, CPU's and microprocessors such as the 68010 have provided two levels of differentiation (supervisor and user) based on the traditional types of software existing in a system: system and application. In the iAPX286, four levels of protection allow further refinement of systems and applications privileges.

As an applications example, the kernel code and data needed to implement the most basic mechanisms of an operating system could be placed at Privilege Level 0. Privilege Level 1 could contain system utilities and access-protection and policy enforcement procedures. Extensions to the operating system could be placed at Privilege Level 2. Finally, applications could reside at Privilege Level 3.

With the protection architecture of the iAPX 286, access rights are unidirectional. Tasks that are executing at a more privileged level may access data at the same or less privileged levels. The CPU enforces this protection mechanism by comparing the 2-bit wide Descriptor Privilege Level (DPL) field in a descriptor with the Current Privilege Level (CPL) maintained within the processor during execution. The CPL associated with the current code is maintained in the least-significant bits in the code segment register.

In Multics,<sup>1</sup> access to higher levels occurs only through well-defined gates, where access privileges can be verified before less-privileged code is permitted to execute more privileged code. The iAPX 286 directly supports this concept through its implementation of a special access protocol mediated by gates. Specifically the iAPX 286 identifies four types of special "control descriptors" (call, task, interrupt and task-gates) that reside in the descriptor tables.

Gates redirect control to a destination address that cannot be directly accessed by the caller. For application level, code gates form the only vehicle for control transfers to code at more privileged levels.

<sup>1</sup> E. I. Organick, The Multics System: An Examination of its Structure (Cambridge, MA: The MIT Press), 1972

When less privileged code uses a call gate to access a subroutine located at a more privileged level, from both the applications and system programmer's point of view, the subroutine call proceeds just as a call from the same privilege level including pushing any parameters on the stack. The iAPX 286 automatically copies the subroutine parameters from the stack of the callers privilege level to another stack at the target privilege level. A count field in the call gate descriptor specifies the number of words to copy from the originating stack. Once the target subroutine has terminated, it is free to return results or pointers or pointers through one of the registers or by reference to the callers data area in the normal manner.

## Motorola section: MEMORY MANAGEMENT

---

### Motorola statements:

Motorola takes a few picks at the iAPX 286, but this section typically lacks informative statements.

Motorola tries to imply that somehow "programs have to obey certain rules of the scheme, and have to provide certain information to the manager based each routine to be run, resulting in excessive overhead." However, Motorola never takes responsibility for proving the negative comment! No attempt is made to justify the criticism.

### Intel analysis:

Motorola is continuing low-grade "dump and walk on" statements typical of the numerous unsupported or incorrect statements throughout its paper.

The iAPX 286 certainly allows programs to be written without regard to their exact location in physical memory or requiring knowledge of how much physical memory is present.

In fact the iAPX 286 possesses a comprehensive, efficient memory management integrated on-chip, flexible yet forming the basis of low-cost, standard operating systems for virtual memory environments. Ahead is what iAPX 286 memory management is all about!

As an economic necessity, iAPX 286 memory management provides a cost effective alternative to massive amounts of semiconductor memory, which continues to maintain a significant cost per bit disadvantage over disk storage. In addition to providing a more efficient use of existing physical memory, memory management also simplifies software development because of the resulting independence of code and lack of constraints in using address space in programs for either code or data.

Towards that end, the iAPX 286 considerably extends the view of memory organization with its Protected Virtual Address mechanism. Here, as in the Burroughs B5000<sup>1</sup> or Multics Honeywell<sup>2</sup>, a 16-bit segment portion of the address 32-bit pointer becomes a selector, or index, into a table of descriptors that define the virtual-to-physical address translation. The programmer only specifies a selector into the descriptor table. The software development tools resolve the selector to be used, just as a linker resolves external references of non-virtual systems. At run time, the processor uses the information stored in the descriptors for its virtual-to-physical address translation process.

The iAPX 286 divides its virtual address space per task into two equal partitions, global and local virtual memory, which reside in two separate

<sup>1</sup> D. P. Siewiorek, C. G. Bell, and A. Newell, Computer Structures: Principles and Examples (New York: McGraw-Hill, 1982)

<sup>2</sup> E. I. Organick, op. cit.

descriptor tables. Each task in the multitask environment possesses one-half gigabyte of private virtual address contained in the local descriptor table (LDT) and shares with other tasks a one-half gigabyte global virtual address space contained in the global descriptor table (GDT).

Motorola's statements:

This section focusses on the issue of large data structures. The example Motorola offers for performing large array access with the iAPX 286 is claimed to be an adaptation of an Intel paper but in fact such an adaptation has never been recommended by Intel.

Intel analysis:

The large array access example published on Motorola page 26 is completely misleading. Intel has never recommended such a technique for accessing large arrays; Instead an accurate example is shown comparing the 68000 with iAPX 286.

68000/68010 Large-Array Access

Clock Count

```
16      MOV.L    index,D0      ; Get index
12      LSL.L    #2,D0         ; Multiply by 4
12      MOV.L    #array,A0     ; Get base address of array
16      MOV.L    0(D0,A0)      ; Get array element
56 clocks + 12/wait state
5.44 usec at 12.5MHz 1 wait for unprotected 68000/68010
8.0 usec at 10MHz 2 wait for protected 68000/68010
```

iAPX 286 Large Array Access

Clock Count

```
5      mov      bx,index.low   ; Get 32-bit index
5      mov      ax,index.high
2      shl      bx,1           ; Multiply by 2
2      rcl      ax,1
2      shl      bx,1           ; Multiply by 2 again (4 total)
2      rcl      ax,1
3      add      bx,offset array ; Form address of data item
3      adc      ax,0
8      shl      ax,3           ; Form segment table index
3      add      ax,seg array
17     mov      es,ax           ; ES:BX points at element
5      mov      ax,es:[bx]     ; Get element
5      mov      dx,es:[bx+2]   ; 32-bit element in dx:ax
62 clocks
7.75 usec at 8MHz 0-wait protected iAPX 286
```

In reality, the protected iAPX 286 is faster than the protected 68000, even when one assumes the Motorola 68451 MMU just happens to already contain the correct descriptors! This example above is the worst-case example of dealing with 64K segments in software. Summarizing, the speed of the iAPX 286 in performing simple references that account for most of program execution time are performed with full iAPX 286 speed. For the few large-array accesses which occur, the iAPX 286 slows down to 68000/68010 speed.

The future 68020 example again does not include the instruction fetching required. The two-instruction sequence would require at least four 32-bit instruction fetches. The instruction execution time they show should include this time multiplied by the instruction buffer hit rate and the added time required for memory management for those accesses which will require reloading the MMU.

At 16MHz, clock speed, at least three wait states would be necessary for Motorola 68451 memory management and usage of reasonable speed memories in each future 68020 bus cycle. The miss rate of the instruction buffer must also be included to be accurate. With a very effective instruction buffer (miss rate of .4), a realistic estimate of the future 68020 execution time is  $(10+2*3+(4*(4+3)*.4))/16 = 1.6$  usec. This almost 3 times slower than their claim in the simplistic future 68020 example.

## Subsection: Incrementing A Smalltalk or Graphics String Pointer

### Motorola statements:

Returning a third time to the autoincrement addressing mode, Motorola presents another misleading example.

### Intel analysis:

Here is Motorola majoring in the minors, again!

The major Motorola inaccuracy about the iAPX 286 to be pointed out immediately is that the 286 does not require a large array to be in consecutive segments in physical memory but only requires consecutive segment descriptor slots in the descriptor tables in some cases.

The example shown by Motorola for the pointer increment is misleading. The fetching of a instruction or data item followed by a post increment is such a tiny part of any such interpreter that using it for an example is silly. Motorola completely ignores the hundreds or thousands of other instructions required to perform a Smalltalk or graphics operation. The speed of the iAPX 286, especially with the 80287, would show immediately in comparison with a 68000.

The example shown by Motorola pretends that all accesses will require crossing a segment boundary. If a data structure is split across multiple segments, segment crossing will occur in less than one out of a thousand times! Any overhead to test a pointer after such a increment is only 3 clocks to test a condition code.

And down at the very basic level, Motorola uses the wrong clock counts for the iAPX 286 in the example: 16/4, instead of the correct 7/3 for the conditional jump.



Motorola statements:

Motorola mentions several business computers based on the 68000, and mentions LISP, an artificial intelligence language. Motorola also slips in an unrelated and false statement about EDN benchmark I.

Intel analysis:

In this section Motorola mentions several 68000-based personal computer applications. It is an understatement to say the iAPX 86 is used in more business computers than the 68000 and 68010. Incidentally the companies Motorola mentions also offer iAPX 86-based personal computer products. Their iAPX 86 based computer reaps the benefits of a much larger software base than for any 68000 based computer. Note that Tandy has recently announced a personal computer based on the 80186.

One informal survey notwithstanding, LISP is easily available for the iAPX 86. Five versions of LISP are available for the iAPX 86 based personal computers from Norrell Data Systems, Software Tools, Lifeboat Associates, Software Warehouse, and Supersoft.

Bringing in EDN benchmark I (Quicksort) out of the blue, Motorola apparently needs help to correctly determine which of the two items take longer:

(a) Two iAPX segment load instructions - 19 cycles each - 4 usec total

(b) Entire EDN I execution by 68000 - 21 milliseconds.

Obviously (b) takes longer, however Motorola stated that (a) takes longer.

## Subsection: Dynamic Storage Areas and Sophisticated Software Systems

### Motorola statements:

Motorola makes some statements which sound limiting to the iAPX 286, because Motorola again takes the liberty of making misleading statements.

### Intel analysis:

Dynamic variables are an important part of the iAPX 86 family. The usage of dynamic variables is reflected in the architecture. It is more important to provide fast access to small dynamic variable areas than large areas. The iAPX 86 family provides short instructions to access small dynamic variable areas. Longer instructions can access larger areas.

The ENTER and LEAVE instructions of the iAPX 286 efficiently build and remove stack frames used in high-level language procedure calls. Programs that allocate large dynamic variable areas will use them heavily before freeing them. A large dynamic variable area implies that a lot of data will be processed. The overhead to allocate dynamic variables is negligible compared to the amount of time spent processing the data.

Intel development tools for the iAPX 86 and iAPX 286 provide support for more than 64K of dynamic storage in modules. The PASCAL compiler will automatically allocate dynamic storage for such variables.

The discussion of high-level language run-time environments misses the point. The run-time environment does not require call gates. Call gates are only required to change privilege levels or allow run-time redirection of calls. Intel language run-time environments do not require either.

Intel has developed a run-time standard called UDI to allow our languages and the code they generate to run on many different operating systems. Motorola does not offer the same range of operating environments for its run-time environments. UDI-86 has been developed for ISIS, RMX-86, RMX-88, MSDOS, CP/M-86, and Series-IV that allows the same object code to run on either system.

Intel call gates are much faster to change privilege levels than TRAP instructions and copy instructions on 68000.

The comment on the iAPX 432 not being able to manage dynamic stack/variable allocation is incorrect. The iAPX 432 can allocate a new memory area for each invocation of a subroutine to fully contain the addressing capabilities of that procedure. This provides a degree of run-time access checks impossible to duplicate on the 68000 at similar performance. The iAPX 432 allocates memory areas in microcode.

Though Motorola does not take responsibility of naming the 432 benchmark, a reference is being made we believe to the article "A Performance Evaluation of the iAPX 286" published in Computer Architecture News, July 1982 which benchmarks the iAPX 286 and 68000 running PASCAL programs. Later versions of the iAPX 432 running the benchmarks mentioned by Motorola show the iAPX 432 improved its performance 3 times. This level of performance includes complete self-checking and healing capabilities built into the silicon.

Motorola forgot to mention that those PASCAL benchmarks show the iAPX 286 has 2-3 times the performance of a 68000. Motorola possibly hid the source to save itself the embarrassment of the more complete story.

Subsection: Massive Descriptor Overhead for all iAPX 286 Native Mode  
Operating Systems

Motorola's statements:

It appears Motorola would enjoy having the customer base believe the iAPX 286 protected, virtual architecture contains hidden pitfalls.

Intel analysis:

Motorola misunderstands the point again. More time will be spent by programmers managing MMU hardware for protected 68000 systems than programmers will spend on software to manage iAPX 286 MMU.

The iAPX 286 memory management hardware provides a standard method of protecting and managing memory for the iAPX 286. Software that manages iAPX 286 descriptors will work on any iAPX 286 design. This software is written once.

Many 68000 systems develop their own memory management hardware to overcome the performance, cost, availability, and board area problems of the 68451 MMU. As a result, most 68000 systems use different hardware for their MMU. Any operating system written for such non-standard hardware is not transferable to another 68000 design. 68000 operating system designers must constantly rewrite the 68000 MMU management software for different hardware.

There are no pitfalls about the predictable, efficient iAPX 286 memory management technique. The technique which Motorola attempts to present confusingly ("descriptors which describe descriptor tables") is called aliasing.

Aliasing is an Operating System concept described in the iAPX 286 Operating System Writer's Guide, chapter 5. Aliasing is the straightforward technique where one descriptor within the Global Descriptor Table (GDT) and one within the Local Descriptor Table (LDT) are data segment descriptors containing a memory base and limit equal to the base and limit of their respective table, thereby overlaying the table with an "alias" definition as a data segment. In this way, the O.S. kernel can treat the descriptor tables as data segments to read/write them if necessary.

Regarding execution efficiency, the run-time overhead of the iAPX 286 descriptors occurs once: the 19 clock cycles it takes the iAPX 286 to cache the descriptor during a segment register load instruction. Protected 68000 systems take a performance hit on every memory access with an external MMU, besides the operating system code required to manage the MMU. The end result is that iAPX 286 systems run faster than protected 68000 systems.

## Motorola Section: I/O INTERRUPTS ON THE MC68000 AND THE iAPX 286

### Motorola statements:

Motorola describes a variety of interrupt routines. However they do not include the most efficient way of performing the function of EDN benchmark A: increment a counter and return.

### Intel analysis:

Because Motorola is not showing the most efficient algorithms for real address mode and for protected, virtual address mode, Motorola is misleading the reader again in this section. The iAPX 86/286 code examples they show on page 32 are incorrect. The iAPX 86/286 can use the same type of instructions for the interrupt handler as used by the 68000. The correct real mode iAPX 86/286 code, as published in EDN benchmark, is shown below: —

### Real Mode iAPX 86/286 Code for EDN Benchmark A

Clocks

28			; Interrupt
7	inc	cs:count	; Bump counter
21	iret		; Return from interrupt
56	clocks		

The published EDN A numbers are specified to be four times the numbers shown for processing a single interrupt. Placing the counter in the code segment is valid for the unprotected, real address mode iAPX 86 and iAPX 286 since it can be accessed by any program like any other memory-based variable.

A time comparison of an unprotected iAPX 286 running in real address mode against an unprotected 68000 is shown below. The 68000 time is from Motorola's 10MHz execution time adjusted for 12.5MHz as shown on page 37. The Intel time is the time required for the iAPX 286 to execute the benchmark shown in the September, 1981 issue of EDN. Both processors have similar amounts of address access time for memories as measured at the processor pins.

#### Unprotected EDN A Benchmark

8MHz iAPX 286 0 wait-state  
242ns address access time

12.5MHz 68000 1 wait-state  
250ns address access time

28.25 usec

31.4 usec

The correct protected mode iAPX 286 code example is shown below. This code is of the same form as real mode. The difference is that the counter is in the stack instead of the code segment. In this particular benchmark where the sole function is simply to increment a single counter before returning, the ultimate speed happens to be achieved by locating the parameter in the code segment if in real mode, or in the stack segment when in the protected, virtual memory environment. The protected mode code uses the iAPX 286 feature that allocates a different stack for each privilege level. An interrupt procedure in a protected iAPX 286 system runs at level 0. It will always have the same level 0 stack each time it is invoked. The counter will always be at the same offset from the base inside the stack segment.

#### Protected Mode iAPX 286 Code for EDN Benchmark A

Clocks

93				; Interrupt to different level
7	inc	ss:count		; Bump counter
57	iret			; Return from interrupt to
157	clocks			; another privilege level

The following is an execution time comparison of a protected 68000 with a protected iAPX 286. The 68000 requires a 68451 for protection. Two wait states are required by every bus cycle to provide similar memory address access times as the 8MHz iAPX 286.

The comparison below uses Motorola's numbers for the 68000 as shown on page 37 with an extra 72 clocks added to account for the second wait state necessary to provide equivalent memory access times. Note that Motorola assumes the unlikely condition of no MMU segment reload overhead. Intel's numbers for the iAPX 286 are for the protected EDN A benchmark.

#### Protected EDN A Benchmark

8MHz iAPX 286 0 wait-state  
with descriptor overhead

10MHz 68000 2 wait-state  
no MMU segment reload overhead

(242ns address access time)

(250ns address access time)

78.6 usec

BIASED TOWARD  
68000

46.4 usec

Since no MMU overhead is  
taken by Motorola

### Subsection: Task Switch Time Comparison

A typical 68000 task requires 5-7 segments in the 68451 MMU to define its code, data, and stack areas. The average segment descriptor usage is 5.2 segments. A 68000 system with one 68451 MMU can run about 5 tasks with all their memory management info in the MMU. Running more tasks requires the operating system to multiplex the limited 68451 MMU segment resources across several tasks.

The average 68000 task switch time depends on whether the segment descriptor information for the new task is in the MMU. Appendix E shows the 68000 task switch code and timing. In the best case, the new task segment descriptor information is already in the MMU, 108 usec is required. If the new task is not in the MMU, a 10MHz protected 68000 requires 530 usec to reload the MMU with the new task segment descriptors. The worst-case task switch time requires looking through all MMU entries to reload 7 segments requiring 763 usec.

The average 68000 task switch time depends on the number of task switches that require reloading the MMU. If only 20% of all task switches require reloading the MMU, then the average task switch time is  $.8 \times 108 + .2 \times 530 = 192$  usec.

Motorola does not compare their task switch time to Intel's.

- a. The iAPX 286 task switch time is only 22 usec with up to about 4000 tasks! The iAPX 286 task switch time is independent of the number of tasks involved. The average task switch time is 22 usec.
- b. The virtual memory 68010 does not support task switching on a hardware operation. The nearest functional equivalent on a 68010 requires an average task switch time over eight times longer. The worst-case 68010 task switch time is over 30 times slower (22 usec versus 763 usec). Still, the 68010 does not provide protection integrity guaranteed by the hardware, as does the iAPX 286. The protected 68000 average task switch time is over eight times slower! The worst case 68000 task switch time is over 30 times slower! (22 usec versus 763 usec)

### Analysis focus: Fair Interrupt Comparison

The interrupt performance presented by Motorola is unfair to the iAPX 286 because the 68000 execution time overlooks any descriptor reload time for the 68451 MMU. The 68451 MMU will not normally be able to hold all the required descriptors for all the tasks in a protected 68000 system. The iAPX 286 execution time includes all the required descriptor load operations. The 68451 segment descriptor reload time required in a task switch should be represented somewhere in protected 68000 benchmarks.

It is possible to build an OS on the 68000 where all the descriptors for the OS are always in the MMU. If interrupts are handled in the same address space as the OS then an interrupt does not require reloading MMU entries.

The interrupt example discussed by Motorola is correct in that the 68451 MMU can automatically use the operating system segment descriptors for an interrupt handler. However such an OS must still deal with reloading the MMU when it comes time to switch to a new task not currently in the MMU.

A good place to account for the 68451 MMU reload time is in the interrupt benchmarks. To account for the relative frequency of interrupts and task switches and task switch to unused task, we shall assume a 1/12 miss rate for the MMU. This miss rate adds 44 usec (530/12) to the interrupt times. The fairer comparison is shown below:

EDN A Interrupt Benchmark with MMU Overhead

8MHz iAPX 286 0 wait-state

10MHz 68000 2 wait-state

78.6 usec

90 usec

Analysis focus: Misleading iAPX 286 Access Time Claims

The memory access time claims made by Motorola for the iAPX 286 on page 33 are also wrong. An 8MHz iAPX 286 system can use 180ns memories and run at 0 wait states. This is a far cry from the 80ns numbers they claim! See pages 4-20 to 4-23 of the iAPX 286 Hardware Reference Manual, Intel order number 210760, for a complete description of the memory interface logic to provide 180ns of address access time.

Motorola does not show the required memory access times for their processor in the comparison on page 33. Using the same levels of address buffers and data buffers as in the iAPX 286 system, the 0-wait 12.5MHz 68000 requires 170 - 22 - 18 - 22 = 108ns memories. The two 22ns figures are for address buffer and data buffer delays into large busses at worst case temperature and voltage. The 18ns is for the address decode logic to select the proper memory device. With one wait-state, the 12.5MHz 68000 could use 188ns memories. The 1-wait 10MHz 68000 requires 150 - 22 - 18 - 22 = 88ns memories. With two wait states, the 10MHz protected 68000 requires 188ns memories.

Subsection: Intel's Architecture Foils Intel's Own Programmers

Motorola misrepresents the story of the EDN benchmarks. The iAPX 86 EDN benchmark K, originally coded for the April 1981 EDN article, matched the Carnegie-Mellon algorithm given to Intel by EDN. The Motorola benchmark K published in the April issue was coded to an optimized algorithm not the same as that specified by Carnegie Mellon.

The second EDN article in September 1981 was published to force comparison of all benchmarks using the same algorithms and data. The iAPX 86 algorithm sacrificed address space to use the same algorithm as the 68000.

If the algorithm is changed from the 68000 algorithm, the full iAPX 86 address space can be used. A better algorithm was measured on the iAPX 286 for Benchmark K in the iAPX 186,286 Benchmark Report. It allows addressing of a bit array anywhere. Appendix D shows the algorithm.



## Motorola Section: PACKAGING

---

### Motorola statements:

Motorola performs some die temperature "calculations" of die temperature for its products and for the iAPX 286. Motorola uses only the 68000 die, not the more complex and hotter 68010. Motorola does not state the maximum current of its components. For the Intel component, Motorola uses a very incorrect Thermal Resistance value ( $\theta_{JA}$ ) when performing the calculation. And the  $I_{CC}$  value they use for the calculation is even given added tolerance at lowest die-temperature (0°C), when  $I_{CC}$  is the greatest. Therefore the Motorola statement that die-junction temperature ( $T_J$ ) is 224°C is approximately 100°C higher than the true range of die temperatures under conditions of highest ambient temperature.

Based on the incorrect die temperature calculations, Motorola implies the reliability of the 80286 is low. That also is an incorrect statement.

### Intel analysis:

Intel provides extensive actual data throughout this analysis.

Motorola is quite wrong about iAPX 286 die temperature. Their errors are twofold:

- 1) Motorola uses an inflated thermal resistance of 50°C/watt thermal resistance of the 68-lead JEDEC type A leadless chip carrier package and socket. The correct thermal resistance of the ceramic 68-lead JEDEC type A package in Textool 5400 socket is 32°C/watt.
- 2) Motorola uses the data sheet value of  $I_{CC}$  which is 600mA: given added tolerance by Intel even at at the worst case  $I_{CC}$  condition when the die has just been turn on at at lowest possible ambient temperature, 0°C.

Motorola's calculation is based upon an inflated  $\theta_{JA}$  and an  $I_{CC}$  value that Intel provides added  $I_{CC}$  tolerance even under the worst-case situation of  $I_{CC}$  (at 0°C die temperature). Why? Intel typically "guardbands" the value so that users are always assured of providing adequate power supply if they use the specified  $I_{CC}$ 's when totalling up the  $I_{CC}$  needed for the entire system. The value is not especially suited for die temperature calculations and Motorola appears to misconstrue the spec. Normally, competitors do not try to mislead as Motorola repeatedly appears to be doing throughout its document. Intel performs the proper calculations internally and applies the results to our testing methodologies so the components are guaranteed to work under worst case conditions. Providing excellent quality product is one of Intel's greatest concerns.

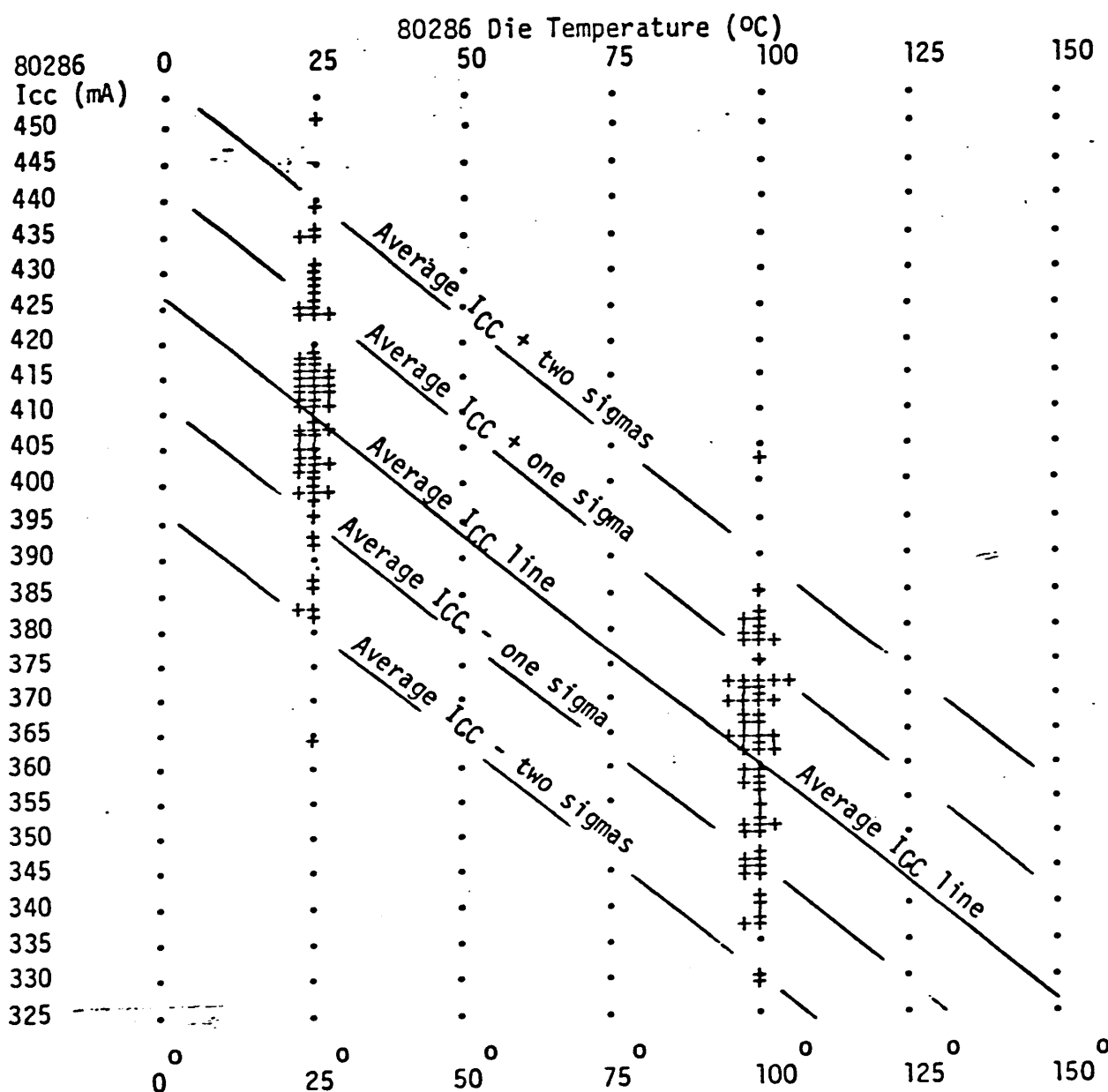
For maximum die temperature calculations, the approach is to measure the  $I_{CC}$  values of actual components, determine the mean and standard deviation of the distribution and calculate the range of die temperatures to be expected at maximum ambient temperature.

Below is detailed raw data providing exactly what one needs for die temperature calculations. The  $I_{CC}$  data below is from 68 components, randomly sampled from portions of two 80286 fabrication lots, 30 units from one lot and 38 from the other. The data shows what  $I_{CC}$  was at die temperature of 25°C and at 100°C. No data points were thrown out at all. Each "+" on the graph represents one  $I_{CC}$  datapoint.

From the graph below, one sees the  $I_{CC}$  consumed by the 80286 is dependent upon the die (chip) temperature. Because the 80286 is an HMOS-II circuit, its power consumption decreases as die temperature increases. For that reason it is incorrect to use the  $I_{CC}$  measured at 0°C, for example, to calculate the die temperature at ambient temperature conditions of 70°C. To be more accurate, one must use the  $I_{CC}$  data taken at higher die temperature, which is closer to the die temperature achieved when the 80286 is in 70°C ambient air and power has been on for a long time (steady state).

The die temperature at the time this data was taken is known because the package and die were stabilized initially in a power-off state with the die, the package and the surrounding air at 25°C. Then power was turned on and the  $I_{CC}$  measurement taken one second afterwards (the die was still essentially at 25°C). Then the measurements were repeated in the same way but with all temperatures at 100°C.

80286  $I_{CC}$  Data versus Die Temperature (68 devices)



STATISTICS: When Die Temp. = 25°C      When Die Temp. = 100°C.  
Mean  $I_{CC}$  = 410 mA      Mean  $I_{CC}$  = 360 mA  
Std. Dev. = 15.9 mA      Std. Dev. = 14.6 mA  
"Architecture Comparison" / Intel Analysis / Pg 77

From the raw data we see the dependency of average  $I_{CC}$  upon the die temperature. Notice that the standard deviation (clustering) of the  $I_{CC}$  measurement remained about the same while the average changed.

Notice also that the standard deviation of the  $I_{CC}$  is only 15mA while the magnitude of the  $I_{CC}$  at room temperature averages 410 mA. Since the standard deviation is only 3.6% of average  $I_{CC}$ , this means the Intel HMOS-II fabrication process is very tightly controlled. This is especially considering that 30 of the units were from one fab lot and 38 were from another lot.

The implication of tight process control at Intel is that this data is very representative of the 80286 components the customer purchased two months ago, today, six months from now, etc.

Now, let's calculate the die temperature under steady state conditions of 70°C ambient air temperature.

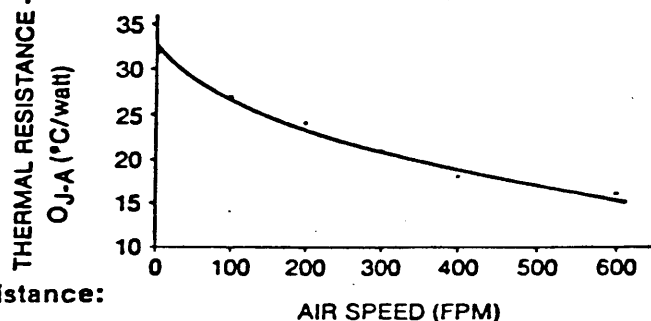
$$\text{Die temperature} = \text{Ambient temp} + (\text{Power} \times \theta_{JA})$$

or, rewriting "Power" as " $V_{CC} \times I_{CC}$ "...

$$\text{Die temperature} = \text{Ambient temp} + (V_{CC} \times I_{CC} \times \theta_{JA})$$

What we know are: Ambient Temp = 70°C  
 $V_{CC}$  = 5.25V worst case  
 $\theta_{JA}$  = 32°C per Watt of power dissipated

How do we know  $\theta_{JA}$  isn't 50 like Motorola says? Because Intel has taken many measurements of that also, which are not shown here (can you trust us, please). Furthermore our data taken two years ago matches data taken independently by Textool, and published in their data sheet.



By the way, if the socket and PC board are mounted vertically, and not horizontally, then the  $\theta_{JA}$  drops to 28°C/Watt we have measured. But for this calculation we assume the worst-worst-worst case!!

Thermal Resistance:

Junction to Ambient (Still Air)

=  $\theta_{J-A}$

= 32°C/watt (reprinted with permission)

Thermal Resistance  
Vs.  
Air Flow

The Textool data gives a value of  $\theta_{JA} = 32^\circ\text{C/Watt}$  worst case with no airflow. That agrees to the degree with Intel's value found by our own thermal lab.

Solving the equation seems to be a complex situation, because we know from the graph that the lower hotter the die temperature the less  $I_{CC}$  the die draws, but in the course of drawing any  $I_{CC}$  the die is going to heat up. So where will the die temperature stabilize?

To solve this, we will rewrite  $I_{CC}$  in terms of die temperature, using the relation we can derive from the graph. We will get a slightly different equation depending on which  $I_{CC}$  line we want to use:

Average  $I_{CC}$  + two standard deviations  
 Average  $I_{CC}$  + one standard deviation  
 Average  $I_{CC}$   
 Average  $I_{CC}$  - one standard deviation  
 Average  $I_{CC}$  - two standard deviations

Let's do all of them! From the graph we can determine that:

The slope of the line is:  $(.410-.360)/(25-100) = -.000762$   
 All  $I_{CC}$  lines have the same slope, just different magnitudes, therefore:

generally:

$$I_{CC} = (-.000762 * \text{dietemp}) + C_x$$

for each particular  $I_{CC}$  line:

Average  $I_{CC}$  + two sigma =  $(-.000762 * \text{dietemp}) + .4588$   
 Average  $I_{CC}$  + one sigma =  $(-.000762 * \text{dietemp}) + .4429$   
 Average  $I_{CC}$  =  $(-.000762 * \text{dietemp}) + .427$   
 Average  $I_{CC}$  - one sigma =  $(-.000762 * \text{dietemp}) + .4111$   
 Average  $I_{CC}$  - two sigma =  $(-.000762 * \text{dietemp}) + .3952$

With this substitution the Die temperature equation becomes the following and is easily solved now:

$$\text{Dietemp} = \text{Ambient temp} + (V_{CC} * [(-.762 * \text{dietemp}) + C_x] * \theta_{JA})$$

simplified, we get:

$$\text{Dietemp} = \frac{\text{Ambient temp} + (V_{CC} * \theta_{JA} * C_x)}{1 + (.000762 * V_{CC} * \theta_{JA})}$$

The answer for die temperature with worst-worst-worst conditions is at hand:

worst case ambient temperature = 70°C  
 worst case  $V_{CC}$  = 5.25V  
 worst case  $\theta_{JA}$ : horizontal position, no airflow = 32°C/W

Worst case die temp + two sigmas = 130.4°C  
 Worst case die temp + one sigma = 128.0°C  
 Worst case die temp = 125.7°C  
 Worst case die temp - one sigma = 123.3°C  
 Worst case die temp - two sigmas = 120.9°C

Intel Thermal Lab measurements on all package types and various die-size combinations, such as the 80286, have been and continue to be a factor of Intel temperature and reliability calculations. Intel temperature calculations stand. The 224°C claim by Motorola is almost 100°C above the accurate maximum 80286 value and therefore has very misleading implications concerning component reliability.

Regarding reliability, the 80186/80286 CPUs, floating-point Coprocessors 8087/80287, Advanced DMA Controller 82258, and support components 82284 and 82288 are all subject to standard Intel reliability criteria including die temperature. Motorola claims to the contrary are baseless. As with the other Intel chips, much reliability data on the 80286 has been taken, per our standard requirements. It is a highly reliable component. A Reliability Report will be available in June.

## Motorola Section: SUMMARY

---

### Motorola's statement:

Motorola finishes off with a repeat inaccuracy by stating that the iAPX 286 does not support virtual memory. This is a completely false statement.

### Intel summary:

The Intel iAPX 86 family also involved tradeoffs. Compatibility and efficient implementation were given more importance by Intel than Motorola. Intel offers a larger software-compatible microprocessor family with more support than Motorola. The result is the world-standard microprocessor architecture as measured by volume shipments and amount of software written for it.

The 68000 offers no real support for high-reliability systems. High reliability requires a completely integrated design. The iAPX 432 is Intel's first product in this area.

The iAPX 286 supports 32-bit data faster than the 68000. The iAPX 286 supports larger data areas than the 68000 or future 68020. It has comparable performance to either of those products in accessing large arrays of data. Intel development tools support development of large, sophisticated software systems much better than Motorola development tools.

Segmentation in the iAPX 86 family is the most powerful method for extending the family to use new VLSI capabilities in a software transparent manner. The task switch and call gate capabilities of the iAPX 286 represent the first step in that direction.

The iAPX 86 family is certainly different from the M68000, however we claim all the advantages in performance, availability, ease-of-use, code, and execution efficiency. The claim that the M68000 is superior to the iAPX 286 is not founded on anything published by Motorola.

Intel desires to maintain close working relationships with our customers, who require credible information and competent guidance from their vendors; therefore, Intel will continue to provide information of value, and industry leadership.

Simply stated, we encourage customers to contact Intel representatives for information on Intel products.

## Comment on Appendix A: MC68000 Pascal is 45% Faster than iAPX 286 Pascal

This section is the most misleading section of the entire document. For some reason Motorola failed to mention that the benchmark report which claims better 68000 performance clearly demonstrated iAPX 286 performance superiority.

The benchmarks mentioned are from an article, "A Performance Evaluation of the Intel 80286" published in Computer Architecture News Vol. 10 No. 5, Sept. 1983 by David Patterson of the University of California at Berkeley. Motorola probably failed to mention the name of the article in their document to prevent anyone from seeing the results!

The article presents the results of running four programs in Pascal on the iAPX 286, iAPX 432, VAX 11/780, and 68000. The same Pascal source was used for all the benchmarks. The numbers for the 8MHz 0-wait 68000 are converted from the 16MHz numbers listed in the report. The 8MHz 68000 numbers in the report are at 4 wait states. It is assumed that one wait state adds 20% to the 68000 0-wait state numbers, see section on 68000 wait state performance, and 2 wait states add 40% to 68000 0-wait state numbers. The numbers in parenthesis are normalized performance numbers using the iAPX 286 performance level as 1.0. Performance is in direct proportion to the reciprocal of execution time. The lower the performance number, the slower the processor executes the program.

### Berkeley Pascal Benchmark Results

Processor	Search	Sieve	Puzzle	Acker	(Ave)
8MHz 80286 0-wait 242ns address access time	1.4 ms (1.0)	168 ms (1.0)	9.138 sec (1.0)	2.218 sec (1.0)	(1.0)
8MHz 68000 0-wait 290ns address access time	2.6 ms (.54)	392 ms (.43)	18.36 sec (.50)	5.5 sec (.40)	(.47)
12.5MHz 68000 1-wait 250ns address access time	2.0 ms (.7)	302 ms (.56)	14.1 sec (.65)	4.2 sec (.53)	(.61)
10MHz 68000 2-wait 250ns address access time	2.9 ms (.48)	439 ms (.38)	20.5 sec (.45)	6.16 sec (.36)	(.42)
VAX 11/780	1.6 ms (.88)	220 ms (.76)	11.9 sec (.77)	7.8 sec (.28)	(.67)

The iAPX 286 has 64% more performance than an unprotected 12.5MHz 68000 at the same address access time! The protected iAPX 286 has over 2 times the performance than a protected 10MHz 68000 at the same address access time!

It is interesting to note that the 12.5MHz 68000 offers only 30% more performance (.61/.47) than the 8MHz 68000 using 14% faster memories (250/290) than at 8MHz. The 12.5MHz 68000 has only 12%  $((.61*250)/(.47*290))$  more performance per nanosecond of address access time than the 8MHz 68000!

The 68000 performance advantage claim based on program size is completely wrong. The measured numbers using similar memory system speeds prove the performance advantage of the iAPX 286.

The reason the code size numbers are larger for Intel Pascal is that our compiler includes more items in the code size it lists than Motorola's Pascal compiler. Intel Pascal code size includes code used for I/O initialization and code for I/O operations used to time the program. The code counted by Intel's Pascal compiler is executed only once, it does not contribute significantly to the overall execution time of the benchmark. The same sort of initialization code is also required by 68000 Pascal. The inner loops of the programs are similar in size for both the Intel and Motorola Pascal code.



## Comment on Appendix B: Independent Benchmarks Show MC68000 Faster than iAPX 286

Motorola misleads the reader in this section again. This section is a poor attempt to hide Intel's performance advantage in the iAPX 186 and iAPX 286 as compared to the M68000. Motorola's 68000 execution time numbers are inconsistent with their own claims!

Motorola claims they used the exact same programs as in the EDN benchmark report titled "16-Bit -uP Benchmarks -- an update with explanations" on pgs. 169-203 in EDN Sept. 16, 1981. Motorola states "the following times are for an EDN benchmark study published in EDN magazine April 1, 1981 and Sept. 16, 1981."

The 68000 and 8086 numbers published on page 37 do not match the numbers published in the EDN article. The inconsistencies are shown in the table below. Since the document uses a different clock frequency for the M68000 execution times than in the EDN report, the clock counts are listed for a side-by-side comparison to the EDN numbers.

### Comparison of EDN Numbers with Motorola Numbers

Benchmark	Claimed 12.5MHz MC68000L12	EDN number 10MHz MC68000L10	Claimed 10MHz 8086-1	EDN number 10MHz 8086-1
EDN A: I/O Interrupt and return	25.6 usec 320 clocks	32 usec 320 clocks	43.2 usec 432 clocks	46.4 usec 464 clocks
EDN B: I/O Interrupt Kernel with FIFO processing	259 usec 3238 clocks	321.6 usec 3216 clocks	396 usec 3960 clocks	402 usec 4020 clocks
EDN E: Character String Search	127 usec 1587 clocks	225.2 usec 2252 clocks	201 usec 2010 clocks	211 usec 2110 clocks
EDN F: Bit set, reset, test	55 usec 688 clocks	69.6 usec 696 clocks	127 usec 1270 clocks	119 usec 1190 clocks
EDN H: Linked-list insertion	116 usec 1450 clocks	121 usec 1210 clocks	269 usec 2690 clocks	210 usec 2100 clocks
EDN I: Quicksort	13.9 ms 174K clocks	17.35 ms 174K clocks	38.3 ms 383K clocks	38.3 ms 383K clocks
EDN K: Bit-matrix transpose	289 usec 3613 clocks	366.2 usec 3662 clocks	939 usec 9390 clocks	523 usec 5230 clocks

clock counts  
should match  
even though frequency  
- is different.

clock counts  
should match.

The 68000 execution times for benchmarks E and H have changed significantly from the numbers given in the EDN report. The claims for 8086-1 execution times for benchmarks A, H, and K are different from the EDN 8086-1 numbers. Why are the 68000 numbers different?

The iAPX 186, 286 execution times are not from any Intel source. The results claimed by Motorola are invalid. The performance advantage of the iAPX 186 and iAPX 286 in real systems as documented in the iAPX 186,286 Benchmark Report remains valid.

The following table shows the more accurate comparison of the unprotected 68000 versus the unprotected iAPX 286. It uses the new EDN 68000 numbers published by Motorola. The 12.5MHz 1-wait numbers are frequency adjusted MC68000+MMU numbers from 10MHz to 12.5MHz. The normalized performance numbers are given in parenthesis using the iAPX 286 as 1.0.

Unprotected 12.5 Mhz 1-wait 68000 Performance			
versus			
Unprotected 8 Mhz 0-wait iAPX 286 Performance			
Benchmark Name and Description	8MHz iAPX 286		12.5MHz 68000
	0-wait bus cycles 242ns access time		1-wait bus cycles 250ns access time
EDN A: I/O Interrupt and return	28.25 usec 226 clocks (1.0)		31.4 usec 392 clocks (.90)
EDN B: I/O Interrupt Kernel with FIFO processing	229.75 usec 1838 clocks (1.0)		314.4 usec 3930 clocks (.73)
EDN E: Character String Search	131.25 usec 1050 clocks (1.0)		141.6 usec 1770 clocks (.93)
EDN F: Bit set, reset, test	90.9 usec 727 clocks (1.0)		66.4 usec 830 clocks (1.37)
EDN H: Linked-list insertion	108.1 usec 865 clocks (1.0)		144.0 usec 1800 clocks (.75)
EDN I: Quicksort	14.75 ms 118K clocks (1.0)		17.1 ms 214K clocks (.86)
EDN K: Bit-matrix transpose	234.9 usec 1879 clocks (1.0)		335.2 usec 4190 clocks (.70)
Equally Weighted Average	(1.0)		(.89)

For these assembly language programs, the iAPX 286 has 12% (.11/.89) more performance than the 68000. Note that these programs cover the types of applications Motorola claims superiority in.

The following table shows the more accurate comparison of a protected 68000 versus a protected iAPX 286. It uses the new EDN 68000 numbers published by Motorola. The 10MHz 2-wait numbers are derived by doubling the clock count difference between the 0-wait and 1-wait numbers, published by Motorola on page 37, and adding them to the 0-wait clock counts. The two interrupt benchmarks include 44 usec added to include descriptor reload work, see section on fair interrupt comparisons.

Protected 10 Mhz 2-wait 68000 Performance  
versus  
Protected 8 Mhz 0-wait iAPX 286 Performance

Benchmark Name and Description	8MHz iAPX 286	10MHz 68000
	0-wait bus cycles 242ns access time	2-wait bus-cycles 250ns access time
EDN A: I/O Interrupt and return	78.6 usec 629 clocks (1.0)	90.4 usec 464 clocks (.87)
EDN B: I/O Interrupt Kernel with FIFO processing	310 usec 2480 clocks (1.0)	506.2 usec 4622 clocks (.61)
EDN E: Character String Search	131.25 usec 1050 clocks (1.0)	195.3 usec 1953 clocks (.67)
EDN F: Bit set, reset, test	90.9 usec 727 clocks (1.0)	97.2 usec 972 clocks (.94)
EDN H: Linked-list insertion	108.1 usec 865 clocks (1.0)	215.0 usec 2150 clocks (.50)
EDN I: Quicksort	20.5 ms 164K clocks (1.0)	25.4 ms 254K clocks (.81)
EDN K: Bit-matrix transpose	234.9 usec 1879 clocks (1.0)	476.7 usec 4767 clocks (.49)
Equally Weighted Average	(1.0)	(.70)

The protected iAPX 286 has 43% (.3/.7) more performance than the protected 68000 in these assembly language benchmarks.

The comment about the relative performance of the 10MHz 8086-1 and protected 8MHz iAPX 286 is inaccurate. Only in Benchmark A does the protected iAPX 286 run slower than the 8086-1. This is due to the fact that benchmark A only measures interrupt entry/exit speeds.

Benchmark A consists of taking an interrupt, incrementing a counter, then returning. The 8086-1 is faster in benchmark A because the protected iAPX 286 performs more work to execute the interrupt routine in a protected environment. The protected iAPX 286 has added descriptor access overhead in entering the interrupt; it accesses the interrupt descriptor table, new code segment descriptor, new stack segment descriptor, and saving old stack pointer. On return, the protected iAPX 286 also performs extra work, read old stack pointer, read old stack descriptor, and read old code segment descriptor. Little work is performed upon data (only increment of the counter). Real interrupt routines perform much more work inside them than benchmark A performs. In benchmark B, more work is done within the interrupt handler. The iAPX 286 is faster than the 8086-1 in benchmark B.

The 8MHz iAPX 286 has a similar address access time (242ns) as the 10MHz 8086-1 (245ns).

#### Analysis focus: 68000 has Poor Wait State Performance

The 68000 is surprisingly sensitive to wait states. In the benchmark numbers on page 37 of the Motorola document, they show the effect of one wait state on the 68000. The results are shown below.

68000 Wait State Performance			
Benchmark		0-wait 68000	1-wait 68000
EDN A:	I/O Interrupt and return	320 clocks (1.0)	392 clocks (1.22)
EDN B:	I/O Interrupt Kernel	3238 clocks (1.0)	3930 clocks (1.21)
EDN E:	Character String Search	1587 clocks (1.0)	1770 clocks (1.12)
EDN F:	Bit set, reset, test	688 clocks (1.0)	830 clocks (1.21)
EDN H:	Linked-list insertion	1450 clocks (1.0)	1800 clocks (1.24)
EDN I:	Quicksort	174K clocks (1.0)	214K clocks (1.23)
EDN K:	Bit-matrix transpose	3613 clocks (1.0)	4190 clocks (1.16)
Equally Weighted Average		(1.0)	(1.20)
ONE WAIT STATE SLOWS 68000 20%.			

On page 4-4 of the iAPX 286 Hardware Reference Manual, the effects of wait states on the iAPX 286 are documented. For high level languages, one wait state adds 23% to the execution time. For assembly language programs, one wait state adds 19% to the execution time.

This is surprising since one wait state increases the 68000 bus cycle by 25% yet one wait state increases the iAPX 286 bus cycle by 50%. One wait state increase 68000 execution time 20% yet one wait state increases 80286 execution time only 19-23%. The 68000 almost increases its execution time in direct proportion to the increase in bus cycle due to wait states. The internal pipelining of the iAPX 286 reduces the impact of wait states below their impact on bus cycle time.

#### Comment on Motorola Appendices C and D and E.

These Motorola Appendices contain only source code listings. No comments to be made.

80286 32-Bit Multiply

```

;      Multiply the 32-bit DX:AX by DI:CX
;      Leave the result in DX:AX
;
dmul  proc    far           Clocks
      call    dmul         ; 28

      mov     bx,ax         ; 2      Save low multiplicand
      mov     ax,dx         ; 2      Get high multiplicand
      mul     cx            ; 21     Form low*high
      mov     si,ax         ; 2      Save result
      mov     ax,di         ; 2      Get high multiplier
      mul     bx            ; 21     Form low*high
      add     si,ax         ; 2      Update high 16 bits of product
      mov     ax,cx         ; 2      Get low multiplicand
      mul     bx            ; 21     Perform low*low
      add     dx,si         ; 2      Form full result
      ret                     ; 26     Done
      31
dmul  endp

```

MC68000 32-bit multiply

```

*      Multiply D0 by D1 and leave result in D0.  D2 and D3 are changed
*      No overflow checking is performed
*
*      O-wait      Number
*      clocks      of memory
*                  references
*
JSR      dmul      20      5

dmul
MOVE.L   D0,D2     4      1      Save multiplicand
SWAP     D2         4      1      Get high word of multiplier
MULU     D1,D2     70     1      Form high*low
MOVE.L   D1,D3     4      1      Get high word of multiplicand
SWAP     D3         4      1
MULU     D0,D3     70     1      Form low*high
ADD.L    D3,D2     6      1      Form partial product
SWAP     D2         4      1      Put into low word
MULU     D1,D0     70     1      Form low*low
ADD.L    D2,D0     6      1      Form full result
RTS                     16     4
*      278      19
*      297 clocks at 1 wait
*      316 clocks at 2 waits

```

MC68000 32-bit divide

\* Divide D0 by D1 leaving quotient in D0 and remainder in D1  
 \* No error checking is performed  
 \*

\* O-wait      Number  
 \* clocks      of memory  
 \*              references  
 \*

JSR      ddiv      20      5

ddiv

CMP.L	#\$0FFFF,D1	14	3	See if divisor is 16 bits
BCC	divisor32	8	1	Jump if not
SWAP	D0	4	1	Look at high word of dividend
CMP	D1,D0	4	1	See if quotient fits in 16 bits
BCC	quotient32	8	1	Jump if not

\* Both quotient and divisor are 16 bits  
 \*

SWAP	D0	4	1	Put dividend back together
DIVU	D1,D0	140	1	Form quotient and remainder
MOVE	D0,D1	4	1	Form quotient in D0 with top
EXG	D0,D1	6	1	16 bits cleared
CLR	D1	4	1	Prepare to form remainder in
SWAP	D1	4	1	D1 with top 16 bits cleared
RTS		16	4	
		<u>236</u>	<u>22</u>	

\* 258 clocks at 1 wait  
 \* 280 clocks at 2 waits  
 \*

quotient32

		2	1	Overhead from jump
MOVE.L	D1,D2	4	1	Put top 16 bits of dividend into
MOVE	D0,D2	4	1	16 bits of D2 with top cleared
DIVU	D1,D2	140	1	Form top 16 bits of quotient
SWAP	D2	4	1	Put them into top 16 bits and get
MOVE	D2,D0	4	1	full 32-bit remainder in D0
SWAP	D0	4	1	
DIVU	D1,D0	140	1	Form lower 16 bits of quotient
MOVE	D0,D2	4	1	Form 32-bit quotient in D2
SWAP	D0	4	1	Get remainder
MOVE	D0,D1	4	1	Form remainder in D1 with top clear
MOVE.L	D2,D0	4	1	Put quotient in right place
RTS		16	4	
		<u>392</u>	<u>28</u>	

\* 420 clocks at 1 wait  
 \* 448 clocks at 2 waits  
 \*

Appendix B  
68000 32-Bit Divide continued

* divisor is 32 bits					
*					
*					
*                   0-wait           Number					
*                   clocks           of memory					
*                   references					
*					
divisor32					
*					
			2	1	Overhead from jump
	CLR.L	D2	6	1	Clear temporary remainder
	MOVEQ	#31,D3	4	1	Set loop count = 32
divloop					
	LSL.L	#1,D0	10	1	Get top bit of dividend
	ROXL.L	#1,D2	10	1	Put into low of temporary remainder
	ADDQ	#1,D0	4	1	Set quotient bit
	SUB.L	D1,D2	6	1	Set if overrun
	BCC	skipclear	10	2	Jump if not
*					
			-2	-1	Adjustment for no jump
	ADD.L	D1,D2	6	1	Adjust result
	SUBQ	#1,D0	4	1	Adjust quotient
skipclear					
	DBRA	D3,divloop	10	2	Loop for all 32 bits
*					
			2	1	Adjustment for exit from loop
	MOVE.L	D2,D1	4	1	Put remainder in place
	RTS		16	4	
*					
92+50*32+8*z   21+8*32+z   z is number of zeros in quotient					
1692+8z   277+z					
*					
1696 + 9z   clocks at 1 wait					
*					
2246 + 10z   clocks at 2 waits					



32-Bit Divide for 80286

```

;      32-bit unsigned divide of DX:AX by DI:DX
;      Quotient is left in DX:AX   Remainder is left in DI:DI

```

```

ddiv   proc   far           clocks
       call  ddiv           ; 28

       or    di,di          ; 2   Look for 16-bit divisor
       jne   divisor_32     ; 3   Jump if 32-bit divisor

       cmp   cx,dx          ; 2   See if quotient will be 16-bit value
       jbe   quotient_32    ; 3   Jump if 16-bit divide won't work

       div   cx              ; 22  Do 16-bit divide
       xor   si,si           ; 2   Clear upper 16-bits of remainder
       mov   di,dx           ; 2   Set remainder
       xor   dx,dx           ; 2   Clear upper quotient
       ret                                ; 26 Done
                                92

```

```

;
;      The divisor is 16-bits but the quotient will not fit in 16 bits
;
quotient_32:
; 44 Clock count from previous call and tests
       mov   di,ax          ; 2   Save low dividend
       mov   ax,dx          ; 2   Divide high dividend by divisor
       xor   dx,dx          ; 2
       div   cx              ; 22
       xchg  ax,di          ; 3   Save high quotient in DI
                                ;   Put low dividend in AX
                                ;   Remainder is high dividend for next divide
       div   cx              ; 22 Form low quotient in AX
       xchg  di,dx          ; 3   Restore high quotient, set low remainder
       xor   si,si          ; 2   Clear high remainder
       ret                                ; 26 Done
                                T28

```

```

;
;      The divisor is 32 bit, do it the long way
;
divisor_32:
; 38 Clock count from call and test
       push  bp             ; 3   Save working register
       mov   bp,cx          ; 2   Save low divisor
       mov   bx,di          ; 2   Save high divisor
       xor   si,si          ; 2   Clear remainder value
       mov   di,si          ; 2
       mov   cx,32          ; 3   Set loop count
                                52

```

Appendix C  
iAPX 286 32-bit Divide continued

```

;
;
; Run through the following loop 32 times
; At first, the dividend is shifted left one bit at a time into the
; remainder registers until the result is large enough to be greater
; than the divisor. At that point, the highest bit of the divisor is
; set.
;
div_32_loop:
    sal    ax,1        ; 2  Shift dividend left one
    rcl    dx,1        ; 2
    rcl    di,1        ; 2  Put shift out into remainder
    rcl    si,1        ; 2
    sub    di,bp        ; 2  Subtract divisor from dividend
    sbb    si,bx        ; 2
    inc    ax           ; 2  Set low quotient bit
    jnb    quotient_set ; 3  Jump if no override
    ; Clock count if jump is taken
    add    di,bp        ; 2  Restore dividend
    adc    si,bx        ; 2
    dec    ax           ; 2  Remove quotient bit
quotient_set:
    loop   div_32_loop  ; 10 Continue till all 32-bits are done

    pop    bp           ; 5  Restore register
    ret     ; 26 Done
    1133
ddiv     endp

```

iAPX 86/286 EDN Benchmark K: Bit Matrix Transpose

```

;
; Base address of array is in DS:SI
; Bit offset of first bit in first byte is in BX
; Array size in bits is in CX
;
; 1879 Clocks on iAPX 286
;
ENDK:
    push    si                ; Save working registers
    push    bx
    push    cx
    mov     bp,cx             ; Get array size in bits
    shr     bp,1              ; Form byte size from array bit size
    shr     bp,1
    shr     bp,1
    mov     dx,cx             ; Set initial column pointer
    and     cl,7              ; Form bit-shift count
    dec     dx                ; Adjust outer loop counter
    mov     ch,[bx]           ; Get first data value
    jmp     short enter_outer

outer_loop:
    mov     bl,ch             ; Save old bit position
    ror     ch,cl             ; Move to next bit position
    cmp     ch,bl             ; See if byte boundary was crossed
    cmc
    adc     si,bp             ; Go to next byte boundary
    ror     ch,1              ; Update to next bit position
    adc     si,0              ; Go to next byte if byte boundary was crossed

enter_outer:
    push    dx                ; Save current row length
    push    si                ; Save row byte pointer
    mov     di,si
    mov     al,ch
    mov     ah,ch
    jmp     short inner_loop

flip_bits:
    xor     [di],ah           ; Flip both bits
    xor     [si],al
    dec     dx                ; See if done
    jz      exit_inner

```

Appendix D continued  
EDN Benchmark K for iAPX 86/286

```
inner_loop:
    mov     bh,ah           ; Save current bit position
    ror     ah,cl           ; Go to next column bit position
    cmp     ah,bh           ; See if we crossed a byte boundary
    cmc
    adc     di,bp           ; Bump byte pointer if we did
    ror     al,1            ; Bump row bit address
    adc     si,0            ; Update row byte address if byte was crossed
    mov     bx,ax           ; Get copies of bit positions
    and     bh,[di]         ; Get one bit position
    and     bl,[si]         ; Get second bit position
    xor     bl,bh           ; See if same values
    jpo     flip_bits       ; Jump if they should be flipped

    dec     dx              ; Bump counter
    jnz     inner_loop      ; Continue if not done
exit_inner:
    pop     si              ; Restore row byte address
    pop     dx              ; Restore outer counter
    dec     dx              ; See if done
    ja      outer_loop

    pop     cx              ; Restore registers
    pop     bx
    pop     si
    ret
```

MC68000 + MC68451 Task Switch Code

\* Enter from an interrupt, save current task state then load a new one  
 \* Control is received here from the interrupt vector table  
 \* We are using the supervisor stack and user status and PC is on sup. stack

		0-wait clocks	Number of memory references	
		44	8	Interrupt time
interruptn				
MOVE.L	A6,-(SP)	14	3	Save user A6
MOVE	#\$2700,SR	12	2	Mask off further interrupts
MOVE.L	utask,A6	20	5	Get address of user task block
MOVE.L	(SP)+,72(A6)	24	6	Put user A6 into UTB
MOVEM.L	D0-D7/A0-A5,16(A6)	152	31	Save user registers except A7
MOVE	(SP)+,8(A6)	16	4	Save user status value
MOVE.L	(SP)+,4(A6)	24	6	Save user PC
MOVE	USP,A0	4	1	Get user A7
MOVE.L	A0,76(A6)	16	4	Save in UTB
MOVE.L	#inthndlr,A6	12	3	Get address of new task
JMP	taskswitch	12	3	Switch state to new task
		<u>350</u>	<u>76</u>	
	426 clocks at 1 wait			
	502 clocks at 2 waits			

\* Switch the CPU and MMU state to that required for the new task  
 \* A6 points at the user task block for the new task

taskswitch				
MOVE.B	(A6),D0	8	2	Get address space id for MMU
MOVE.L	A6,utask	20	5	Set current task id
MOVE.B	D0,MMUbase+ASTUD	16	4	Set MMU user data space id
		18		MMU delays
MOVE.B	D0,MMUbase+ASTUC	16	4	Set MMU user code space id
		18		MMU delays
ORI.B	#\$80,D0	8	2	Set OS id bit for MMU
MOVE.B	D0,MMUbase+ASTSD	16	4	Set MMU OS data space id
		18		MMU delays
MOVE.L	#supstack,A7	12	3	Set OS stack
MOVE.L	4(A6),-(SP)	24	6	Set user PC for RTE
MOVE	8(A6),-(SP)	16	4	Set user status
MOVE.L	76(A6),A0	16	4	Get user stack
MOVE	A0,USP	4	1	
MOVEM.L	16(A6),D0-D7/A0-A6	136	34	Reload user register state
RTE		20	5	Return to user state
		<u>366</u>	<u>78</u>	
	444 clocks at 1 wait			
	522 clocks at 2 waits			

# Appendix E continued

*	Test if the current task has its address space info in the MMU				
*	Insert this code before loading DO with user id for MMU				
*					
*		0-wait	Number		
*		clocks	of memory		
*			references		
*					
	BTST.B	#inMMU,12(A6)	16	4	Test MMU status bit
	BEQ	outMMU	10	2	Jump if not in MMU
setupMMU					
	ADDQ	#1,82(A6)	14	4	Update task usage count
*			<u>40</u>	<u>10</u>	All info in MMU, get user id
*		50 clocks at 1 wait			
*		60 clocks at 2 waits			
*					
*	Handle the case of no space in MMU				
*	Fill free spaces then dump old spaces				
*	Insert this code after RTE from task switch				
*					
*			2	1	From BEQ instruction
outMMU					
	LEA	84(A6),A5	8	2	Get start of MMU segment info
	MOVE.L	#MMUbase,A1	12	3	Base of MMU list, 32 bytes/MMU
	CLR	DO	4	1	Form 16 bit index value
	MOVE	mmufseg,DO	16	4	Get number of free MMU entries
	MOVE.L	#mmuflist,A0	12	3	Get start of list
	LEA	(A0,DO.W),A0	12	2	Set end of list
	BEQ	freeseqs	10	2	Jump if no free space
*			<u>76</u>	<u>18</u>	

# Appendix E continued

*	Allocate all free entries with the list of segments starting at A5				
*	D0 has count of free segments remaining				
*					
*		0-wait	Number		
*		clocks	of memory		
*			references		
*					
*		-2	-1	Adjustment for no jump	
*	allocseg				
	BTST.B	#ENABLED,7(A5)	16	4	See if segment entry valid
	BEQ	exitsetMMU	8	1	Exit if not used
*					
*	Locate the MMU with the free segment, then load the accumulator with				
*	all pertinent information. Use the load descriptor operation and test				
*	for a collision.				
*					
*	loadentry				
	MOVE.B	-(A0),D1	10	2	Get free segment number
	MOVE	D1,D2	4	1	Get MMU number in A2
	LSR	#5,D2	16	1	
	LSL	#6,D2	18	1	
	LEA	\$20(A1,D2.W),A2	12	2	A2 points at ACO of MMU
	MOVE	D1,D2	4	1	Form DP value
	ANDI	#31,D2	8	2	Form descriptor number in D1
	MOVE.B	D2,\$29(A1)	12	3	Set DP in all MMUs
*			18/25		MMU delays
*	MOVE.L	(A5)+,(A2)+	20	5	Load LBA and LAM
*			36/50		MMU delays
*	MOVE.L	(A5)+,(A2)+	20	5	Load PBA and ASN and status
*			36/50		MMU delays
*	MOVE.B	(A5)+,(A2)+	12	3	Load ASM
*			18/25		MMU delays
	MOVE.B	D1,(A5)+	8	2	Save descriptor id of segment
	MOVE.B	\$7(A2),D1	16	4	Attempt to load descriptor
*			34/39		MMU delays
	BNE	mmucoll	8	1	Jump if collision
	DBRA	D0,allocseg	10	2	Loop till free entries used
*			200+142/189	39	
*	381 clocks per entry at 1 wait and 1 MMU				
*	428 clocks per entry at 1 wait and more than 1 MMU				
*	420 clocks per entry at 2 waits and 1 MMU				
*	467 clocks per entry at 2 waits and more than 1 MMU				

# Appendix E continued

```

*   No more free entries exist in the MMU, go find the least used task and
*   kick it out of the MMU
*   A1 points at MMUbase  A0 points at head of free MMUentry list
*   D0 has free entry count  A5 points at next segment to load
*
*
*           0-wait    Number
*           clocks    of memory
*                   references
*           2         1    DBRA exit penalty
freeseqs
  MOVE.L    #taskhead,D1.    12      3    Get start of task list
  MOVE      #$0FFFF,D3      12      2    Set most used count
*
*   Scan through the task list looking for an unused task or quiet one
*
taskscan
  MOVE.L    D1,A3            4        1    Set base of entry
  MOVE      82(A3),D1        12       3    Get task usage count
  BTST.B    #inMMU,12(A3)    16       4    See if in MMU
  BEQ       usedtask         8        1    Jump if not in MMU

  MOVE      82(A3),D1        12       3    Get usage count
  BEQ       foundent         8        1    Exit if unused task found

  CMP       D3,D1            4        1    See if least used
  BCC       usedtask         10       2    Jump if used more
*
*           -2        -1    Adjustment for no branch
  MOVE      D1,D3            4        1    Set new low
  MOVE.L    A3,A2            4        1    Save task address
usedtask
  MOVE.L    88(A3),D1        16       4    Get address of next entry
  BNE       taskscan         10       2    Look at next if not
*           -2        -1    Adjustment for no branch
*           24 + 100n    5 + 22n/wait
*                   n is number of tasks scanned
*
*   29 + 122n clocks at 1 wait
*   34 + 144n clocks at 2 waits

```



— •

\*\*\*\*\*

**foundent**

## free loop

✱

**\***

BRA      loadentry      10      2      Go try filling the entries

**\***

**\***

**\***

**\***

★

**\***

**\***

BRA	setupMMU	10	2	Continue in search
-----	----------	----	---	--------------------

★

## Appendix E continued

The first part of Appendix E shows the code required for a very simple task if all the segments for a task are already in the MMU. In the simplest, fastest case 1024 clocks are required. The remaining code of Appendix E shows the code required if the new task does not have all its segments in the MMU. This code implements an very simple LRU algorithm for moving task segments into the MMU only when needed. The segments from other less used tasks are removed as needed to make space. A practical system might require further complications of fixed interrupt handler segments and shared segments.

The extra test to see if all the new task code is in the MMU adds 60 clocks at 2 wait-states. With these figures, the best case interrupt response time is 1084 clocks at 2 wait states assuming the segments for all interrupt handlers are always kept in the MMU(s).

The task switch time that requires reloading the MMU has six distinct stages for n segment entries and W wait states:

Step	Clocks	2-wait clocks
Saving current task state	$350 + 76W$ clocks	502 clocks
Test for presence in MMU	$40 + 10W$ clocks	60 clocks
Setup for loading MMU	$72 + 15W$ clocks	102 clocks
Scanning for LRU entry	$24 + 102N + 5W + 20NW$ clocks	$34 + 144N$ clocks
Freeing LRU entry	$44 + 172N + 10W + 26NW$ clocks	$64 + 224N$ clocks
Loading free entries	$334N + 47NW$ clocks	$428N$ clocks

Reloading the MMU for a task switch will require clearing 5.2 segments and writing 5.2 segments on the average. A scan of the running task list is also required to identify the LRU task. It is assumed that 8 entries are scanned on the average before finding a free entry. The execution time of a task switch at 2 wait states is  $502 + 60 + 102 + 34 + 144*8 + 64 + 224*5.2 + 428*5.2 = 5304$  clocks.

The worst case task switch time occurs when freeing 7 segments, scanning 16 tasks, and loading 7 segments. At 2 wait states, the worst case task switch is  $502 + 60 + 102 + 34 + 144*16 + 64 + 224*7 + 428*7 = 7630$  clocks.