COMPILER AND OPERATING SYSTEM REQUIREMENTS FOR 16-BIT MICROCOMPUTER ARCHITECTURES: INTEL 8086, ZILOG Z8000 AND MOTOROLA MC68000

by

Mahadevan Ganapathi
and
James R. Goodman

Computer Sciences Technical Report #452

October 1981

Compiler and Operating System requirements for 16-bit Microcomputer Architectures: Intel 8086, Zilog Z8000 and Motorola MC68000

Mahadevan Ganapathi
James R. Goodman

University of Wisconsin - Madison

Abstract

The current generation of microprocessors are being programmed, to a much greater extent than earlier microprocessors, in high level languages. Their capabilities are such that the functions normally performed by an operating system require some sophistication. As a result, operating systems and compiler requirements are of considerable importance in comparing the various products available. A number of the needs of operating systems and compilers are identified and the three most popular 16-bit microprocessors are evaluated in the light of these criteria. It is shown that all processors have shortcomings that limit the generality of their application.

[1] Introduction

Advances in semiconductor technology have permitted the introduction of single-chip processors that have many features which, until recently, were found only on main-frame computers. While the three most popular, currently available processors have been compared in numerous places for design and performance considerations, [Toong 81, Grappel 81], little has been said about the capabilities of these processors to support the environment typical of larger processors, viz., sophisticated operating systems and application programs, written in high-level languages.

Such operating systems introduce requirements for a number of features that have not previously been available on microprocessors. Also, with the significantly greater computational capability of the new processors comes the necessity for accessing considerably more memory than in the past. Significant progress has been made in recent years in main frames in methods for managing memory, and this knowledge is quickly being applied to the highly similar situations of microprocessors.

It is now desirable to support numerous concurrent processes which can be guaranteed not to interfere with each other. The operating system itself needs protection from processes which may not be entirely trustworthy. It also must have special capability for manipulating certain resources, allocating and deallocating them in an appropriate manner. Thus a method is needed for the restriction of access to areas of memory and to instructions which are capable of affecting the allocation of resources.

Support is needed to enable the operating system to control the use of the processor itself, and for the rapid switch from one process to another, storing away the former's state, so that it can be continued at a later time. In addition, some hardware assistance is necessary to allow processes to cooperate in a variety of ways and to synchronize their operations.

The increased capabilities of the new microprocessors has meant, to a large extent, that increasing proportions of applications are written primarily in high-level languages. Thus the efficiency of compiler-generated code has become much more critical than on earlier machines, where many applications were written entirely in assembly language. The ease with which code can be generated and its compactness, as well as the optimization of the processor's registers, are significantly affected by the regularity of its registers and the orthogonality of its addressing modes.

The following section gives an overview of the architectures of Intel 8086 [Intel 78], the Motorola MC68000 [Motorola 79, Stritter 79], and the Zilog Z8000 [Zilog 79, Peuto 79]. Section 3 discusses the features that make code generation easy for compilers and analyzes the considerable differences in the processors' architectures that influence this aspect. Producing a code generator for these architectures poses some problems. Some of them are identified in the light of attributed grammar descriptions for these target architectures [Ganapathi 80]. Section 4 details the requirements posed by an operating system, and compares the three processors in their support. Section 5 summarizes the strengths and weaknesses of the processors with respect to the requirements introduced in sections 3 and 4.

[2] Hardware

This section will describe the features of the three microprocessors which are important for the discussion of this paper. It is not intended as a comprehensive analysis of the three. For such a study, the reader is referred to [Toong 81].

The 8086 is the descendent of the Intel 4004, 8008, 8080, and 8085 microprocessors, and strongly reflects that heritage. The bus structure is compatible with peripherals that interface with the 8080 or the 8085. The design of the 8086 is generally upward compatible, which has attracted many applications which were originally implemented on the earlier Intel microprocessors. Although upward compatibility enhances software portability, it carries forward the drawbacks and deficiencies of earlier designs. The instruction set of the 8086 is not orthogonal. Most instructions apply only to certain registers, which are therefore not explicitly named. This approach appears to produce more compact code, and minimizes the problem of allocating registers, but can make code generation tortuous under less-than-favorable circumstances. Contrary to this philosophy of upward compatibility, the architectures and instruction sets of the Zilog Z8000 and the Motorola MC68000 represent significant deviations from their predecessors, the Z80 and MC6800, respectively. Both the Z8000 and MC68000 have adopted relatively regular architectures and instruction sets typical of minicomputers.

The register set of the 8086 was designed to be upward compatible with the Intel 8080/8085. This compatibility is somewhat achieved by pairing 8080 registers (AH, AL, BH, BL, CH, CL, DH, DL) to define four 8086 16-bit regis-

ters. Each half of the four registers is therefore independently accessible with much the same instruction set as the 8080. The pairs can also be accessed to perform 16-bit arithmetic. The architecture has been extended to include seven new registers: base pointer, source index, destination index, and four segment pointer registers. The source index (SI) and destination index (DI) are for data movement (normally within the current data segment). The base pointer register can be used to access scalars and structured data objects by containing the base address of the structure. In a block structured language, the base pointer can serve as the frame pointer for an activation record.

Among the Z8000's architectural resources are 24 16-bit registers, sixteen of which are general purpose, of which all but one can be used as accumulators. With the exception of R0, all registers can be used as index registers, base registers, and memory pointers for indirect addressing. Except for the stack pointer, no registers are ever implied by an instruction. More register flexibility is achieved by the overlapping and pairing of registers. For byte operations, the first eight 16-bit registers are treated as sixteen 8-bit registers. The sixteen 16-bit registers are grouped in pairs to form 32-bit longword registers. Similarly, the register set is grouped into quadruples to form 64-bit registers. The quadwords are used by a few instructions such as multiply, divide and sign extend for greater precision.

The MC68000 contains eight 32-bit data registers, nine 32-bit address/stack pointer registers and a 16-bit status register. All address registers can serve as stack pointers. In addition, all data and address regis-

ters may serve as index registers. The register set of the MC68000 exhibits a high degree of orthogonality. For example, all data registers function identically as do all the address registers. However, it is not completely orthogonal, and therefore presents some problems for compiler code generation [Gilmore 80]. There are no instructions for converting binary to or from BCD or ASCII. There is severe lack of orthogonality in instructions to operate on address registers (e.g. comparing pointers with zero).

The processor has user and supervisor modes of privilege with a stack pointer and a status byte provided for each mode. The 32-bit program counter supports 32-bit calculations, but only the low-order 24 bits are used in conjunction with the address bus.

All three architectures support byte addressing at even or odd addresses. In addition, the Z8000 and the MC68000 allow accessing of bit data by specification of the bit number along with the address of the byte containing the bit. However, only the 8086 is capable of accessing non-aligned data, i.e. words in memory can be located at even or odd addresses. For word fetches, it automatically performs the proper memory access, one access if at an even boundary, two if at an odd. Except for performance degradation, this access scheme remains transparent to the programmer. It is recommended that word operations be aligned at even addresses, especially on the stack, since odd address references to the stack may adversely affect context switching time for interrupt processing (real time) or task switching.

The 8086 has four segment registers to handle memory management on chip by segmentation. One segment register exists for each of code, data, stack,

and an extra segment (normally used as secondary data segment). Since I/O can be memory mapped, the chip can control a full 64 kilobytes of address space for I/O ports. This division of instruction and data space allows re-entrant code. Physical addresses are determined by adding a sixteen-bit effective address to a sixteen-bit segment register, offset by four bits, giving a twenty-bit address. Thus the 8086 can access up to one megabyte of memory. Each segment is 64 kilobytes allowing up to 256 kilobytes to be addressed as combined instruction, stack, and data.

The segmented memory map is under explicit program control, rather than hardware control; so the programmer must be careful. If the program's instruction space is greater than 64 kilobytes, allowing the instruction pointer (PC) to increment will cause the program to jump to the beginning of the segment. To execute the extra (i.e. > 64k) part of the program, the code segment register (CS) must be correctly updated. This could be accomplished with a compiler/loader package. A similar situation can occur with the data segment, stack segment, and extra segment. Note that this makes detection of stack overflow particularly unpleasant.

The 8086 has no memory protection and no privileged instructions.

Two versions of the Z8000 microprocessor exist: the Z8001 segmented MPU and the Z8002 non-segmented MPU. The Z8001 can directly address eight megabytes of memory, whereas the Z8002 directly addresses only 64 kilobytes. The Z8001 provides direct addressing in applications requiring large amounts of memory. The Z8010 Memory Management Unit provides address translation and memory protection for up to 8 megabytes of memory. For still larger memory

requirements a Z8001 and multiple Z8010 units permit the use of several 8 megabyte address spaces (up to 48 megabytes total for code, data, and stack in system and normal processor modes). The 23-bit addresses of the Z8001 are split into two parts, a 7-bit segment number and a 16-bit offset address. The Z8001 can operate using this segmentation mode or can operate in a nonsegmented mode.

The segmented mapping is very similar to the method used on the DEC PDP-11. The Z8010 converts the segmented address into the actual physical address transparent to the user software (unlike the 8086). If a reference occurs to a segment that has been protected, the Z8010 can generate a trap signal to the Z8001 microprocessor using the segment trap input. This method provides a simple mechanism for memory protection. In addition, the Z8000 provides user and supervisor modes that restrict the use of certain critical instructions.

The MC68000's 24-bit program counter provides a very large direct memory addressing range of 16 megabytes. This addressing capability, coupled with a proposed external memory management unit (MC68451), will allow the development of large, modular programs. A proposed memory management controller may be used when the processor is in user mode to manage up to 64 megabytes for the programmer. The controller's memory management operations are completely transparent to a process in user mode. These operations can be changed only in the supervisor mode. The memory management controller will provide both management of variable sized segments and dynamic management of multi-user memory relocation and protection (e.g. protection of read-only data or code segments).

To service devices with real-time requirements, the 8086 provides 256 interrupts, 255 of which are maskable and all are vectored. Each vector element needs four bytes to indicate the offset and segment address. To service the 255 maskable interrupts, the following procedure is adopted. First, the MPU issues two interrupt acknowledge signals. The interrupt interface must respond with an interrupt number on the second signal. Secondly, the 8086 pushes the status flags and performs an indirect transfer through the vector corresponding to the interrupt number.

In order to remain compatible with the 8080/8085, the 8086's interrupts 0 through 31 are reserved for the manufacturer's use. Five interrupts are reserved for internal conditions: zero divide, single step, overflow, one byte interrupt instruction (trap), and a nonmaskable interrupt. These interrupts have top priority, followed by the rest of the maskable interrupts. Also, note that because of the register structure, handling of interrupts will usually mean pushing all registers onto the stack upon entering an interrupt routine and restoring them upon exit. The 8086 exhibits a serious deficiency in that all zeros is a legitimate opcode and it does not trap on illegal instructions. This deficiency poses problems when recovering from certain kinds of errors.

The Z8000 has seven interrupts and traps, both internal and external. These interrupts are arranged into levels of priority. Three of them are external inputs: nonmaskable, vectored, and nonvectored interrupt. The vectored and nonvectored interrupts are maskable. One of the four traps is also external and is from the Z8010 memory management unit. The remaining three

traps occur when certain privileged instructions are attempted in normal mode, system call instructions, and illegal instructions. The descending order of priority is as follows: internal traps, nonmaskable interrupts, segment trap, and vectored and nonvectored interrupts.

When an interrupt occurs, the program status is pushed onto the system stack along with an additional word that indicates the reason for occurrence. In the case of internal traps, the first word of the trapped instruction is the "reason" word. For the segment trap and all interrupts, the reason word is the vector on the data bus that is read by the MPU during the acknowledge machine cycle. The new processor state (i.e. address of trap or interrupt routine) is then fetched from the new program-status table in system memory that is specified from a new program-status area pointer. After the interrupt or trap routine completes, the old (saved) processor state is restored and execution continues.

Interrupts in the MC68000 are given a priority and may be vectored or nonvectored. The priorities range from zero to seven, with seven being the highest (nonmaskable). The incoming interrupt request is only recognized if it is greater in priority than the current processor priority. Level seven is an exception that will acknowledge another level seven interrupt request immediately. Vector interrupt processing begins after completion of the current instruction. At this point the processor compares the incoming priority with the mask in the status register. If the incoming level is of sufficient priority, it is processed, otherwise the request is postponed.

Exceptions, either internally or externally generated, cause the MC68000 to enter supervisor mode. The associated vector number is determined through an external fetch in the case of a vectored interrupt and internally for all other cases. Then, the current status is saved on the supervisor stack and finally, the program counter value is fetched from the exception vector and processing of the exception routine proceeds. There are 256 possible exception vectors and each vector contains four bytes. The upper 192 vectors are reserved for vectored interrupts and the lower 64 dedicated to traps and non-vectored interrupts. Several hardware traps are provided to indicate abnormal internal conditions: bus error, address error, illegal instruction, zero divide, overflow (TRAPV), privilege violation, and unimplemented instructions. Since many existing peripheral devices cannot supply a vector number, provisions are made to allow non-vectored interrupts. For such cases, the MC68000 will look at the priority level and fetch the appropriate auto-vector. Traps are handled like interrupts except that the vector number is specified internally.

If the processor is currently processing a level six interrupt and a level two interrupt request appears, the processor will ignore this request until the level six interrupt has completed. Afterwards, if the level two interrupt is still present, the processor will honor the request. However, if the processor was operating on a level four interrupt at the time the level six was honored, then the processor will ignore the level two request, complete the level four interrupt routine, and finally do the level two request if it is still present.

[3] Machine-dependent issues in compilers

The design of the instruction set of an architecture is crucial to the efficiency of object programs produced by a compiler. The phases in a compiler that depend on the target architecture are storage assignment (packing and binding), code generation and object code optimization (some of which can be done machine independently [Aho 77]). Machine dependent optimization is concerned with utilizing special instructions (e.g. increment instead of addition by one), addressing modes (e.g. using auto-increment to subsume additions) and span-dependent instructions [Szymanski 78, 80].

In some compilers, a separate storage and temporary storage allocation phase decides which source language variables are to be bound to registers. Such allocation is based on global flow analysis [Beatty 74, Johnsson 75]. Others allocate registers during code generation. Registers are useful for short naming as well as high speed access. Storage assignment (e.g. binding constants, simple and aggregate variables to machine storage locations such as static area in memory, registers, or hardware stack) may thus be based on a preplanned strategy (global flow analysis) or simply done "on-the-fly". Some storage allocation and reclamation is done at fixed intervals in a program (e.g. allocatable at block entry and releasable at block exit for simple variables, arrays). Others are done at arbitrary moments (in the case of pointers, as long as they allow access to heap objects, no deallocation can be done). To implement block structure and recursion, a stack and a display mechanism are commonly used. The availability of a display pointer (DP), frame pointer (FP) or base register and hardware stack (with a stack pointer

SP) simplifies implementation of the run-time display mechanism. Variables whose space requirements can be determined at compile time (e.g. integers, reals, characters and booleans) can be bound to general purpose registers or addresses with a fixed offset from the DP, FP or a pointer to the static area in memory. All three architectures provide a stack and the above display registers. However, it would be of great advantage if they also provided hardware support for up-level addressing of variables (i.e. addresses of variables that are neither local nor global). Addresses for dynamic arrays, strings and pointers are to be calculated at run-time. Therefore, storage assignment for such variables is done at run-time. Usually, for dynamic arrays, the dimensions of the array are known at block entry. Since space for arrays is releasable at block exit, arrays can be assigned areas on the stack and accessed through a dope vector. Strings and pointers to dynamic aggregates, however, cannot be stored on the stack. They need a heap with routines for heap management and garbage collection supplied by the compiler. Procedure calling conventions and parameter linkage are other issues dependent on storage assignment.

Code generation is the process of mapping some intermediate representation of the source program into assembly or binary machine-code. It is concerned with the selection of addressing modes and instructions to generate code for addressing of variables, expression evaluation and implementation of control constructs. Addressing modes are used to access variables and constants that are bound to storage locations. Depending on program-counter addressing, integer constants need not be bound to storage locations (they can be stored as part of an instruction, e.g. immediate addressing). Addressing

modes such as indexing and auto-increment can often be used to subsume code for explicit addition. The operations (instructions) and data-types of the target machine determine the language operators that are directly executable and those that are to be simulated using a sequence of instructions. For example, on machines with no floating-point hardware, arithmetic on 'reals' are to be simulated in software.

Due to mixed mode arithmetic (coercions specified by compiler front ends) and the possibility of binding a variable to more than one machine data-type (e.g. an integer may be bound to a byte or word), code generators have to emit instructions for data-type conversions. e.g.

(i) C {real} := A {integer} + B {real}
conversion: A {integer} to A {real}

(ii) C {word} := A {byte} + B {word}
conversion: A {byte} to A {word}

Furthermore, for non-orthogonal instruction-sets (orthogonality is discussed in a later paragraph), code has to be generated for addressing mode conversions. e.g. no memory-to-memory arithmetic is possible on the Intel 8086 and Z8000,

C {memory} := A {memory} + B {memory}
conversion: A {memory} to A {register}

The abstractions of hardware (machine implemented primitives) essential for code generation are data-types, addressing modes and instructions. Data types are groups of bits that can participate as operands to instructions (e.g. byte, word, longword). The interpretation of these groups of bits by

the central processing unit depends on their representation (e.g. sign magnitude, 2's complement). Data types of the processors under comparison are:

Z8000	bit, byte, word, longword, quadword, BCD
Intel 8086	byte, word, BCD
MC68000	bit, byte, word, longword, BCD

These architectures do not fully support data types for business oriented applications and string operations.

Addressing modes are access paths to retrieve operands (which are data types residing in storage locations such as memory, stack, or register). The time taken to access an operand depends on the access path and the storage location in which the operand resides (e.g. it is faster to retrieve an operand from a register than from memory or the stack). The size (space occupied) of a machine instruction is also determined by the addressing mode. The addressing modes available for the 3 processors are summarized below:

8086	Z8000	MC68000
immediate	immediate	immediate
reg direct	reg direct	reg direct
reg indirect	reg indirect	reg indirect
reg indirect indexed	reg indirect indexed	reg indirect indexed
reg indirect w/ offset	reg indirect w/ offset	reg indirect w/ offset
reg indirect indexed with offset	reg indirect indexed with offset	reg indirect indexed with offset
-	reg indirect with post-decrement	reg indirect pre-decrement
-	reg indirect with post-increment	reg indirect with post-increment
absolute	absolute	absolute
-	absolute indexed	-
-	relative	relative
-	-	relative indexed with offset

The MC68000 has a more complete set of addressing modes. On the Z8000,

only 10% of the instructions can have all addressing modes. Thus, it has a highly non orthogonal addressing scheme. When migrating from the segmented version to the non-segmented version, a change in the addressing modes is required and three more instructions are to be generated. The 8086 has a very irregular register structure. The stack pointer cannot be used as an index register. When multiplying two variables, the AX and DX registers must be used. The MC68000 has asymmetric address and data registers. Due to pin limitations, only 24 bit addresses are supported.

The 8086 has a segmented (non-uniform) address space. Intra-segment subroutine calls do not place the segment address on the stack. Therefore, there are two separate return instructions: an inter-segment return and an intra-segment return. Also, subroutines must always be called either via inter-segment calls or via intra-segment calls. This choice is left to the compiler writer.

The operations of machines can broadly be classified (with respect to mapping source-language operators to machine op-codes) under the following categories:

(1) Data-transfer (move) instructions are used in implementing source language assignments to variables. Some assignments can be subsumed as part of other operations (e.g. a := a + b can be implemented as add b, a). Assignments of aggregates may not be implementable in a single data-transfer instruction; often a series of "move"s or a loop is required to implement them. The Intel 8086 and Z8000 have block transfer instructions that can be used to implement such aggregate assignments. Data-transfer instruc-

tions are also used by code generators for among other things, assignment of temporaries and altering addressing modes of operands.

(2) Arithmetic instructions are used to implement arithmetic-expression evaluation and address calculations. Non-commutative operations must be provided with both varieties of operands (e.g. A := A - B, A := B - A). All the three architectures lack this requirement.

(3) Boolean instructions are used to implement Boolean-expression evaluation under two contexts: (a) as RHS of assignment statements and (b) as predicates to control constructs. Therefore, there should be instructions that (a) provide an explicit boolean result and (b) allow a branch to a location depending on the boolean condition. None of these processors provide both these varieties.

(4) Control instructions (compare and branches) are used to implement relational operators (sometimes an implicit comparison with zero) where the result decides the control flow of the user program (usually the result of comparison is a condition-code setting).

(5) Procedure call and return instructions are used to implement procedure (subroutine) calls and returns. The 8086 provides very minimal support for subroutine entry and exit. It does not have a multiple register save and restore instruction. Thus, procedure calls and returns generate additional instructions for each register saved and restored. The Z8000 has a multiple load instruction. It has a return-and-add to the stack pointer instruction that pops the callee's arguments from the stack. The MC68000 allows a variable number of registers to be saved during a context switch, thus minimizing the overhead during a procedure call.

(6) Special instructions are not essential to code generation per se but their use generally results in optimized object code. Some examples are:

(a) combination of arithmetic and control operations implementable in a single instruction (e.g. loop, loope, loopne and repeat prefixes for instructions on the 8086).

(b) shift operations on integers (often used to replace multiplication by a power of two). The 8086 hardware does not provide multiple shifts.

(7) Instructions for parallel processing and operating system functions (e.g. traps and interrupt instructions). The 8086 and the MC68000 provide instruction support for traps and exceptions.. Some more support is needed for unimplemented instruction traps on the 8086.

By design, the 8086 is very space efficient. It has both byte and word offsets, 8-bit operands, signed short immediates and byte encoded instructions. The Z8000 and the MC68000 have 16-bit offsets only. Furthermore, on the Z8000, the instructions must be multiple word lengths. Byte efficiency is lost if instructions are to be word aligned.

The orthogonality of an instruction set is the regularity with which any op-code (without data-size encoding within the opcode itself) can be used with any machine-primitive data-type and addressing mode. The orthogonality of the instruction set makes the architecture easy to learn and program. It reduces the time required to write programs but may result in lower code density. Irregularities adversely affect code-generation efficiency. All three architectures have some degree of non-orthogonality. For example, on the Intel 8086, the definition of many instructions implies the use of specific registers as

well as certain addressing modes. The 8086 hardware does not allow push, multiply and divide with immediate addressing mode. On the Z8000 and the Intel 8086, no memory-to-memory arithmetic is possible. If both operands are in memory locations then one of them must be moved to a register before an operation is performed. The 8086 does not have any instruction to test the contents of a memory location (the location has to be explicitly compared with zero). On the 8086 and the MC68000, when loading part of a register, extra code must be generated to sign extend or zero fill the upper byte. On all three processors, condition codes are not defined for some instructions. The MC68000 is intended to be a 32-bit architecture (it has 32-bit address and data registers) but it does not support 32-bit division and multiplication. Furthermore, there are no bit manipulation instructions. It is hard to perform mixed data width operations on the MC68000 as there are no instructions for conversion between certain data types. Although it provides an auto-increment addressing mode, no auto-increment can be done in memory. These examples are some of the pains that the compiler writer must go through when generating code for the 8086, Z8000 or the MC68000.

Addressing-mode and data-type conversions are used by code generators to implement source-language operations with incompatible machine data-types. These architectures differ in the type of special instructions they provide and in the degree of orthogonality of the instruction set. If a sufficient set of conversion possibilities is not provided by the architecture or if they cannot be synthesized by the compiler writer, the code generator may be unsuccessful when trying to generate code for valid input.

[4] Operating system support

The purpose of an operating system is to control the resources of a system and assist in the execution of application programs. Resources are those facilities, hardware and software (e.g. central processors, memories, I/O devices, compilers) that are needed to do the work required of the system by its users. Most systems facilitate the sharing of their resources. Primary memory, disk files, and the central processor can all be shared by different programs. This sharing must be controlled by the operating system to prevent unintentional and/or unauthorized use and/or destruction. In order to share resources more efficiently, most systems provide a large amount of concurrent activity at a number of different levels: terminal handling for many simultaneous users; transfer of information to and from I/O devices may take place at the same time the program is being executed by the central processor; a number of different user jobs may be partially completed; and multiple central processors may be in operation. These examples are just a few that are typical of concurrent activity in modern operating systems.

A large part of an operating system is generally dedicated to causing direct actions in the hardware (e.g. I/O transfers). Most operating systems have the responsibility to control the communication with the external world through a potentially large number of different devices.

There are a number of services provided by a modern operating system that assist in the execution of user programs. These services include I/O, error recovery, interrupt coordination, and memory allocation. The main functions that an operating system does in a multiprogrammed environment include

scheduling of the central processing unit for servicing different tasks, loading information into memory and providing protection so that multiple jobs can share memory, control access to shared I/O devices, and many other bookkeeping details necessary to keep track of multiple jobs and control shared resources.

Single Processor Operating System Support

Previous generations of microprocessors were limited by the available technology at that time. Such microprocessors, in particular the Intel 8080 and the Motorola 6800, were limited by the number of registers, width of the data-paths, size of the address space, and capabilities of the instruction set because technology could not support more features on a single chip. These microprocessors are mainly used in process control applications (e.g. intelligent device controllers) and small, single-user stand-alone systems. The 8086, MC68000, and Z8000 microprocessors with their extended capabilities, have many attributes that belong to minicomputers of the 1970s. Therefore, they can be considered for more applications including modern time-sharing and multiprogramming environments. To facilitate classical single processor operating systems it is important to support multiprogramming with good hardware support for task switching, flexible interrupt and trap structure, memory management, flexible I/O capabilities, and at least two execution modes of privilege.

In order to design reliable and efficient operating systems with clean user interfaces, the existence of both user and supervisor modes of execution privilege is crucial. An operating system, when executing in supervisor mode, can execute privileged instructions, access all resources, and perform the

overhead tasks for user-mode (application) programs. Programs executing in user mode have their access controlled and so their effect on other parts of the system is limited. Thus, the supervisor mode is a mechanism for providing protection and security in a computer system. The 8086 does not have two such modes of execution privilege - a major drawback for its use in time-sharing systems. However, most of its deficiencies in protection needs are remedied by Intel's follow-on product: the iAPX-286 [Childs 80a, 80b]. The Z8000 and MC68000 have both user and supervisor modes of execution privilege. For both processors, the supervisor mode is entered during all trap, interrupt, and reset operations where the operating system can handle them appropriately. Each mode has its own stack and stack pointer. Having two sets of stack pointers facilitates task switching when interrupts or traps occur. The user stack is kept clean of system information, since the information saved on the occurrence of interrupts or traps is always saved on the system (supervisor) stack before the appropriate program status is loaded. This technique can allow the development of a cleaner system in that the user stack is kept clear of system information.

User processes usually communicate via special instructions with operating systems that provide I/O or other shared functions. The trap instruction is used on the MC68000 and the SC (system call) instruction is used on the Z8000.

Of the three microprocessors, the MC68000 seems to have the most extensive internal error detection trap mechanism. Hardware traps and interrupts are provided to detect abnormal internal and external conditions that are of

major importance in an operating system in order to provide error recovery and security in the computer system. The MC68000 and 8086 have instructions (TRAPV and INTO respectively) to cause a trap on an overflow condition. The ability to support a large number of interrupts favors the development of large multi-user operating systems because more peripherals can be supported. The interrupt and trap structure described in the hardware section of this paper shows that the MC68000 is best suited for larger systems.

The extent of the I/O capabilities of a microprocessor is important when considering the machine for multi-user operating systems. Since computer systems may have many different types of peripherals, it is important that microprocessors have flexible I/O capabilities. The MC68000's asynchronous bus can handle peripherals of different speeds. The local busses of the 8086 and the Z8000 are synchronous and consequently faster. Compatibility with peripherals of different speeds is achieved via the system bus (Intel's Multibus). All three processors are expected to have interfaces to the Multibus.

The 8086 and Z8000 can have their I/O devices either memory or port mapped. The MC68000 only supports memory mapped I/O. A disadvantage of port mapped I/O is that it uses more opcodes and adds to the complexity of the microprocessor. The advantage of memory mapped I/O is that one can use the large repertoire of instructions and addressing modes that operate in the memory space instead of just a few I/O instructions. The disadvantages of memory mapped I/O are:

(1) there are fewer addresses available in memory space,
(2) a number of memory locations are reserved, and
(3) memory mapped I/O complicates cache management.

Certain instructions found in the 8086 and Z8000 are useful for handling blocks of data. For example, The Z8000 has block input and output instructions. These instructions reduce kernel code size and speed up block I/O transfers by eliminating an explicit loop. Often a user program requests a block of data from the operating system. The system commonly transfers the data from the input device into its own data space. The system must then do a memory to memory block transfer to deliver the data to the user program. In the 8086 and Z8000, this function can be implemented with a block transfer or string instruction.

The Intel 8086, Zilog Z8000, and Motorola MC68000 all have extended addressing capabilities. Memory addressing does not limit the usefulness of any of these three microprocessors. Since few current microprocessor based systems address as much as 64 kilobytes, address capabilities should not be much of a factor when considering small dedicated systems. Few single-user microprocessor application programs, except compilers and interpreters use more than 16 kilobytes. The extended addressing capabilities should be a factor in a system competing with high-end minicomputers. In multiprogramming environments one must consider the memory management capabilities in terms of overall system security. The 8086 is probably less useful here. Among other reasons, its address space is inadequate.

A desirable memory protection scheme allows a kernel to easily assign, to data and code segments, attributes such as read only, read-write, and system use. The relocation and memory protection mechanisms for the Z8000 and MC68000 are provided by an external memory management unit. These devices

seem sufficiently capable of managing memory for multiprogrammed operating systems since the address space is segmented. The use of segmentation provides a good solution to the problem of dynamic user memory expansion, relocation, and protection, as well as virtual memory implementation. However, the loss is speed. It appears that in both, the Z8000 and the MC68000, zero wait-state operation with memory management is essentially impossible.

Important functions that an operating system provides in a multiprogrammed environment are scheduling of the central processing unit and controlling accesses to shared resources. In a multiprogrammed system, task switches occur frequently and necessitate the saving of the current process state and loading of another process's state. Task switching requires saving all registers in memory so that they can be restored when the process is restarted. Both the Z8000 and the MC68000 contain a single instruction to move multiple registers to memory and vice-versa. This facility provides an increase in overall system performance since task switching is a frequent operation in a multiprogrammed operating system.

Since many computer resources cannot be reasonably shared simultaneously, some method for mutual exclusion must be enforced. All three microprocessors have test-and-set instructions that can be used to develop semaphores to provide the necessary mutual exclusion. These semaphores can also be used to synchronize cooperating processes.

Multiprocessor Operating System Support

The same reasons that make these 16-bit microcomputers attractive or unattractive for single processor operating systems also apply to multiprocessor systems. A result of the availability of powerful but inexpensive hardware is the current development of multiprocessor operating systems. The underlying idea is to build a multiprocessor system of many inexpensive microprocessors that can equal or exceed the capabilities of a large single processor system at reduced cost, increased reliability, and increased speed through parallelism. A rapid drop in hardware prices coupled with the advance in hardware capabilities makes it no longer necessary to share a central processer and memories to the utmost to achieve an acceptable price/performance ratio. Therefore, inexpensive processors can be dedicated to certain functions and still maintain cost effectiveness. The multiple processor arrangement considered here is the multiple-instruction multiple-data stream (MIMD) computer [Flynn 66]. This computer comprises a number of processors that are connected together by time-shared busses, cross-point switches or multi-port memory modules. Interconnection may be tightly coupled through a shared memory or loosely coupled via communication lines.

There seem to be three basic approaches to multiple processor operating systems. The first approach is a loosely coupled network of general purpose processors. Each processor runs its own unique kernel or the same kernel as the other processors in the network (e.g. Arpanet, Ethernet). However, no operating system functions are divided among two processors. Each processor has the ability to perform all operating system functions. In the second ap-

proach, the operating system is executed on a dedicated processor with application programs. Some operating system functions may be executed on slave processors. In the third approach, the operating system might be distributed across the set of all processors in the system. Recently, the distributed approach has become more popular for the following reasons: (1) it offers higher processing power and throughput, (2) processors are treated as identical system resources by the operating system so that if any single processor fails, the system will continue to operate at a possibly reduced rate (i.e. the system exhibits a fail-soft behavior), (3) distribution tends to make more effective use of the processor resource than when one processor is dedicated to an operating system (i.e. dynamic load balancing is more efficient). Such an approach tends to distribute intelligence towards peripherals resulting in increased modularization. Included in the second class would be intelligent device controllers (e.g. disk controller) where most of the scheduling and resource management duties of the operating system could be off-loaded into the device rather than the main-frame. Screen editors could also fit into this second category. A front-end processor (probably inside the terminal itself) could contain the software for editing as well as have the communication software to talk to the back-end residing in the main-frame computer. The main-frame would need to only handle the file system management of the text file being edited, thus, reducing the load on the main-frame and distributing the work.

With the possibility of off-loading operating system functions and/or system programs into specialized slave processors, tightly coupled multiple processor arrangements seem to be a rather attractive alternative to single

processor operating systems. Processors in such a configuration would need hardware and/or software locking (semaphore) operators in order to mutually exclude other processors from critical resources or communicate via shared memory in a producer-consumer arrangement. All three microcomputers have these hardware capabilities, although each processor provides this capability differently. For example, the MC68000 provides both hardware and software interlocks for multiprocessor systems. The CPU chip contains bus arbitration logic for a shared bus and shared memory environment. Multiprocessor systems are also supported with software instructions (e.g. test-and-set). The TAS (test-and-set) instruction allows a processor to interrogate a test byte in some shared memory, set the condition codes accordingly, and then place a '1' in the most significant bit of the byte. If memory (or another resource) is busy, the processor is denied access until the test byte is cleared. Note that the TAS instruction is an indivisible read-modify-write instruction. In this manner processors can be mutually excluded from a given resource. It should be noted that on the Multibus it is impossible to tell when the bus is locked (the Multibus is used as a system bus by all three processors).

The presence of a large segmented address space is important in shared memory configurations in order to provide greater flexibility and protection. Both the Z8000 and the MC68000 with their external memory management units provide this capability, whereas, the 8086 seems to be deficient in this respect.

Unfortunately, the problems with distributed systems today are mostly software and not hardware. Thus, one should not expect detailed solutions in

the hardware architecture to a software problem that has not yet been solved. Yet some basic capabilities do exist to allow sharing of address spaces. Large segmented address spaces and hardware exclusion mechanisms in conjunction with software provide mutual exclusion and control of shared critical resources.

[5] Conclusions

A microprocessor would gain easy acceptance if it could fit immediately into applications of current 8 and 16-bit microprocessors and at the same time have an advanced architecture that could be expandable to ensure long product lifetime. All three microprocessors meet the first goal easily. They do so with an order of magnitude more throughput than that of other single chip microprocessors. To meet the second goal, the MC68000 and Z8000 designers have departed from the traditional byte oriented microprocessor design and have adopted the more regular architecture of minicomputers.

Of the three microprocessors considered, the Motorola MC68000 seems to be the best suited for multiprogramming and time-sharing applications. It provides good support for addressing and protection needs of software systems. The Z8000 is comparable but its architecture is not as flexible. The 8086 is less sophisticated than the other two, but the primary design goal was to be upward compatible with the 8080/8085 microprocessors. It appeared two years earlier than the other processors. Its successor has appeared earlier than those of the Z8000 and the MC68000.

A normal way of comparing microcomputers is by making some performance evaluations. Attempting to account for differences in performance due to architectural differences between microprocessors is very difficult since most measurement techniques rely on instruction mixes or particular algorithms (e.g. the Gibson Mix). Instruction mixes are based on the frequencies of use of each instruction in a processor. They are usually developed on a base processor and then applied to other target machines. Many different instruction mixes can be developed that characterize different applications. However, evaluations using these mixes that involve different architectures are usually neither very accurate nor representative. The other methods of evaluation (kernels, synthetic jobs, and benchmarks) require that each processor must be programmed. Unfortunately, this observation implies that the results are dependent on the programmer's skills, choice of algorithms for each processor, and the programmer's expertise with the processors in question. Also, if the programs are written in a high level language, then the code generating facilities of the compiler is tested in addition to the processor capabilities.

These products are different in terms of the silicon technology that is being used (the 1980 version of the 8086 can run at a clock rate of 10 Mhz). Performance can be increased by (a) increasing the clock speed and (b) architectural improvements such as microcode support for string operations in the MC68000. People generally select microprocessors for two main reasons. The first is to build hardware products from them. The other use is for developing software for these architectures. Therefore, making a choice among these architectures is really difficult.

Selection of microprocessors also depends on other issues such as support component availability, software and hardware compatibility with existing systems, second sourcing and end-user prices. Probably, these issues far outweigh the differences among the processors as shown in this paper.

Acknowledgements

The help provided by Bob Kasten, Ed Desautels, Charlie Fischer and Don Neuhengen together with the entertainment provided by the staff of the Computer Systems Lab is gratefully appreciated. We also wish to thank Bob Childs, Dick Hodgman and Glen Myers for some of their comments.

References

[Aho 77] A.V. Aho and J.D. Ullman, "Principles of Compiler Design", Addison-Wesley publishing Co., 1977.

[Beatty 74] J.C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code", IBM Journal of Research and Development 18(1): 20-39, January 1974.

[Childs 80a] R. Childs and J. Klebanoff, "iAPX-286 Microprocessor Architecture Overview", Intel International Invitational Technical Symposium, October 1980.

[Childs 80b] R. Childs, "The iAPX-286 Architecture: Memory Management and Protection Model", Intel International Invitational Technical Symposium, October 1980.

[Ganapathi 80] M. Ganapathi, "Retargetable Code Generation and Optimization using Attribute Grammars", PhD dissertation, Technical Report #406, University of Wisconsin - Madison, 1980.

[Gilmore 80] J. Gilmore, ACM Computer Architecture News, Vol. 8 No. 7, 15 December 1980.

[Flynn 66] M.J. Flynn, "Very High-Speed Computing Systems", Proceedings of the IEEE, December 1966.

[Grappel 81] R.D. Grappel and J.E. Hemenway, "A tale of four microprocessors: Benchmarks quantity performance", EDN, April 1 1981.

[Intel 78] MCS-86 User's Manual. Intel Corporation, Pub. No. 9800722A, July 1978.

[Johnsson 75] R.K. Johnsson, "An Approach to Global Register Allocation", PhD dissertation, Carnegie-Mellon University, 1975.

[Motorola 79] MC68000 16-Bit Microprocessor User's Manual. Motorola Inc., September 1979.

[Stritter 79] E. Stritter and T. Gunter, "A microprocessor architecture for a changing world: The Motorola 68000." IEEE Computer, February 1979, pp. 18-27.

[Szymanski 78] T.G. Szymanski, "Assembling Code for Machines with Span-Dependent Instructions", CACM, Vol. 21 No. 4 pp. 300-308, April 1978.

[Szymanski 80] T.G. Szymanski and B. Leverett, "Chaining Span-Dependent Jump Instructions", ACM Transactions on Programming Languages and Systems, Vol. 2 No. 3, 1980.

[Toong 81] H.D. Toong and A. Gupta, "An Architectural Comparison of Contemporary 16-bit Microprocessors", IEEE Micro, Vol. 1 No. 2, pp. 26 - 37, May 1981.

[Zilog 79] Z8001/Z8002 CPU Product Specification. Zilog Inc., Pub. No. 03-8002-01, Preliminary ed., March 1979.

[Peuto 79] Peuto, B.L. "Architecture of a new microprocessor." IEEE Computer, pp. 10 - 21, February 1979.