Microcomputers: An Alternative for Digital Controllers

Richard Brooke Wavell

University of Central Florida
MICROCOMPUTERS: AN ALTERNATIVE FOR DIGITAL CONTROLLERS

BY

RICHARD BROOKE WAVELL
B.S.E., Tulane University, 1971

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Graduate Studies Program of the College of Engineering at the University of Central Florida; Orlando, Florida

Spring Quarter 1979
MICROCOMPUTERS: AN ALTERNATIVE FOR DIGITAL CONTROLLERS

BY

RICHARD B. WAVELL

ABSTRACT

Two candidate microprocessors are selected to demonstrate that microprocessors can be effectively utilized in digital control applications. The Intel 8085A-2 is selected to represent 8-bit microprocessors and the Intel 8086 is selected to represent 16-bit.

A baseline second order digital filter algorithm is developed for comparison of the performance of the two microprocessors for both software and hardware multiply implementations. Emulation techniques are utilized to facilitate performance comparison, without large investments in software and hardware development. In addition, they provide a method for evaluating new microprocessors, before hardware becomes available. The emulations are then used to verify and to determine the sampling frequency realizable with each digital filter implementation.
ACKNOWLEDGMENTS

The author is indebted to Dr. Fred O. Simons for his invaluable counsel and to the management of the Computational Sciences Laboratory of Martin Marietta Aerospace, Orlando Division, for use of their computer systems during this project.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>I. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>II. Background</td>
<td>3</td>
</tr>
<tr>
<td>III. Microcomputer and Software Selection</td>
<td>8</td>
</tr>
<tr>
<td>IV. Digital Filter Implementation</td>
<td>12</td>
</tr>
<tr>
<td>V. Microcomputer Emulation</td>
<td>19</td>
</tr>
<tr>
<td>VI. Microcomputer Evaluation and Conclusions</td>
<td>25</td>
</tr>
<tr>
<td>Appendix A. Intel 8085A-2 and 8086 Emulator Listings and Output</td>
<td>31</td>
</tr>
<tr>
<td>Appendix B. Excerpts From The Intel 8085A-2 User Guide</td>
<td>64</td>
</tr>
<tr>
<td>Appendix C. Excerpts From The Intel 8086 User Guide</td>
<td>73</td>
</tr>
<tr>
<td>References</td>
<td>80</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>Comparison of 8-Bit and 16-Bit Filter Outputs With Floating Point Results</td>
<td>17</td>
</tr>
<tr>
<td>II.</td>
<td>Timing Results of Microprocessor Comparisons</td>
<td>27</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Signal Flow Graph of Standard Form</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>Signal Flow Graph of Canonic Form</td>
<td>5</td>
</tr>
<tr>
<td>3.</td>
<td>Flowchart of Filter Equations</td>
<td>13</td>
</tr>
<tr>
<td>4.</td>
<td>Sampling Frequency vs. Filter Order for Different Digital Filter Implementations</td>
<td>28</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

Since the introduction of the first general purpose 8-bit microprocessor in 1971, microcomputers have become an integral part of a wide variety of applications, ranging from computer terminals to burglar alarm systems. Almost all of these applications share one or more of several basic characteristics. They are either used for logical decision making, arithmetic computations, or data manipulation. Applications which involve microcomputers in the role of digital computers have just begun to be explored. Microcomputers have not seriously been considered as digital controllers since their instruction execution times have only recently become fast enough to elevate them to the status required for any significant amount of real time signal processing. Over the last eight years there has been well over an order of magnitude increase in instruction execution times, which has warranted increasing interest in these devices for digital control purposes. There is almost a limitless number of products where digital control techniques could be employed to improve performance if there were a cost effective way of implementing such techniques. As microcomputers become faster and less costly they will embrace an increasing number of these applications.

It is the purpose of this thesis to investigate the potential role of several state of the art microcomputer systems as real time digital controllers. Its purpose is not to discuss design techniques for digital filter applications [1, 2, 3], but rather to demonstrate the
feasibility of implementing such designs with currently available general purpose microcomputer systems.

Through these examples, general techniques will be developed for analyzing new microcomputer systems as they appear on the market. This can be accomplished without a large investment in hardware and software development, through use of computer emulation techniques.
II. BACKGROUND

In considering implementation of digital filters, an analysis using a general second order transfer function as a baseline is appropriate. An nth order transfer function can always be partitioned into a series of cascaded first and second order transfer functions with real coefficients, through factorization. Performing a partial fraction expansion will result in a summation of first and second order transfer functions with real coefficients. Gold and Rader [1] point out that as the filter order increases there becomes an increasing sensitivity to the values of the filter coefficients. They conclude that implementation with the cascade or parallel combinations of first and second order systems are the best solution to this problem. In addition, many digital controllers contain nonlinear elements which will often result in separating higher order filters into lower order subelements in order to insert the proper nonlinear effects. The second order digital filter is therefore a basic building block for an nth order filter.

The general second order $z$-transform can be represented in the following form:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{z^2 + A_1 z + A_2}{z^2 + B_1 z + B_2} \tag{1.1}$$

There are many forms in which this filter may be implemented, including breaking the second order filter into two cascaded or paral-
First order filters, if the poles are real. The choice of forms is determined by many factors, including pole and zero locations, coefficient values, and other design considerations. Perhaps the most obvious form is the direct form which can be derived as follows:

Solving for \( Y(z) \) yields:

\[
Y(z) = X(z) + X(z)z^{-1}A_1 + X(z)z^{-2}A_2 - Y(z)z^{-1}B_1 - Y(z)z^{-2}B_2 \quad (1.2)
\]

This equation is represented in signal flow form by Figure 1.

The direct form has the disadvantage that its output is a function of two previous values of both its input and output. By defining an intermediate summation point or state, \( P(z) \), it is possible to derive a canonic form whose output, \( Y(z) \), is a function of only two previous values of \( P(z) \).

Multiplying numerator and denominator of equation (1.1) by \( z^{-2} \) yields:

\[
\frac{Y(z)}{X(z)} = \frac{1 + A_1 z^{-1} + A_2 z^{-2}}{1 + B_1 z^{-1} + B_2 z^{-2}} \quad (1.3)
\]

Let us define a state \( P(z) \) such that:

\[
\frac{Y(z)}{P(z)} = 1 + A_1 z^{-1} + A_2 z^{-2} \quad (1.4)
\]

then

\[
Y(z) = P(z) + P(z)z^{-1}A_1 + P(z)z^{-2}A_2 \quad (1.5)
\]

and

\[
\frac{Y(z)}{X(z)} = \frac{\frac{Y(z)}{P(z)}}{1 + B_1 z^{-1} + B_2 z^{-2}} \quad (1.6)
\]

Cross multiplication of equation (1.6) and some manipulation yields:

\[
P(z) = X(z) - P(z)z^{-1}B_1 - P(z)z^{-2}B_2 \quad (1.7)
\]
Figure 1. Signal Flow Graph of Direct Form

Figure 2. Signal Flow Graph of Canonic Form
Equations (1.6) and (1.8) are represented in signal flow form in Figure 2.

This implementation was chosen as the baseline for microprocessor comparisons since it requires a minimal number of operations and delay elements and, thus, should require less computer execution time than others.

If equations (1.5) and (1.7) are inverse transformed to the time domain, the following equations are obtained:

\[ y(nT) = p(nT) + p(nT-T)A_1 + p(nT-2T)A_2 \]  \hspace{1cm} (1.8)

\[ p(nT) = x(nT) - p(nT-T)B_1 - p(nT-2T)B_2 \]  \hspace{1cm} (1.9)

If variables names are assigned to the previous values of \( p \), \( p(nT-T) \) and \( p(nT-2T) \), a set of arithmetic computations suitable for programming on a digital computer can be written. These equations, if executed every \( T \) seconds, will produce values of \( y(nT) \) for a given input \( x(nT) \).

Using upper case variables to maintain compatibility with computer printout, let \( P = p(nT) \), \( S = p(nT-T) \), \( T = p(nT-2T) \), and \( Y = y(nT) \).

Then

\[ P + X - SB_1 - TB_2 \]  \hspace{1cm} (1.10)

and

\[ Y + P + A_1 S + A_2 T \]  \hspace{1cm} (1.11)

\[ T + S \]  \hspace{1cm} (1.12)

\[ S + P \]  \hspace{1cm} (1.13)
Note that equations (1.12) and (1.13) will update the previous values of $P$ in preparation for the next pass through the filter program $T$ seconds later.

A comparative analysis of the implementation of equations (1.10) - (1.13) on different microprocessors, using different available software and hardware techniques, will therefore provide a baseline for judging their relative performance. In addition, since, as discussed previously, the second order digital filter is a basic building block for higher order filters, relative performance can easily be extrapolated for higher order implementations.
III. MICROCOMPUTER AND SOFTWARE SELECTION

Most microprocessors fall into two basic categories; those with 8-bit word lengths and those with 16-bit word lengths. It was, therefore, decided to consider a typical example of each category for digital filter implementations.

Since Intel Corporation introduced the first general purpose 8-bit microprocessor, the 8008, in 1971, it has evolved over the years becoming the 8080 in 1973 and the 8080A in 1975. The 8080 has become the standard of the industry and so it is appropriate to evaluate the newest and fastest version in the 8080 family, the 8085A-2. The 8085A-2 became available only in late 1978 and, as will be demonstrated, has practical capability in digital control applications.

In 1978 Intel also introduced a new 16-bit microprocessor, the 8086, which promises to be as widely used as the 8080. The 8086 is even faster than the 8085A-2 and, due to other features which will be discussed in detail subsequently, it is an excellent choice to represent the capability of 16-bit microprocessors in digital control applications.

It is the purpose of this chapter to describe the Intel 8085A-2 and the Intel 8086 microcomputer systems as they relate to digital control applications only. For a detailed functional description and discussion of system operation, refer to references 4 and 5.
The hardware features and design of microprocessors are only relevant in this discussion if they bear directly on their ability to implement digital filter algorithms. Algorithm execution speed and precision overshadow other factors such as memory addressing or I/O capability. Therefore, in evaluating the performance of microprocessors as digital controllers, the instruction set and their execution times are of primary importance.

The 8085 system has seven general purpose 8-bit registers: A, B, C, D, E, H, and L. Six of these registers may be used in 16-bit pairs as BC, DE, and HL. The 8085 has instructions capable of transferring memory directly through the A register or through the other general purpose registers if the HL register pair is used to specify the destination/source address. The 8085 has eight I/O ports which may be accessed directly through the A register. A flag register is available to indicate the status of results of arithmetic operations. The flag register may then be used to enable subsequent testing and branching. Appendix B contains excerpts of the 8085A instruction set description as used in subsequent discussions.

After each instruction description is listed a number of states. This number indicates the number of clock cycles required to execute that instruction. The 8085A-2 can be driven by a clock with a frequency of up to 5 MHz. Therefore, each clock cycle defines a time interval of 200 nanoseconds. For example, the ADD instruction requires 4 clock cycles or only 800 nsec execution time.

The 8085 has arithmetic instructions to enable 8-bit addition and subtraction and 16-bit double register addition. There is no hardware
multiplying capability. This lack of sophistication has been a primary
deterrent in using general purpose microprocessors to implement digital
filters.

There are two alternatives available. One is to write a software
multiply subroutine. The other is to access a hardware multiply unit
through the I/O ports. The TRW Model TDC1008J performs an 8×8 bit
multiplication in 70 nanoseconds. The multiplier is presented on one
I/O port and the multiplicand on another, using OUT instructions.
Hardware multipliers are generally fast enough to allow the two OUT
instructions to be immediately followed by an IN instruction to obtain
the resulting product. Their primary disadvantage is that they add
complexity and substantial cost to the microcomputer system.

The 16-bit Intel 8086 is not only faster than the 8085A-2, but
also has a much more powerful instruction set.

The 8086 has four general purpose 16-bit registers: AX, BX, CX, and
DX. It is capable of accessing up to 256 I/O ports and has 4 dif-
f erent addressing modes. The primary purpose of three of the address-
ing modes is to allow the addressing of memory through use of a regis-
ter or sum of registers, as a pointer to the source or destination
address. This enables the addressing of up to 1 megabyte of memory.
The 8086 also has a direct addressing capability to or from any of its
general purpose registers. This results in a reduced instruction
count, since a register does not have to be preloaded with a destina-
tion address. In addition, there is no "bottleneck" register, as with
the 8085 which allows direct addressing only with the A register.
Appendix C contains excerpts of the 8086 instruction set description,
as used in subsequent discussions. The 8086 is capable of both 8- and 16-bit addition and subtraction. Its most exciting arithmetic instruction is, however, the multiply instruction.

Although there is no built-in hardware multiply there is a multiply instruction which enables a 16×16 multiply within 130 clock cycles. The multiply instruction is accomplished through a microcoded Read Only Memory (ROM) or firmware approach. This microcoded multiply algorithm enables the elimination of many of the intermediate instruction cycles normally required for execution of software multiplication.

The 8086 system can be driven with a clock frequency of up to 8 MHz, resulting in only 125 nanoseconds per clock cycle. Appendix C lists the number of clock cycles required for each instruction execution. For example, a 16-bit ADD register to register instruction requires 3 clock cycles or 375 nanoseconds for execution. Instructions which access memory must have a number of clock cycles added to the basic execution time, which represent the time required from the addressing mode used. In the case of direct addressing this would require an additional 6 clock cycles or 750 nanoseconds, wherever the symbol EA is indicated in the instruction timing description. Use of a 16×16 hardware multiplier would result in even greater speed. Again the I/O ports could be used to supply the multiplier, multiplicand, and to retrieve the product. The TRW model TDC1010J will perform a 16×16-bit multiplication within 115 nanoseconds, enabling the product formation to be transparent to the 8086 program. As with the 8085, the primary disadvantages of a hardware multiplier are cost and additional interfacing requirements.
IV. DIGITAL FILTER IMPLEMENTATION

As described in chapter I, the second order digital filter provides the basis for implementation of higher order filters and, therefore, is an appropriate baseline for comparison of microprocessor performance. Several implementation techniques will be chosen and the candidate microprocessors discussed in chapter II will be compared with respect to their performance in realizing the second order filter equations, as described in equations (1.3) - (1.6) of chapter I. A flowchart of these equations is shown in Figure 3 as they would be programmed on a digital computer.

Upon examination of the equations to be programmed, an operation count reveals that two additions, two subtractions, and four multiplications are required. If the coefficients $A_1$, $A_2$, $B_1$, and $B_2$ are assumed to be stored in memory then approximately 13 memory accesses are required. Due to the lengthy execution times required for software and firmware multiplies, the multiplication times become the dominant time factor in the overall filter execution speed. In view of this fact, two basic implementations will be considered for the 8085 system. One will use a software multiply routine and the other will make use of the external hardware multiplier described in chapter II. For the 8086, the built in firmware multiply will be considered as one option and an external hardware multiply will be considered for the other.
Figure 3. Flowchart of Filter Equations
There are other options which could have been considered. However, they are either not decisive in terms of performance improvement, or they are very dependent on the poles and zeros of the particular filter being implemented. For example, 16-bit filters could be implemented on the 8085 but, as will be seen subsequently, an increase of execution time by a factor of 4 or 5 normally required for double precision software arithmetic would make their application quite limited.

Higher level programming languages could be used to generate the filter code. Intel has a modified PL-1 language available for the 8085 and 8086 called PL-M. Use of PL-M would allow more rapid coding and debug of digital filters, particularly if complex nonlinear elements were involved. Preliminary instruction counts indicate that use of PL-M results in approximately a 40% increase in execution time for the second order filter described here and for many applications this could be quite tolerable. However, use of PL-M is not really a different implementation technique. It is really a different way to generate the machine code required and will not be considered further here.

Scaling is a consideration which must be made whenever digital filters are being implemented on a fixed point computer. Unfortunately, it is difficult to generalize about its application, as it is highly dependent on both the pole-zero locations and the frequency band of application. To obtain maximum precision it is desirable to scale the filter coefficients as close as possible to unity. If this is done, depending on the range of values of the coefficients and the scale factors of the filter states, this may result in products of coefficients and states which are scaled for different values. At this point
arithmetic shifts may be required to rescale each such product to con-
form to the scale of that of the filter state which represents the
final summation of product terms. For comparative purposes these
scaling considerations might be more important for an 8-bit microproces-
sor than for a 16-bit. Fortunately, the addition of shift instructions
required by scaling would have only a minor impact on the filter's total
execution time. Because the shift instruction deals only with a single
register and does not require memory access it requires 800 nanoseconds
per bit on the 8085 system. It requires 250 nanoseconds for 1 bit or
1 microsecond plus 500 nanoseconds per bit for 2 or more bits on the
8086 system.

To improve precision another option might be to retain the full
16-bit product when using an 8-bit machine. After the sum of products
was formed to obtain the filter state, it could then be shifted appro-
priately and truncated to obtain an 8-bit result. This technique would
avoid memory access of double precision quantities and make use of the
8085 double precision ADD instruction.

For comparative purposes it was decided to select a set of second
order coefficients as an example and to derive the theoretical response
of the filter to a step input. The response could then be scaled and
truncated to the appropriate number of bits, which would serve as a way
of verifying different implementations in only a few passes through the
filter. One pass through the filter code could then be timed and the
results used to determine the sampling frequency realizable. For this
reason, coefficients were selected such that no internal rescaling was
required.
Let:
\[ H(z) = \frac{z^2 + 0.252406z}{z^2 + 0.504812z + 0.367879} \]  \hspace{1cm} (3.1)

If we take \( x(t) = 0.5 u_{-1}(t) \) then forming \( Y(z) = H(z) X(z) \), we obtain
\[ Y(z) = \frac{z}{2(z-1)} \left( \frac{z^2 + 0.252406z}{z^2 + 0.504812z + 0.367879} \right) \]  \hspace{1cm} (3.2)

Performing a partial fraction expansion yields
\[ Y(z) = \frac{0.334387z + 0.110804}{z - 1} + \frac{0.110804 / -41.6412^\circ}{z + 0.252406 - 0.551516j} + \frac{0.110804 / 41.6412^\circ}{z + 0.252406 + 0.551516j} \]  \hspace{1cm} (3.3)

\[ = \frac{0.334387z + 0.110804e^{-0.726776j}}{z - 1} + \frac{0.110804e^{0.726776j}}{(ze^{j2} - e^{-0.5})(ze^{-j2} - e^{0.5})} \]  \hspace{1cm} (3.4)

Taking the inverse z transform, with \( T = 1 \)
\[ y(n) = 0.334387 + 0.221608e^{-0.5n} \cos(-0.726776 + 2.0n) \]  \hspace{1cm} (3.5)

Substituting successive values of \( n \) in the expression for \( y(n) \) yields the following table of \( y(n) \) vs. \( n \) and the 16- and 8-bit scaled equivalents of \( y(n) \).
### TABLE I

Comparison of 8-Bit and 16-Bit Filter Outputs With Floating Point Results

<table>
<thead>
<tr>
<th>n</th>
<th>Floating Point</th>
<th>16-Bit Fixed Point</th>
<th>8-Bit Fixed Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.5</td>
<td>16384</td>
<td>64</td>
</tr>
<tr>
<td>1</td>
<td>.3737</td>
<td>12245</td>
<td>47</td>
</tr>
<tr>
<td>2</td>
<td>.2536</td>
<td>8309</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>.3607</td>
<td>11819</td>
<td>46</td>
</tr>
<tr>
<td>4</td>
<td>.3508</td>
<td>11495</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>.3164</td>
<td>10367</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>.3374</td>
<td>11055</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>.3395</td>
<td>11124</td>
<td>43</td>
</tr>
<tr>
<td>8</td>
<td>.3307</td>
<td>10836</td>
<td>42</td>
</tr>
<tr>
<td>9</td>
<td>.3344</td>
<td>10957</td>
<td>42</td>
</tr>
</tbody>
</table>
Table I provides a way of verifying that the code generated for the four filter implementations and the performance data derived from those implementations is correct. From the response \( y(n) \) obtained, it can be seen that this example is not typical of a second order digital filter. However, the response was chosen to facilitate verification of the filter code and, therefore, frequency domain characteristics were not a consideration. The lightly damped response with a half of full scale output for \( n=1 \) provided the ability to rapidly determine open paths and incorrect gains.

At this point the filter coding must be programmed on the appropriate microprocessors and executed to obtain timing information. The question now arises, "What is the most efficient way of verifying the microprocessor code and determining the sampling frequency realizable?" If the microcomputer hardware has not yet become available or if it is an expensive or time consuming effort to set it up, debug the code, and make the necessary time studies an alternative might be worth considering.

This alternative is the subject of the following chapter.
V. MICROCOMPUTER EMULATION

It was decided that since many candidate microcomputer systems might have to be examined in order to determine the one best suited for a particular real time control application, the time and expense involved in obtaining and interfacing the required hardware could be prohibitive. Even the particular implementations that have been undertaken for comparison on the 8085 and 8086 systems would have been a major hardware and software task.

Instead, the filter coding and timing were checked via emulations of the 8085 and 8086 microprocessors and associated hardware. Emulators essentially enable one computer to behave like another. The emulating computer is programmed to behave identically to and accept instructions of the emulated computer. This concept becomes particularly useful if the emulator itself is written in a higher order language like FORTRAN and can be used on a general purpose computer system. A FORTRAN emulator allows the emulated computer to be emulated on any computer system which has a FORTRAN compiler.

Some microprocessor manufacturers have written emulators for some of their products, however, many microprocessors do not have emulators available although they may appear to be potentially applicable as digital controllers. For example, Intel has an emulator available for the 8085 system but does not have or plan to have one available for the 8086 system in the near future.
It was decided that to demonstrate that emulators could be written in FORTRAN relatively easily, which would enable verification and timing of digital filters, as an important step in facilitating future evaluations of new microcomputer systems before hardware became available.

The emulations presented here do not attempt to emulate the entire instruction set of either the 8085 or 8086 system. Rather, it is intended to show that the instructions required for digital filter coding can be implemented as needed. In the study undertaken, the 8085 system requires a software multiply subroutine as one option. This subroutine requires a larger variety of instructions than is typically required in a digital filter coding. For this reason the emulation of the 8085 system is more complete than that of the 8086 system.

Both the 8085 and 8086 emulators are organized in essentially the same manner. Appendix A contains the source listings of the emulators of both systems and includes the instruction listings of the two filter implementations analyzed for each system. The emulators are each organized into a main program and a number of subroutines, where the main program contains the emulated microprocessor source code (i.e., digital filter code) and each subroutine emulates a separate microprocessor instruction. The emulated source code is written such that each instruction to the microprocessor is written as a call to a subroutine whose name corresponds to that of the instruction desired. The instruction operand fields are represented by and transmitted through the arguments to the call statement.

For example, the instruction MOV M, B in 8085 assembly language means move the contents of register B to the location in memory
specified by the contents of the H and L registers. This would be
coded in the emulator as CALL MOV (M, B). Subroutine MOV would then be
expected to emulate the actions of the MOV instruction. In the emula-
tor program M and B are FORTRAN variables containing values which
specify to the MOV subroutine what specific type of MOV instruction to
perform and on what registers it is to be executed.

In the program there is initially an unlabeled COMMON which is
used to reserve storage for and transmit values of the microprocessor
registers, I/O ports, flags and memory to the instruction emulator sub-
routines. It also stores a time counter which is updated at the end of
each emulator subroutine by the appropriate number of clock cycles,
thus providing accurate timing information. Following the unlabeled
COMMON is COMMON/PRINT/ which is used to transmit flags to the emulator
subroutines in order to signal whether debug print information is
desired and to display the contents of all registers, I/O ports, flags,
and a specified number of memory locations.

Following the COMMONs are the DATA statements which equate the
register, flag, and memory symbolic names with pointers to the COMMON
storage. For example, register names C, D, E, H, L, A, and B are
assigned values 1, 2, 3, 4, 5, 6, and 7, respectively. These values
are used to index into the IREG array within the unlabeled COMMON.
DATA statements are then defined to initialize memory locations. The
DATA statements are in lieu of equate statements, normally required by
an assembler to initialize memory.

The microprocessor filter code then follows the DATA statements.
At the end of the filter code a time test is made to determine whether
or not to terminate the filter program. If the specified time has not
been reached an output print is made and control is transferred to the
beginning of the filter loop for the next pass through. As can be seen
for the JMP instruction, it was not possible to write an emulator sub-
routine which when called would return to a different specified place
in the microprocessor code. This is probably the primary disadvantage
of the FORTRAN emulator. However, by using the JMP subroutine to update
the time counter appropriately it was possible to achieve the desired
result through the insertion of an in line FORTRAN GO TO statement. The
only disadvantage is that it can tend to clutter the microprocessor code
with irrelevant statements. Luckily control transfer is not often
required in filter programs and, therefore, has not presented much
inconvenience.

In the case of the 8085 software multiply filter implementation,
an assembly language subroutine call was required. This was implemented
simply with a direct FORTRAN CALL to a subroutine which contained calls
to instruction emulators achieving a realistic 8085 software multiply
algorithm. This subroutine makes use of both the NC and the z flag and
performs the shifting, adding, and testing to achieve a 16-bit product
of two 8-bit integers.

Both the 8085 and 8086 systems were emulated as digital filters
using hardware multiply devices, as mentioned previously. A hardware
multiply emulator subroutine was called each time an operand appeared
on either of the I/O ports assigned to it for the multiplier and multi-
plicand. The hardware multiply then placed the product on the speci-
fied input port. This is a particularly good example of how a custom
written FORTRAN emulator can be used to great advantage. There are
often times when external devices may wish to be emulated and inter-
faced to the microprocessor. Having complete control over the emulator
code makes this type of study possible since the emulator code can
easily be modified as desired.

Each instruction emulator may contain 10 to 20 FORTRAN statements,
corresponding to as many as 100 instructions in the general purpose host
computer. It may seem to be almost absurd that 100 instructions are
required on one machine to emulate a single instruction on another.
However, it is important to realize that when a higher order language
like FORTRAN is used on a computer whose word length is different from
the microprocessor's that this can result in substantial inefficiency.

To illustrate the techniques utilized, the MOV instruction listed
in Appendix A, page 39, provides a good example. Upon execution the
first task is to distinguish between a register to memory MOV, a memory
to register MOV, and a register to register MOV. M was assigned the
value -1 by arbitrary convention, allowing a testing of the first sub-
routine argument for equality with -1. If the first argument has a
value of -1, then it must be a register to memory MOV. If it is not a
-1, then the second argument is tested for equality with -1 to distin-
guish between the remaining two cases.

Then MEM (ADDRS) becomes the contents of memory at location
ADDRS, specified by register pair HL. If a register to memory MOV has
been specified, then MEM (ADDRS) is set equal to IREG (ARG2). If a
memory to register MOV is specified, IREG (ARG1) is set equal to MEM
(ADDRS).
The other instructions are programmed using similar techniques for both the 8085 and 8086 emulators.
VI. MICROCOMPUTER EVALUATIONS AND CONCLUSIONS

The filter implementations proposed in Chapter III for the baseline second order model described in Chapter I of the Intel 8085A-2 and 8086 microcomputer systems were verified and timed using the emulation techniques described in Chapter IV. The source listings of the hardware multiply approaches for the Intel 8085A-2 and the firmware and hardware multiply approaches for the Intel 8086 are contained in Appendix A, in addition to the emulator subroutines for each microprocessor. Also included is an abbreviated output listing from each approach. The output gives the time in clock cycles, at the end of each pass through the filter, as well as the contents of the registers, I/O ports, and specified memory locations. The filter output is found on I/O port 1.

The step responses can then be verified against the ideal response as tabulated in Table I of Chapter III. The number of clock cycles between each pass through the filter can then be used to determine the sampling frequency realizable for each implementation.

The 8085A-2 software multiply implementation requires approximately 2130 clock cycles per pass. 2130 clock cycles multiplied by 200 nanoseconds per clock cycle is equivalent to 0.426 millisecond per pass or a sampling frequency of 2.350 KHz. The 8085A-2 hardware multiply implementation can be evaluated similarly for 313 clock cycles per pass, resulting in a sampling frequency of 16.0 KHz.
The 8086 firmware multiply implementation can be evaluated with a clock interval of 125 nanoseconds, for 654 clock cycles per pass. This is equivalent to a sampling frequency of 12.2 KHz. The hardware multiply implementation results in sampling frequency of 26.5 KHz for 300 clock cycles per pass. These results are tabulated in Table II.

As can be seen the addition of a hardware multiply unit results in far greater speed improvement than for the 8085A-2 than for the 8086. This is, of course, because the 8086 has a microcoded firmware multiply available. The 8086 holds an advantage when comparing both microcomputers interfaced to hardware multipliers. This is due to the faster instruction execution times available with the 8086 and its more powerful instruction set.

The 8086 would seem to be the favored choice, whether or not its full 16-bit precision was required. However, the 8085A-2 has been available in similar, through slower, form for a number of years and is much better understood and more widely used than the newly released 8086. At this time the 8086 is considerably more expensive than the 8085A-2 and software and hardware are not as widely available.

For low frequency applications where 8-bits precision may be enough the 8085A-2, with a software multiply algorithm, may be sufficient. If more speed is required, the added cost of the hardware multiplier may not justify using the 8085A-2 if the 8086 is found to be adequate, with its firmware multiply capability.

In Chapter III examination of the assembly code required to generate the second order baseline model, for each approach revealed that the primary calculation being performed is a sum of products. It was
TABLE II
Timing Results of Microprocessor Comparisons

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Intel 8085A-2</th>
<th>Intel 8086</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Frequency (MHz)</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Filter Implementation</td>
<td>Hardware Multiply</td>
<td>Software Multiply</td>
</tr>
<tr>
<td>Clock Cycles Required</td>
<td>313</td>
<td>2130</td>
</tr>
<tr>
<td>Maximum Sampling Frequency (KHz)</td>
<td>16.0</td>
<td>2.35</td>
</tr>
<tr>
<td>Maximum Signal Frequency (Hz) (Based on 20 samples per cycle)</td>
<td>800</td>
<td>120</td>
</tr>
</tbody>
</table>
Figure 4. Sampling Frequency vs. Filter Order for Different Digital Filter Implementations
described in Chapter I that higher order filters are most effectively built up by cascading first and second order filters to obtain the order desired.

From these two facts it is then possible to determine the sampling frequency realizable for higher order digital filters. The filter order is roughly proportional to its execution time and, therefore, is usually inversely proportional to its realizable sampling frequency. This relationship may then be extended to each microprocessor implementation and the relationship between sampling frequency and filter order can be plotted, as shown in Figure 4.

Figure 4 can now be used as a tool for microprocessor selection. The filter order desired is first selected on the abscissa. The sampling frequency realizable for each microprocessor implementation can then be found on the ordinate. Any implementation which is capable of a sampling frequency at least as high as required can then be selected.

It has been demonstrated that there are 8-bit and 16-bit microprocessors available which are capable of sampling frequencies of up to 26.5 KHz for second order digital filter implementations. This computational speed is fast enough that many state-of-the-art digital autopilots could easily be implemented on the 8086. For example, a typical digital autopilot with 6 first order filters and 2 second order filters could be implemented with a sampling frequency of 2.6 KHz. This compares extremely favorably with the sampling frequencies in the 0.1 - 1.0 KHz range required in many current missile systems.
It has been further shown that custom emulators, written in FORTRAN, are an effective way of evaluating microprocessor performance and verifying and timing microprocessor code. The initial filter implementations, written in 8085 and 8086 assembly language, were about 20 - 50% slower than the versions presented here. Through use of the emulations and study of the 8085 and 8086 characteristics, several iterations were made in the coding to reduce the execution times to current levels.

These emulation techniques can be applied to new microprocessors as software descriptions become available. This would allow additional plots to be added to Figure 4. It can then be determined if any performance advantage is offered by the new microprocessor, prior to any commitment to extensive software and hardware development.
APPENDIX A

INTEL 8085A-2 AND 8086 EMULATOR LISTINGS AND OUTPUT
THE FOLLOWING PROGRAM IS AN EMULATOR FOR MANY OF THE INSTRUCTIONS
OF THE INTEL 8085 MICROPROCESSOR. IT IS WRITTEN IN FORTRAN FOUR AND
IS COMPATIBLE WITH MOST FORTRAN COMPILERS. THE MAIN PROGRAM CONTAINS
MEMORY AND REGISTER LOCATION DEFINITIONS AND THE MICROPROCESSOR
PROGRAM, WRITTEN IN INTEL 8085 ASSEMBLY LANGUAGE, THE SUBROUTINES
CONTAIN THE EMULATOR IMPLEMENTATION AS WELL AS A DEBUG SUBROUTINE TO
DISPLAY REGISTER, I/O PORT, AND MEMORY VALUES.

THE ONLY LIBERTY THAT IS TAKEN FROM STANDARD 8085 ASSEMBLY CODE IS
THAT EACH INSTRUCTION IS PRECEDED BY A CALL STATEMENT AND THAT THE
ARGUMENTS ARE ECONOMIZED IN PARENTHESIS. JUMP STATEMENTS ALSO
REQUIRE A FORTRAN COMPUTED GO TO STATEMENT AFTER THEM TO TRANSFER
PROGRAM EXECUTION

COMMON I,FG(7),I0BLK(8),TIME,MEM(256),IFLAG(4)
COMMON PRINT/,IFLOMP,MEMLOC
INTEGER B,C,D,E,H,L,A
INTEGER COEF1,COEF2,COEF3,COEF4,TEMP1,TEMP2,TEMP3,
,STATES,STATET,PTEMP,Z,CY,F,S
SET IFLOMP = FOR JUMP OF ALL REGISTER,PORTS AND MEMORY AFTER EACH
INSTRUCTION, SET IFLOMP=0 FOR DISPLAY ONLY WHERE DEBUG IS CALLED
IN SOURCE CODE
SET MEMLOC = NUMBER OF MEMORY LOCATIONS TO BE DISPLAYED IN DEBUG PRINT
DATA IFLOMP /0/,MEMLOC/6/
DEFINE REGISTER ADDRESSES
DATA R/6/,C/1/,D/2/,E/3/,I/4/,L/5/,A/7/
DEFINE FLAGS
DATA MZ/7/,Z/1/,NC/2/,CA/3/,PO/4/,PE/5/,P/6/,M/7/
Equate symbols with memory locations
THIS IS IN LIEU OF ASSEMBLY LOCATE STATEMENTS
DATA TEMP1/1/,TEMP2/2/,TEMP3/3/,STATES/4/,STATET/5/,PTEMP/6/,
# COEFA1/7, COEFA2/8, COEFB1/9, COEFB2/10

C * INDICATES MEMORY REFERENCE TO ASSEMBLER.
DATA M/-1/

C INITIALIZE MEMORY LOCATIONS AND TIME COUNTER
DATA MEM(7)/32/MEM(8)/7/MEM(9)/64/MEM(10)/47/

C START OUTPUT ON NEW PAGE
WRITE(6,100)
100 FORMAT(1H1)

C DEFINE CONSTANT INPUT ON PORT 0 TO OBTAIN STEP RESPONSE
10BLK(1) = 64

C SET CLOCK CYCLE COUNTER EQUAL ZERO
ITIME = 0

C THE FOLLOWING IS ACTUAL INTEL 8085 CODE FOR A DIGITAL
C FILTER IMPLEMENTATION
C THIS IMPLEMENTATION IS A GENERAL SECOND ORDER LEAD/ LAG WHICH
C ASSUMES THE AVAILABILITY OF A HARDWARE MULTIPLY WITH INPUTS ON
C PORTS 2 AND 3 AND OUTPUT BEING AVAILABLE ON PORT 4

C INITIALIZE ACCUMULATOR TO 0
CALL MVI (A,0)

C ZERO STATES
CALL STA (STATES)

C ZERO STATET
CALL STA (STATET)

C BEGINNING OF REAL TIME LOOP
C COMPUTE STATES * COEFB1 (S#1)
1 CALL MVI (L,STATES)
CALL MOV (C,M)
CALL MVI (L,COEFB1)
CALL MOV (U,M)
CALL MPY
CALL MVI (L,TEMP1)
CALL MOV (M,B)

C FORM STATET * COEFB2 (T#B2)
CALL MVI (L,STATET)
CALL MOV (C,M)
CALL MV1 (L,COEFB2)
CALL MOV (U,M)
CALL MPY
CALL MV1 (L,TEMP2)
CALL MOV (M,B)
C INPUT X AT PORT 0
CALL IN (0)
C FORM X = STATES*COEFH1 - STATET = COEFB2 (X-S*B1)
CALL MV1 (L,TEMP1)
CALL SUR (M)
C FORM X - STATES*COEFH1 - STATET = COEFB2 (X-S*B1-T*B2)
CALL MV1 (L,TEMP2)
CALL SUR (M)
C STORE IN PTEMP (P)
CALL STA (PTEMP)
C FORM COEFH1 = STATES (S*A1)
CALL MV1 (L,STATES)
CALL MOV (C,M)
CALL MV1 (L,COEFH1)
CALL MOV (U,M)
CALL MPY
CALL MV1 (L,TEMP3)
CALL MOV (M,B)
C FORM STATET = COEFH1 (T*A2)
CALL MV1 (L,STATET)
CALL MOV (C,M)
CALL MV1 (L,COEFH2)
CALL MOV (U,M)
CALL MPY
CALL MOV (A,B)
C FORM STATET = COEFH1 + STATES - COEFH1 (T*A2+S*A1)
CALL ADD (M)
C FORM STATET = COEFH1 + STATES - COEFH1 + PTEMP (T*A2+S*A1+P)
CALL MV1 (L,PTEMP)
CALL ADD (M)
C OUTPUT RESULT TO PORT 1
CALL OUT (1)
C UPDATE STATES TO STATET AND PTM TO STATES
CALL LDA (STATES)
CALL STA (STATET)
CALL LDA (PTM)
CALL STA (STATES)
C JUMP TO START OF LOOP FOR NEXT PASS
CALL JMP (LOOP)

C END OF 8085 SOURCE CODE
C
C CALL DEBUG PRINT TO MONITOR FILTER OUTPUT
CALL DEBUG(2)
C A STATEMENT IS PUT HERE TO TERMINATE FILTER PROGRAM AFTER 5000
C CLOCK CYCLES
IF (ITIME, GT, 250000) CALL EXIT
GO TO 1
END

SUBROUTINE DEBUG(NARG1)
C DISPLAY SUBROUTINE TO PRINT MEMORY LOCATIONS, REGISTERS, AND I/O PORTS
COMMON IREG(7), IOHLK(8), ITIME, MEM(256), IFLAG(4)
COMMON PRINT/, IFLDMP, MEMLOC
IF ((NARG1, EQ, 1), AND, (IFLDM, EQ, 0)) RETURN
WRITE(6, 100) ITIME
100 FORMAT(1H, 110, 25H NUMBER OF CLOCK CYCLES)
WRITE(6, 200) IREG(1), I=1, 7
200 FORMAT(1H, 10X, 35H CONTENTS OF REGISTERS C, D, E, F, L, R, A, 1X, 719)
WRITE(6, 250) IFLAG(1), I=1, 4
250 FORMAT(1H, 19X, 26H CONTENTS OF FLAGS Z, CY, P, S, 1X, 415)
SUBROUTINE MPY
C SOFTWARE MULTIPLY SUBROUTINE
C COMMON IREFG(7),10BLK(B),ITIME,MEM(256),IFLAG(4)
C DEFINE REGISTER ADDRESSES
DATA MZ/C7/7/1/,NC/2/.,CA/3/.,PD/4/.,PE/5/.,P/6/.,M/7/
DATA M/-1/
INTEGER DONE
DATA B/6/.,C/1/.,D/2/.,E/3/.,F/4/.,L/5/.,A/7/
C TIME OVERHEAD FOR CALL STATEMENT
ITIME = ITIME +18
C INITIALIZE MOST SIGNIFICANT BYTE OF RESULT
CALL MV1 (B,0)
C INITIALIZE HIT COUNTER
CALL MV1 (E,8)
C ROTATE LEAST SIGNIFICANT BIT OF MULTIPLIER TO CARRY AND SHIFT
1 CALL MOV (A,C)
C LOW ORDER BYTE OF RESULT
CALL RAR
CALL MOV (C,A)
CALL OCR (E)
C EXIT IF COMPLETE
CALL J (7,DONE)
GO TO (10,20), DONE
10 CALL MOV (A,B)
CALL J (NC,MULT1)
GO TO (38,40), MULT1
C ADD MULTIPICAND TO HIGH ORDER BYTE OF RESULT IF BIT WAS A ONE
32 CALL ADD (D)
C CARRY = 0 HERE SO SHIFT HIGH ORDER BYTE OF RESULT
42 CALL PAR
   CALL MOV (B,A)
   CALL JMP (MULTU)
   GO TO 1
22 CALL RET
RETURN
END

SUBROUTINE DCR (ARG1)
INTEGER ARG1
COMMON IREG(7), IORLK(8), ITIME, MEM(256), IFLAG(4)
IREG(ARG1)=IREG(ARG1)-1
IF (IREG(ARG1), GT, 127) IREG(ARG1)=127
IF (IREG(ARG1), LT, -127) IREG(ARG1)=-127
IFLAG(4)=0
IF (IREG(ARG1), GT, 3) IFLAG(4)=1
IFLAG(1)=0
IF (IREG(ARG1), EQ, 0) IFLAG(1)=1
ITIME=ITIME+4
CALL DEBUG(1)
RETURN
END

SUBROUTINE PAR
COMMON IREG(7), IORLK(8), ITIME, MEM(256), IFLAG(4)
ITEMP=IREG(7)/2
JTEMP=ITEMP*2
KTEMP=0
IF (IFLAG(2), EQ, 1) KTEMP=64
IFLAG(2)=1
IF (IREFG7), F0, JTEMP) IFLAG(2) = 0
IREFG7 = ITEMP + KTEMP
ITIME = ITIME + 4
CALL DEBUG(1)
RETURN
END

SUBROUTINE FJ (ARG1, ARG2)
COMMON IREFG7, I0RLK(8), ITIME, MEM(256), IFLAG(4)
INTEGER ARG1, ARG2
ITEMP = ARG1 + 1
GO TO (10, 20, 10, 20, 10, 20, 10, 20), ITEMP
10 IADR = (ARG1 + 2) / 2
ARG2 = 2
ITIME = ITIME + 10
IF (IFLAG(IADR), EQ. 1) ARG2 = 1
IF (IFLAG(IADR), EQ. 1) ITIME = ITIME - 5
GO TO 30
20 IADR = ITEMP / 2
ARG2 = 1
ITIME = ITIME + 7
IF (IFLAG(IADR), EQ. 1) ARG2 = 2
IF (IFLAG(IADR), EQ. 1) ITIME = ITIME + 3
30 CALL DEBUG(1)
RETURN
END

SUBROUTINE HMPLY
C HARDWARE MULTIPLY SUBROUTINE
COMMON IREFG7, I0RLK(8), ITIME, MEM(256)
I0RLK(5) = I0RLK(3) * ICBLK(4) / 127
RETURN
END

E 38
SUBROUTINE MOV (ARG1, ARG2)
COMMON IREG(7), 10BLK(8), ITIME, MEM(256)
INTEGER ARG1, ARG2, ADDRES
IF (ARG1, NF, -1) GO TO 10
ADDRES = IREG(4) * 256 + IREG(5)
MEM(ADDRES) = IREG(ARG2)
ITIME = ITIME + 7
CALL DEBUG(1)
RETURN

10 IF (ARG2, NF, -1) GO TO 20
ADDRES = IREG(4) * 256 + IREG(5)
IREG(ARG1) = MEM(ADDRES)
ITIME = ITIME + 7
CALL DEBUG(1)
RETURN

20 IREG(ARG1) = IREG(ARG2)
ITIME = ITIME + 4
CALL DEBUG(1)
RETURN
END

SUBROUTINE MVI (ARG1, ARG2)
COMMON IREG(7), 10BLK(8), ITIME, MEM(256)
INTEGER ARG1, ARG2, ADDRES
IF (ARG1, EO, -1) GO TO 10
IREG(ARG1) = ARG2
ITIME = ITIME + 7
CALL DEBUG(1)
RETURN

10 ADDRES = IREG(4) * 256 + IREG(5)
MEM(ADDRES) = ARG2
ITIME = ITIME + 10
CALL DERUG(1)
RETURN
END

SUBROUTINE OUT(ARG1)
COMMON IREG(7), IOBLK(8), ITIME, MEM(256)
INTEGER ARG1
IOBLK(ARG1+1) = IREG(7)
C CHECK FOR A CHANGE OF STATES ON PORTS 2 OR 3. IF EITHER IS ADDRESSED THEN
C UPDATE HARDWARE MULTIPLY
IF((ARG1.EQ.2).OR.(ARG1.EQ.3)) CALL HMPY
ITIME = ITIME +10
CALL DERUG(1)
RETURN
END

SUBROUTINE LOA(ARG1)
COMMON IREG(7), IOBLK(8), ITIME, MEM(256)
INTEGER ARG1
IREG(7) = MEM(ARG1)
ITIME = ITIME +13
CALL DERUG(1)
RETURN
END

SUBROUTINE IN(ARG1)
COMMON IREG(7), IOBLK(8), ITIME, MEM(256)
INTEGER ARG1
IREG(7) = IOBLK(ARG1+1)
ITIME = ITIME +10
CALL DERUG(1)
RETURN
SUBROUTINE SUB(ARG1)
COMMON IREG(7), IOHLK(8), ITIME, MEM(256), IFLAG(4)
INTEGER ARG1, ADDRES
IF(ARG1.EQ.-1) GO TO 10
IREG(7) = IREG(7) - IREG(ARG1)
ITIME = ITIME + 4
GO TO 20
10 ADDRES = IREG(4) * 256 + IREG(5)
IEREG(7) = IREG(7) * MEM(ADDRES)
ITIME = ITIME + 7
20 IFLAG(2) = 0
IF ((IREG(7).GT.127), OR, (IREG(7).LT.-127)) IFLAG(2) = 1
IF (IREG(7).GT.127) IREG(7) = 127
IF (IREG(7).LT.-127) IREG(7) = -127
IFLAG(1) = 1
IF (IREG(7).NE.0) IFLAG(1) = 0
IFLAG(4) = 1
IF (IREG(7).GT.0) IFLAG(4) = 0
CALL DEBUG(1)
RETURN
END

SUBROUTINE RET
COMMON IREG(7), IOHLK(8), ITIME, MEM(256), IFLAG(4)
ITIME = ITIME + 10
CALL DEBUG(1)
RETURN
END

SUBROUTINE ADD(ARG1)
COMMON IREG(7), I0RLK(8), ITIME, MEM(256), IFLAG(4)
INTEGER ARG1, AUDRES
IF (ARG1, EQ, -1) GO TO 10
IREG(7) = IREG(7) + IREG(ARG1)
ITIME = ITIME + 4
GO TO 20
10 AUDRES = IREG(4) = 256 + IREG(5)
IREG(7) = IREG(7) + MEM(AUDRES)
ITIME = ITIME + 7
20 IFLAG(2) = 0
IF ((IREG(7), GT, 127), OR, (IREG(7), LT, -127)) IFLAG(2) = 1
IF (IREG(7), GT, 127) IREG(7) = 127
IF (IREG(7), LT, -127) IREG(7) = -127
IFLAG(1) = 1
IF (IREG(7), NE, 0) IFLAG(1) = 0
IFLAG(4) = 1
IF (IREG(7), GT, 0) IFLAG(4) = 0
CALL DERUG(1)
RETURN
END

SUBROUTINE JMP(ARG1)
COMMON IREG(7), I0RLK(8), ITIME, MEM(256)
ITIME = ITIME + 10
CALL DERUG(1)
RETURN
END

SUBROUTINE STA(ARG1))
COMMON IREG(7), I0RLK(8), ITIME, MEM(256)
INTEGER ARG1
MEM(ARG1) = IREG(77)
ITIME = ITIME + 13
CALL PERUC(1)
RETURN
END

<table>
<thead>
<tr>
<th>FILTER STEP RESPONSE (SOFTWARE MULTIPLY)</th>
</tr>
</thead>
</table>

2154 NUMBER OF CLOCK CYCLES
CONTENTS OF REGISTERS C, O, E, H, L, P, A: 0 0 0 0 6 0 64
CONTENTS OF FLAGS Z, CY, F, S: 0 0 0 0
CONTENTS OF I/O PORTS 0, 1, 2, 3, 4, 5, 6, 7: 64 64 0 0 0 0 0
CONTENTS OF MEM LOCATIONS 1-MEMLOC: 0 0 0 64 0 64

4277 NUMBER OF CLOCK CYCLES
CONTENTS OF REGISTERS C, O, E, H, L, P, A: 0 0 0 0 6 0 32
CONTENTS OF FLAGS Z, CY, F, S: 0 0 0 0
CONTENTS OF I/O PORTS 0, 1, 2, 3, 4, 5, 6, 7: 64 40 0 0 0 0 0
CONTENTS OF MEM LOCATIONS 1-MEMLOC: 32 0 16 32 64 32

6492 NUMBER OF CLOCK CYCLES
CONTENTS OF REGISTERS C, O, E, H, L, P, A: 0 0 0 0 6 0 25
CONTENTS OF FLAGS Z, CY, F, S: 0 0 0 0
CONTENTS OF I/O PORTS 0, 1, 2, 3, 4, 5, 6, 7: 64 33 0 0 0 0 0
CONTENTS OF MEM LOCATIONS 1-MEMLOC: 16 23 8 25 32 25
<table>
<thead>
<tr>
<th>NUMBER OF CLOCK CYCLES</th>
<th>CONTENTS OF REGISTERS C, D, E, H, I, F, A</th>
<th>CONTENTS OF FLAGS Z, CY, F, S</th>
<th>CONTENTS OF IO PORTS 0, 1, 2, 3, 4, 5, 6, 7</th>
<th>CONTENTS OF MEM LOCATIONS 1-MEMLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>231</td>
<td>0  0  0  0  0  6  0  41</td>
<td>0  0  0  0</td>
<td>0  47  0  0  0  0  0  0</td>
<td>0  41  25  41</td>
</tr>
<tr>
<td>60384</td>
<td>0  0  0  0  0  6  0  35</td>
<td>0  0  0  0</td>
<td>0  45  0  0  0  0  0  0</td>
<td>0  41  35  35</td>
</tr>
<tr>
<td>12797</td>
<td>0  0  0  0  0  6  0  32</td>
<td>0  0  0  0</td>
<td>0  40  0  0  0  0  0  0</td>
<td>0  35  32  32</td>
</tr>
<tr>
<td>14926</td>
<td>0  0  0  0  0  6  0  36</td>
<td>0  0  0  0</td>
<td>0  44  0  0  0  0  0  0</td>
<td>0  36  32  36</td>
</tr>
<tr>
<td>17653</td>
<td>0  0  0  0  0  6  0  35</td>
<td>0  0  0  0</td>
<td>0  44  0  0  0  0  0  0</td>
<td>0  36  36  36</td>
</tr>
<tr>
<td>19184</td>
<td>Number of Clock Cycles</td>
<td>Number of Registers C,D,E,H,L,R,A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------</td>
<td>----------------------------------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>Contents of Flags Z,CY,F,S</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Contents of IO Ports 0,1,2,3,4,5,6,7</td>
<td>04</td>
<td>42</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Contents of MEM Locations 1-MEMLOC</td>
<td>17</td>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>21315</th>
<th>Number of Clock Cycles</th>
<th>Number of Registers C,D,E,H,L,R,A</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>6</th>
<th>0</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contents of Flags Z,CY,F,S</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contents of IO Ports 0,1,2,3,4,5,6,7</td>
<td>64</td>
<td>43</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contents of MEM Locations 1-MEMLOC</td>
<td>17</td>
<td>12</td>
<td>8</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>23446</th>
<th>Number of Clock Cycles</th>
<th>Number of Registers C,D,E,H,L,R,A</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>6</th>
<th>0</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contents of Flags Z,CY,F,S</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contents of IO Ports 0,1,2,3,4,5,6,7</td>
<td>64</td>
<td>43</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contents of MEM Locations 1-MEMLOC</td>
<td>17</td>
<td>12</td>
<td>8</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>25579</th>
<th>Number of Clock Cycles</th>
<th>Number of Registers C,D,E,H,L,R,A</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>6</th>
<th>0</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contents of Flags Z,CY,F,S</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contents of IO Ports 0,1,2,3,4,5,6,7</td>
<td>64</td>
<td>43</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contents of MEM Locations 1-MEMLOC</td>
<td>17</td>
<td>12</td>
<td>8</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>
THE FOLLOWING PROGRAM IS AN EMULATOR FOR MANY OF THE INSTRUCTIONS
OF THE INTEL 8085 MICROPROCESSOR. IT IS WRITTEN IN FORTRAN FOUR AND
IS COMPATIBLE WITH MOST FORTRAN COMPILERS. THE MAIN PROGRAM CONTAINS
MEMORY AND REGISTER LOCATION DEFINITIONS AND THE MICROPROCESSOR
PROGRAM, WRITTEN IN INTEL 8085 ASSEMBLY LANGUAGE, THE SURROUNDTINES
CONTAIN THE EMULATOR IMPLEMENTATION AS WELL AS A DEBUG SUBROUTINE TO
DISPLAY REGISTER, I/O FORT, AND MEMORY VALUES.
THE ONLY LIBERTY THAT IS TAKEN FROM STANDARD 8085 ASSEMBLY CODE IS
THAT EACH INSTRUCTION IS PRECEDED BY A CALL STATEMENT AND THAT THE
ARGUMENTS ARE ENCLOSED IN PARENTHESES. JUMP STATEMENTS ALSO
REQUIRE A FORTRAN COMPLETED GO TO STATEMENT AFTER THEM TO TRANSFER
PROGRAM EXECUTION

COMMON IREG(7), I0BLK(8), I1IME, MEM(256), IFLAC(4)
COMMON/PRINT, IFLOMF, MEMLOC
INTEGER H, C, D, E, H, L, A
INTEGER COEF <-1, COEFA2, COEFB1, COEFH2, COEFC1, TEMP1, TEMP2, TEMP3,
* STATES, STATFT, PTERM, Z, CY, P, S

SET IFLOMF = 1 FOR JUMP OF ALL REGISTER, PORTS AND MEMORY AFTER EACH
INSTRUCTION, SET IFLOMF = 0 FOR DISPLAY ONLY WHERE DEBUG IS CALLED

IN SOURCE CODE

SET MEMLOC = NUMBER OF MEMORY LOCATIONS TO BE DISPLAYED IN DEBUG PRINT

DATA IFLOMF /W/, MEMLOC/Z6/

DEFINE REGISTER ADDRESSES
DATA R/Z6/, C/Z7/, D/Z7/, E/Z7/, F/Z7/, L/Z7/, A/Z7/
DEFINE FLAG ADDRESSES
DATA /1/,CY/2/,P/3/,S/4/
C EQUATE SYMBOLS WITH MEMORY LOCATIONS
C THIS IS IN LIEU OF ASSEMBLY EQUATE STATEMENTS
DATA TEMP1/1/,TEMP2/2/,TEMP3/3/,STATES/4/,STATET/7/,PTEMP/6/
*COEF1/7/,COEF2/7/,COEF3/9/,COEF4/10/
C M INDICATES MEMORY REFERENCE TO ASSEMBLER.
DATA M/7/
C INITIALIZE MEMORY LOCATIONS AND TIME COUNTER
DATA MEM(7)/62/,MEM(8)/0/,MEM(9)/64/,MEM(10)/4/
C START OUTPUT ON NEW PAGE
WRITE(6,100)
100 FORMAT(1H1)
C DEFINE CONSTANT INPUT ON PORT 0 TO OBTAIN STEP RESPONSE
1DBLK(1) = 64
C SET CLOCK CYCLE COUNTER EQUAL ZERO
TIME = 0
C THE FOLLOWING IS ACTUAL INTEL 5085 CODE FOR A DIGITAL
C FILTER IMPLEMENTATION
C THIS IMPLEMENTATION IS A GENERAL SECOND ORDER LEAD/LAG WHICH
C ASSUMES THE AVAILABILITY OF A HARDWARE MULTIPLY WITH INPUTS ON
C PORTS 2 AND 3 AND OUTPUT BEING AVAILABLE ON PORT 4
C INITIALIZE ACCUMULATOR TO 0
CALL MVI (A,0)
C ZERO STATES
CALL STA (STATES)
C ZERO STATET
CALL STA (STATET)
C ZERO H ADDRESS REGISTER TO ENABLE USE OF SINGLE BYTE ADDRESSES
CALL MVI (H,0)
C BEGINNING OF REAL TIME LOOP
C COMPUTE STATES = COEF1 (S+1)
1 CALL LOA (STATES)
CALL OUT (2)
CALL LOA (COEF1)
CALL OUT (3)
CALL IN (4)
C STORE IN REGISTER B
   CALL MOV (B,A)
C FORM STATET * COEFB2 (T*A2)
   CALL LDA (STATET)
   CALL OUT (2)
   CALL LDA (COEFB2)
   CALL OUT (3)
   CALL IN (4)
C STORE IN REGISTER C
   CALL MOV (C,A)
C INPUT X AT PORT 0
   CALL IN (8)
C FORM X - STATES*COEFU1 ( X=A1)
   CALL SUB (B)
C FORM X - STATES*COEFU1 - STATET*COEFB2 ( Y=SB1-Ta2)
   CALL SUB (C)
C STORE IN REGISTER C (F)
   CALL MOV (C,A)
C FORM COEFU1 = STATES (S*A1)
   CALL LDA (STATES)
   CALL OUT (2)
   CALL LDA (COEFU1)
   CALL OUT (3)
   CALL IN (4)
C STORE IN REGISTER B
   CALL MOV (B,A)
C FORM STATET = COEFU2 (T*A2)
   CALL LDA (STATET)
   CALL OUT (2)
   CALL LDA (COEFU2)
   CALL OUT (3)
   CALL IN (4)
C FORM STATET = COEFU2 + STATES* COEFU1 (Ta2+S*A1)
   CALL ADD (B)
C FORM STATET = COFFA2 + STATES + COFFA1 + NTEMP (T=A2+S+A1+p)
    CALL ADD (C)
C OUTPUT RESULT TO PORT 1
    CALL OUT (1)
C UPDATE STATES TO STATET AND FEGC TO STATES
    CALL LDA (STATET)
    CALL STA (STATET)
    CALL MOV (L,STATET)
    CALL MOV (H,C)
C JUMP TO START OF LOOP FOR NEXT PASS
    CALL JMP (1)
C
C END OF 8085 SOURCE CODE
C
C CALL DEBUG PRINT TO MONIT0R FILTER OUTPUT
    CALL DEBUG(0)
C A STATEMENT IS PUT HERE TO TERMINATE FILTER PROGRAM AFTER 5000
C CLOCK CYCLES
    IF(NTIME,RT,5000) CALL EXIT
    GO TO 1
END

8085 FILTER STEP RESPONSE (HARDWARE MULTIPLY)

366 NUMBER OF CLOCK CYCLES
CONTENTS OF REGISTERS C,D,E,H,L,F,A 64 0 0 0 4 0 0
CONTENTS OF FLAGS Z,CY,P,S 0 0 0 0 0
CONTENTS OF IO PORTS 0,1,2,3,4,5,6,7 64 64 0 0 0 0 0 0
CONTENTS OF MEM LOCATIONS 1-MEMLOC 0 0 0 64 0 0 0
### 692 Number of Clock Cycles

| Contents of Registers C, D, E, H, I, F, A | 32 | 0 | 0 | 0 | 4 | 16 | 64 |
|Contents of Flags Z, CY, F, S | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7 | 64 | 48 | 0 | 0 | 0 | 0 | 0 |
|Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 32 | 64 | 0 | 0 |

### 1018 Number of Clock Cycles

| Contents of Registers C, D, E, H, I, F, A | 25 | 0 | 0 | 0 | 4 | 8 | 32 |
|Contents of Flags Z, CY, F, S | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7 | 64 | 33 | 64 | 0 | 0 | 0 | 0 |
|Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 22 | 32 | 0 | 0 |

### 1344 Number of Clock Cycles

| Contents of Registers C, D, E, H, I, F, A | 41 | 0 | 0 | 0 | 4 | 6 | 25 |
|Contents of Flags Z, CY, F, S | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7 | 64 | 47 | 32 | 0 | 0 | 0 | 0 |
|Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 41 | 25 | 0 | 0 |

### 1670 Number of Clock Cycles

| Contents of Registers C, D, E, H, I, F, A | 55 | 0 | 0 | 0 | 4 | 10 | 41 |
|Contents of Flags Z, CY, F, S | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7 | 64 | 45 | 25 | 0 | 0 | 0 | 0 |
|Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 35 | 41 | 0 | 0 |

### 1996 Number of Clock Cycles

| Contents of Registers C, D, E, H, I, F, A | 32 | 0 | 0 | 0 | 4 | 8 | 35 |
|Contents of Flags Z, CY, F, S | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7 | 64 | 40 | 41 | 0 | 0 | 0 | 0 |
|Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 32 | 35 | 0 | 0 |
### 2322 Number of Clock Cycles
- **Contents of Registers C, D, E, H, L, F, A**: 35 0 0 0 4 8 32
- **Contents of Flags Z, CY, P, S**: 0 0 0 0
- **Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7**: 64 44 35 0 0 0 0
- **Contents of Mem Locations 1-MEMLOC**: 0 0 0 36 32 0

### 2648 Number of Clock Cycles
- **Contents of Registers C, D, E, H, L, F, A**: 35 0 0 0 4 9 36
- **Contents of Flags Z, CY, P, S**: 0 0 0 0
- **Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7**: 64 44 32 0 0 0 0
- **Contents of Mem Locations 1-MEMLOC**: 0 0 0 35 36 0

### 2974 Number of Clock Cycles
- **Contents of Registers C, D, E, H, L, F, A**: 34 0 0 0 4 8 35
- **Contents of Flags Z, CY, P, S**: 0 0 0 0
- **Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7**: 64 42 36 0 0 0 0
- **Contents of Mem Locations 1-MEMLOC**: 0 0 0 34 35 0

### 3304 Number of Clock Cycles
- **Contents of Registers C, D, E, H, L, F, A**: 35 0 0 0 4 8 34
- **Contents of Flags Z, CY, P, S**: 0 0 0 0
- **Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7**: 64 43 35 0 0 0 0
- **Contents of Mem Locations 1-MEMLOC**: 0 0 0 32 34 0

### 3626 Number of Clock Cycles
- **Contents of Registers C, D, E, H, L, F, A**: 35 0 0 0 4 8 35
- **Contents of Flags Z, CY, P, S**: 0 0 0 0
- **Contents of 10 Ports 0, 1, 2, 3, 4, 5, 6, 7**: 64 43 34 0 0 0 0
- **Contents of Mem Locations 1-MEMLOC**: 0 0 0 35 35 0
THE FOLLOWING PROGRAM IS AN EMULATOR FOR MANY OF THE INSTRUCTIONS
OF THE INTEL 8086 MICROPROCESSOR. IT IS WRITTEN IN FORTRAN FOUR AND
IS COMPATIBLE WITH MOST FORTRAN COMPILERS. THE MAIN PROGRAM CONTAINS
MEMORY AND REGISTER LOCATION DEFINITIONS AND THE MICROPROCESSOR
PROGRAM. WRITTEN IN INTEL 8086 ASSEMBLY LANGUAGE, THE SUBROUTINES
CONTAIN THE EMULATOR IMPLEMENTATION AS WELL AS A DEBUG SUBROUTINE TO
DISPLAY REGISTER, I/O, FORT, AND MEMORY VALUES.
THE ONLY LIBERTY THAT IS TAKEN FROM STANDARD 8086 ASSEMBLY CODE IS
THAT EACH INSTRUCTION IS PRECEDED BY A CALL STATEMENT AND THAT THE
ARGUMENTS ARE ENCLOSED IN PARENTHESES. JUMP STATEMENTS ALSO
REQUIRE A FORTRAN COMPILED GO TO STATEMENT AFTER THEM TO TRANSFER
PROGRAM EXECUTION

COMMON IREG(4), IORLK(256), ITIME, MEM(256)
COMMON /PRINT/, FFLOPP, MEMLOC
EXTENDED IREG, IORLK, ITIME, MEM

DEFINE REGISTER ADDRESSES
DATA AX/1/, RX/2/, BX/3/, DX/4/

EQUATE SYMBOLS WITH MEMORY LOCATIONS
DATA TEMP1/207/, TEMP2/257/, TEMP3/264/, STATES/265/, STATET/266/
*TEMP/267/, COFFA1/268/, COFFA2/269/, COFFB1/270/, COFFB2/271/

INITIALIZE MEMORY LOCATIONS AND TIME COUNTER
DATA MEM/(77)/527/, MEM(0)/1/, MEM(9)/16538/, MEM(10)/12055/
DATA FFLOPP/1/, FFLOPP/1/
DIMENSION IADR(517)
EQUIVALENCE (IADR(1), IREG(1))
IORLK(1)=16554
ITIME=0

INITIALIZE ACCUMULATOR TO 0
CALL MOV (AX, 0)
C ZERO STATES
   CALL MOV (STATES, AX)

C ZERO STATET
   CALL MOV (STATET, AX)

C BEGINNING OF REAL TIME LOOP

C COMPUTE STATES * COEFF1 (S#P1)
   1 CALL MOV (AX, STATES)
   CALL OUTW (3)
   CALL MOV (AX, COEFF1)
   CALL OUTW (4)
   CALL INW (2)
   CALL MOV (BX, AX)

C FORM STATET * COEFF2 (T#B2)
   CALL MOV (AX, STATET)
   CALL OUTW (3)
   CALL MOV (AX, COEFF2)
   CALL OUTW (4)
   CALL INW (2)
   CALL MOV (CX, AX)

C INPUT X AT PORT A
   CALL INW (0)

C FORM X - STATES*COEFF1 X-S#B1
   CALL SUB (AX, AX)

C FORM X - STATES*COEFF1 - STATET * COEFF2 (X-S#B1-T#B2)
   CALL SUB (AX, CX)

C STORE IN DTEMP (P)

C FORM COEFF1 * STATES (S#A1)
   CALL MOV (TEMP, AX)
   CALL MOV (AX, STATES)
   CALL OUTW (3)
   CALL MOV (AX, COEFF1)
   CALL OUTW (4)
   CALL INW (2)
   CALL MOV (CX, AX)

C FORM STATET * COEFF2 (T#A2)
CALL MOV (AX,STATET)
CALL OUTX (4)
CALL MOV (AX,COFFA2)
CALL OUTX (4)
CALL INX (5)
C FORM STATET + COFFA2 + STATES = COFFA1 (T*A2+S*A1)
CALL ADD (AX,CX)
C FORM STATET + COFFA2 + STATES + COFFA1 + PTEMP = T*A2+S*A1+P
CALL ADD (AX,TEMP)
C OUTPUT RESULT TO PORT 1
CALL OUTX (1)
C UPDATE STATES TO STATET AND PTEMP TO STATES
CALL MOV (AX,STATES)
CALL MOV (STATET,AX)
CALL MOV (AX,TEMP)
CALL MOV (STATES,AX)
C JUMP TO START OF LOOP FOR NEXT PASS
CALL JMP (LOOP)
C
C END OF 8086 SOURCE CODE
C
C A STATEMENT IS PUT HERE TO TERMINATE FILTER PROGRAM AFTER 5000
C CLOCK CYCLES
C
CALL BEMUG(0)
IF (ITIME,GT,100000) CALL EXIT
IF (S=3)
GO TO 1
END

SUBROUTINE INX (AX)
EXTENDED IADR,ITIME,IFEC,IOBLK,4EH
COMMON IADR(517)
DIMENSION IREC(4),IOBLK(256),MEM(226)
EQUIVALENCE (IADR(261),ITIME),(IADR(9),IOBLK(1)),IADR(262),MEM(1)
* INTEGER ARG1
  TEMP=ARG1+1
  IADR(1)=IOBLK(TEMP)
  ITIME=ITIME+1
  CALL DEBUG(1)
  RETURN
END

SUBROUTINE OUTM(ARG1)
EXTENDED IADR,ITIME,IREG,IOBLK,MEM
COMMON IADR(91)
DIMENSION IREG(4),IOBLK(256),MEM(256)
EQUIVALENCE (IADR(261),ITIME),(IADR(5),IOBLK(1)),(IADR(262),MEM(1))
*
INTEGER ARG1
  TEMP=ARG1+1
  IOBLK(TEMP)=IADR(1)
  IF (((ARG1,EQ.4).OR.(ARG2,EQ.3)) CALL HMPY
  ITIME=ITIME+1
  CALL DEBUG(1)
  RETURN
END

SUBROUTINE HMPLY
C HARDWARE MULTIPLY SUBROUTINE
EXTENDED IREG,IOBLK,ITIME,MEM
COMMON IREGS(4),IOBLK(256),ITIME,MEM(256)
IOBLK(6)=IOBLK(4)*IOBLK(5)/32768
IF(IOBLK(6).GT.32767)IOBLK(6)=32767
IF(IOBLK(A).LT.-32767)IOBLK(A)=-32767
RETURN
END

SUBROUTINE ADD(ARG1,ARG2)
EXTENDED IADR, ITIME, IREG, IORLK, MEM
COMMON IADR(517)
DIMENSION IREG(4), IORLK(256), MEM(256)
EQUIVALENCE (IADR(261), ITIME), (IADR(2), IORLK(1)), (IADR(262), MEM(1))

INTEGER ARG1, ARG2
IADR(ARG1) = IADR(ARG1) + IADR(ARG2)
IF (IADR(ARG1), GT, 32767) IADR(ARG1) = 32767
IF (IADR(ARG1), LT, -32767) IADR(ARG1) = -32767
IF ((ARG1, GT, 4), OR, (ARG2, GT, 4)) GO TO 29
ITIME = ITIME + 3
GO TO 10
20 IF (ARG1, LE, 4) ITIME = ITIME + 15
IF (ARG2, LE, 4) ITIME = ITIME + 22
10 CALL DEBUG(1)
RETURN
END
SUBROUTINE SUR (ARG1, ARG2)
EXTENDED IADR, ITIME, IREG, IORLK, MEM
COMMON IADR(517)
DIMENSION IREG(4), IORLK(256), MEM(256)
EQUIVALENCE (IADR(261), ITIME), (IADR(2), IORLK(1)), (IADR(262), MEM(1))

INTEGER ARG1, ARG2
IADR(ARG1) = IADR(ARG1) - IADR(ARG2)
IF (IREG(4), GT, 32767) IREG(4) = 32767
IF (IREG(4), LT, -32767) IREG(4) = -32767
IF ((ARG1, GT, 4), OR, (ARG2, GT, 4)) GO TO 29
ITIME = ITIME + 3
GO TO 10
23 IF (ARG1, LE, 4) ITIME = ITIME + 15
IF (ARG2, LE, 4) ITIME = ITIME + 22
10 CALL DEBUG(1)
RETURN
END

SUBROUTINE IMP(LOOP)
EXTENDED IADR, ITIME, [REG, IORLK, MEM
COMMON IADR(317)
DIMENSION [REG(4), IORLK(256), MEM(256)
EQUIVALENCE (IADR(261), ITIME), (IADR(5), IORLK(1)), (IADR(262), MEM(1)
* )
ITIME=ITIME+2
CALL DERUG(1)
RETURN
END

SUBROUTINE MOV(ARG1, ARG2)
COMMON IADR(317)
DIMENSION [REG(4), IORLK(256), MEM(256)
EQUIVALENCE (IADR(261), ITIME), (IADR(5), IORLK(1)), (IADR(262), MEM(1)
* )
INTEGER ARG1, ARG2
IF (ARG2,MF,8) GO TO 10
IADR(ARG1)=ARG2
ITIME=ITIME+4
GO TO 20
10 IADR(ARG1)=IADR(ARG2)
IF ((ARG1,MF,4).OR.(ARG2,GT,4)) GO TO 30
ITIME=ITIME+2
GO TO 20
30 IF (ARG1,EO,1).OR.(ARG2,EO,1)) GO TO 15
GO TO 16
15 ITIME=ITIME+4
GO TO 20
16 IF (ARG1,GT,4) GO TO 17
ITIME=ITIME+15
GO TO 20
17 ITIME=ITIME+14
CALL DEFRG(1)
RETURN
END

SUBROUTINE MUL (ARG1)
EXTERNAL IADR, ITIME, IREG, I0RLK, MEM
COMMON IADR(917)
DIMENSION IREG(4), I0RLK(256), MEM(256)
EQUIVALENCE (IADR(261), ITIME), (IADR(2), I0RLK(1)), (IADR(262), MEM(1))
EQUIVALENCE (IADR(1), IREG(1))
INTEGER ARG1
IREG(4)=IADR(ARG1)+1, IREG(1), 32768
IF (IREG(4), 61, 32767) IREG(4)=32767
IF (IREG(4), 11, -32767) IREG(4)=-32767
ITIME=ITIME+1
CALL DEFNC(1)
RETURN
END

C THIS SUBROUTINE PROVIDES DISPLAY OF MEMORY, REGISTERS AND I/O PORTS
SUBROUTINE DEFNC(NARG1)
COMMON IREG(4), I0RLK(256), ITIME, MEM(256)
COMMON /PRINT/ IFLOMP, MEMLOC
EXTERNAL IREG, I0RLK, ITIME, MEM
IF ((NARG1, EQ, 1) .AND. (IFLOMP, EQ, 1)) RETURN
WRITE(6, 100) ITIME
100 FORMAT(1H, 10X, 11H, 2SH NUMBER OF CLOCK CYCLES)
WRITE(6, 200) IREG(1), I=1, 4
200 FORMAT(1H, 10X, 33H CONTENTS OF REGISTERS AX, BX, CX, DX, 3X, 41B)
WRITE(6, 300) I0RLK(1), I=1, 8
300 FORMAT(1H, 10X, 33H CONTENTS OF 10 PORTS 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
WRITE(6, 400) MEM(1), I=1, MEMLOC
400 FORMAT(1H, 10X, 34H CONTENTS OF MEM LOCATIONS 1-MEMLOC, 2X, 101B)
IF (NARG1, EQ, 1) WRITE(6, 500)
8086 FILTER STEP RESPONSE (HARDWARE MULTIPLY)

326 NUMBER OF CLOCK CYCLES
CONTENTS OF REGISTERS AX, BX, CX, DX 16384 0 0 0
CONTENTS OF IO PORTS 0, 1, 2, 3, 4, 5, 6, 7 16384 16384 0 0 0
CONTENTS OF MEM LOCATIONS 1-MEMLOC 0 0 0 16384 0 16384

628 NUMBER OF CLOCK CYCLES
CONTENTS OF REGISTERS AX, BX, CX, DX 8115 8269 4135 0
CONTENTS OF IO PORTS 0, 1, 2, 3, 4, 5, 6, 7 16384 12250 0 0 0
CONTENTS OF MEM LOCATIONS 1-MEMLOC 0 0 0 8115 16384 8115

930 NUMBER OF CLOCK CYCLES
CONTENTS OF REGISTERS AX, BX, CX, DX 6262 4095 2048 0
CONTENTS OF IO PORTS 0, 1, 2, 3, 4, 5, 6, 7 16384 8310 0 16384 0 0
CONTENTS OF MEM LOCATIONS 1-MEMLOC 0 0 0 6262 8115 6262

1232 NUMBER OF CLOCK CYCLES
CONTENTS OF REGISTERS AX, BX, CX, DX 10239 3160 1580 0
CONTENTS OF IO PORTS 0, 1, 2, 3, 4, 5, 6, 7 10239 11819 0 8115 0
CONTENTS OF MEM LOCATIONS 1-MEMLOC 0 0 0 10239 6262 10239
| 1534 Number of Clock Cycles | Contents of Registers AX, BX, CX, DX | 6914 | 5167 | 2584 | 0 |
|----------------------------|------------------------------------|------|------|------|---|---|
| Contents of IO Ports 0, 1, 2, 3, 4, 5, 6, 7 | 10384 | 11493 | 0 | 6262 | 0 | 0 |
| Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 8914 | 10239 | 8914 |

| 1836 Number of Clock Cycles | Contents of Registers AX, BX, CX, DX | 8120 | 4498 | 2249 | 0 |
|----------------------------|------------------------------------|------|------|------|---|---|
| Contents of IO Ports 0, 1, 2, 3, 4, 5, 6, 7 | 10384 | 10369 | 0 | 10239 | 0 | 0 |
| Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 8120 | 8914 | 8120 |

| 2138 Number of Clock Cycles | Contents of Registers AX, BX, CX, DX | 9007 | 4898 | 2049 | 0 |
|----------------------------|------------------------------------|------|------|------|---|---|
| Contents of IO Ports 0, 1, 2, 3, 4, 5, 6, 7 | 10384 | 11056 | 0 | 8914 | 0 | 0 |
| Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 9007 | 8120 | 9007 |

| 2440 Number of Clock Cycles | Contents of Registers AX, BX, CX, DX | 8852 | 4549 | 2273 | 0 |
|----------------------------|------------------------------------|------|------|------|---|---|
| Contents of IO Ports 0, 1, 2, 3, 4, 5, 6, 7 | 10384 | 11125 | 0 | 8120 | 0 | 0 |
| Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 8852 | 9007 | 8852 |

| 2742 Number of Clock Cycles | Contents of Registers AX, BX, CX, DX | 9604 | 4467 | 2234 | 0 |
|----------------------------|------------------------------------|------|------|------|---|---|
| Contents of IO Ports 0, 1, 2, 3, 4, 5, 6, 7 | 10384 | 10838 | 0 | 8907 | 0 | 0 |
| Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 8604 | 852 | 8604 |

| 3044 Number of Clock Cycles | Contents of Registers AX, BX, CX, DX | 8786 | 4342 | 2171 | 0 |
|----------------------------|------------------------------------|------|------|------|---|---|
| Contents of IO Ports 0, 1, 2, 3, 4, 5, 6, 7 | 10384 | 19957 | 0 | 8852 | 0 | 0 |
| Contents of Mem Locations 1-MEMLCC | 0 | 0 | 0 | 8786 | 8604 | 8786 |
C 8056 FIRMWARE MULTIPLY IMPLEMENTATION (SOURCE CODE ONLY)
    CALL MOV (AX,0)
C ZERO STATES
    CALL MOV (STATES,AX)
C ZERO STATET
    CALL MOV (STATET,AX)
C BEGINNING OF REAL TIME LOOP
C COMPUTE STATES * COEFH1 (S#1·1)
    1 CALL MOV (AX,STATES)
    CALL MUL (COEFH1)
    CALL MOV (BX,DX)
C FORM STATET * COEFB2 (T#B2)
    CALL MOV (AX,STATET)
    CALL MUL (COEFH2)
    CALL MOV (CX,DX)
C INPUT X AT PORT 0
    CALL INW (0)
    CALL SUR (AX,RX)
C FORM STATES * COEFH1 - STATET * COEFH2       (SR1-T#P2)
    CALL SUR (AX,CX)
C STORE IN PTEMP (P)
C FORM COEF A1 * STATES (S#A1)
    CALL MOV (TEMPP,AX)
    CALL MOV (AX,STATES)
    CALL MUL (COEFA1)
CALL MOV (CX, DX)
C FORM STATET * COFFA2 (T*A2)
CALL MOV (AX, STATET)
CALL MUL (COFFA2)
C FORM STATET * COFAP + STATES * COFIA1 (T*A2+S*A1)
CALL ADD (DX, AX)
C FORM STATET * COFFA2 + STATES * COFIA1 + PTEMP (T*A2+S*A1+P)
CALL ADD (DX, TEMPP)
CALL MOV (AX, DX)
C OUTPUT RESULT TO PORT 1
CALL OUTW (1)
C UPDATE STATES TO STATET AND PTEMP TO STATES
CALL MOV (AX, STATES)
CALL MOV (STATET, AX)
CALL MOV (AX, TEMPP)
CALL MOV (AX, TEMPP)
C JUMP TO START OF LOOP FOR NEXT PASS
CALL JMP (LOOP)

8086 FILTER STEP RESPONSE (FIRMWARE MULTIPLY)

<table>
<thead>
<tr>
<th>Clock Cycles</th>
<th>AX, BX, CX, DX</th>
<th>16384</th>
<th>0</th>
<th>0</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contents of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 Ports:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contents of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New Location</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-HEXTAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clock Cycles</th>
<th>AX, BX, CX, DX</th>
<th>16384</th>
<th>0</th>
<th>0</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contents of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 Ports:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contents of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New Location</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-HEXTAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clock Cycles</th>
<th>AX, BX, CX, DX</th>
<th>16384</th>
<th>0</th>
<th>0</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contents of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 Ports:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contents of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New Location</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3316 NUMBER OF CLOCK CYCLES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTENTS OF REGISTERS AX, BX, DX, 114</td>
<td>8252</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTENTS OF 16 PORTS 0, 1, 2, 3, 4, 5, 6, 7</td>
<td>10394</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTENTS OF ALL LOCATIONS 1-SEP LPC</td>
<td>0</td>
<td>0</td>
<td>8313</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3323 NUMBER OF CLOCK CYCLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENTS OF REGISTERS AX, BX, DX, 115</td>
</tr>
<tr>
<td>CONTENTS OF 16 PORTS 0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>CONTENTS OF ALL LOCATIONS 1-SEP LPC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4044 NUMBER OF CLOCK CYCLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENTS OF REGISTERS AX, BX, DX, 116</td>
</tr>
<tr>
<td>CONTENTS OF 16 PORTS 0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>CONTENTS OF ALL LOCATIONS 1-SEP LPC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4072 NUMBER OF CLOCK CYCLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENTS OF REGISTERS AX, BX, DX, 117</td>
</tr>
<tr>
<td>CONTENTS OF 16 PORTS 0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>CONTENTS OF ALL LOCATIONS 1-SEP LPC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4572 NUMBER OF CLOCK CYCLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENTS OF REGISTERS AX, BX, DX, 118</td>
</tr>
<tr>
<td>CONTENTS OF 16 PORTS 0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>CONTENTS OF ALL LOCATIONS 1-SEP LPC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5334 NUMBER OF CLOCK CYCLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENTS OF REGISTERS AX, BX, DX, 119</td>
</tr>
<tr>
<td>CONTENTS OF 16 PORTS 0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>CONTENTS OF ALL LOCATIONS 1-SEP LPC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6272 NUMBER OF CLOCK CYCLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENTS OF REGISTERS AX, BX, DX, 120</td>
</tr>
<tr>
<td>CONTENTS OF 16 PORTS 0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>CONTENTS OF ALL LOCATIONS 1-SEP LPC</td>
</tr>
</tbody>
</table>
APPENDIX B

EXCERPTS FROM THE INTEL 8085A-2 USER GUIDE
4.6 INSTRUCTION SET ENCYCLOPEDIA

In the ensuing dozen pages, the complete 8085A instruction set is described, grouped in order under five different functional headings, as follows:

1. Data Transfer Group — Moves data between registers or between memory locations and registers. Includes moves, loads, stores, and exchanges. (See below.)

2. Arithmetic Group — Adds, subtracts, increments, or decrements data in registers or memory. (See page 4-13.)

3. Logic Group — ANDs, ORs, XORs, compares, rotates, or complements data in registers or between memory and a register. (See page 4-16.)

4. Branch Group — Initiates conditional or unconditional jumps, calls, returns, and restarts. (See page 4-20.)

5. Stack, I/O, and Machine Control Group — Includes instructions for maintaining the stack, reading from input ports, writing to output ports, setting and reading interrupt masks, and setting and clearing flags. (See page 4-22.)

The formats described in the encyclopedia reflect the assembly language processed by Intel-supplied assembler, used with the Intellic development systems.

4.6.1 Data Transfer Group

This group of instructions transfers data to and from registers and memory. Condition flags are not affected by any instruction in this group.

MOV r1, r2 (Move Register)
(r1) — (r2)
The content of register r2 is moved to register r1.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>S</th>
<th>S</th>
<th>S</th>
</tr>
</thead>
</table>

Cycles: 1
States: 4
Addressing: register
Flags: none

MOV r, M (Move from memory)
(r) — ((H) (L))
The content of the memory location, whose address is in registers H and L, is moved to register r.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Cycles: 2
States: 7
Addressing: reg. indirect
Flags: none

MOV M, r (Move to memory)
((H) (L)) — (r)
The content of register r is moved to the memory location whose address is in registers H and L.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Cycles: 2
States: 7
Addressing: reg. indirect
Flags: none

MVI r, data (Move Immediate)
(r) — (byte 2)
The content of byte 2 of the instruction is moved to register r.

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Cycles: 2
States: 7
Addressing: immediate
Flags: none

MVI M, data (Move to memory immediate)
((H) (L)) — (byte 2)
The content of byte 2 of the instruction is moved to the memory location whose address is in registers H and L.

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Cycles: 3
States: 10
Addressing: immed./reg. indirect
Flags: none
## THE INSTRUCTION SET

**LXI rp, data 16** (Load register pair immediate)

- **(rh) — (byte 3),**
- **(rl) — (byte 2)**

Byte 3 of the instruction is moved into the high-order register (rh) of the register pair rp. Byte 2 of the instruction is moved into the low-order register (rl) of the register pair rp.

| Cycles: | 3 |
| States: | 10 |
| Addressing: | immediate |
| Flags: | none |

**LHLD addr** (Load H and L direct)

- **(L) — (byte 3)(byte 2) — (L)**
- **(H) — (byte 3)(byte 2) + 1**

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register L. The content of the memory location at the succeeding address is moved to register H.

| Cycles: | 5 |
| States: | 16 |
| Addressing: | direct |
| Flags: | none |

**SHLD addr** (Store H and L direct)

- **(byte 3)(byte 2) — (L)**
- **(byte 3)(byte 2) + 1**

The content of register L is moved to the memory location whose address is specified in byte 2 and byte 3. The content of register H is moved to the succeeding memory location.

| Cycles: | 5 |
| States: | 16 |
| Addressing: | direct |
| Flags: | none |

**LDA addr** (Load Accumulator direct)

- **(A) — (byte 3)(byte 2)**

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register A.

| Cycles: | 4 |
| States: | 13 |
| Addressing: | direct |
| Flags: | none |

**STA addr** (Store Accumulator direct)

- **((byte 3)(byte 2)) — (A)**

The content of the accumulator is moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.

| Cycles: | 4 |
| States: | 13 |
| Addressing: | direct |
| Flags: | none |

**LDAX rp** (Load accumulator indirect)

- **((rp))**

The content of the memory location, whose address is in the register pair rp, is moved to register A. Note: only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.

| Cycles: | 2 |
| States: | 7 |
| Addressing: | reg. indirect |
| Flags: | none |
THE INSTRUCTION SET

STAX rp (Store accumulator indirect)  
\[(\text{rp}) - (\text{A})\]  
The content of register A is moved to the memory location whose address is in the register pair rp. Note: only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles:</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>States:</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Addressing:</td>
<td>reg. indirect</td>
<td></td>
</tr>
<tr>
<td>Flags:</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

XCHG (Exchange H and L with D and E)  
\[(\text{H}) - (\text{D}) \quad (\text{L}) - (\text{E})\]  
The contents of registers H and L are exchanged with the contents of registers D and E.

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles:</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States:</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressing:</td>
<td>register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags:</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.6.2 Arithmetic Group

This group of instructions performs arithmetic operations on data in registers and memory.

Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Carry, and Auxiliary Carry flags according to the standard rules.

All subtraction operations are performed via two's complement arithmetic and set the carry flag to one to indicate a borrow and clear it to indicate no borrow.

ADD r (Add Register)  
\[(\text{A}) - (\text{A}) + (r)\]  
The content of register r is added to the content of the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles:</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States:</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressing:</td>
<td>register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags:</td>
<td>Z,S,P,CY,AC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADD M (Add memory)  
\[(\text{A}) - (\text{A}) + ((\text{H}) (\text{L}))\]  
The content of the memory location whose address is contained in the H and L registers is added to the content of the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles:</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States:</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressing:</td>
<td>reg. indirect</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags:</td>
<td>Z,S,P,CY,AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADI data (Add immediate)  
\[(\text{A}) - (\text{A}) + (\text{byte 2})\]  
The content of the second byte of the instruction is added to the content of the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles:</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States:</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressing:</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags:</td>
<td>Z,S,P,CY,AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADC r (Add Register with carry)  
\[(\text{A}) - (\text{A}) + (r) + (\text{CY})\]  
The content of register r and the content of the carry bit are added to the content of the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>S</th>
<th>S</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles:</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States:</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressing:</td>
<td>register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags:</td>
<td>Z,S,P,CY,AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
THE INSTRUCTION SET

Ccondition addr (Condition call)

If (CCC),

((SP) - 1) - (PCH)
((SP) - 2) - (PCL)
(SP) - (SP) - 2
(PC) - (byte 3) (byte 2)

If the specified condition is true, the actions specified in the CALL instruction (see above) are performed; otherwise, control continues sequentially.

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>low-order addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high-order addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycles: 2/5
States: 9/18
Addressing: immediate/reg. indirect
Flags: none

Rcondition (Conditional return)

If (CCC),

(PCL) - ((SP))
(PCH) - ((SP) + 1)
(SP) - (SP) + 2

If the specified condition is true, the actions specified in the RET instruction (see above) are performed; otherwise, control continues sequentially.

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

Cycles: 1/3
States: 6/12
Addressing: reg. indirect
Flags: none

RST n (Restart)

((SP) - 1) - (PCH)
((SP) - 2) - (PCL)
(SP) - (SP) - 2
(PC) - 8 * (NNN)

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by two. Control is transferred to the instruction whose address is eight times the content of NNN.

| 1 | 1 | N | N | N | 1 | 1 | 1 |

Cycles: 3
States: 12
Addressing: reg. indirect
Flags: none

Program Counter After Restart

0 0 0 0 0 0 0 0 0 0 0 0 0 0

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
4.6.4 Branch Group

This group of instructions alter normal sequential program flow.

Condition flags are not affected by any instruction in this group.

The two types of branch instructions are unconditional and conditional. Unconditional transfers simply perform the specified operation on register PC (the program counter). Conditional transfers examine the status of one of the four processor flags to determine if the specified branch is to be executed. The conditions that may be specified are as follows:

<table>
<thead>
<tr>
<th>CONDITION</th>
<th>CCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NZ — not zero (Z = 0)</td>
<td>000</td>
</tr>
<tr>
<td>Z — zero (Z = 1)</td>
<td>001</td>
</tr>
<tr>
<td>NC — no carry (CY = 0)</td>
<td>010</td>
</tr>
<tr>
<td>C — carry (CY = 1)</td>
<td>011</td>
</tr>
<tr>
<td>PO — parity odd (P = 0)</td>
<td>100</td>
</tr>
<tr>
<td>PE — parity even (P = 1)</td>
<td>101</td>
</tr>
<tr>
<td>P — plus (S = 0)</td>
<td>110</td>
</tr>
<tr>
<td>M — minus (S = 1)</td>
<td>111</td>
</tr>
</tbody>
</table>

JMP addr (Jump)

(PC) — (byte 3) (byte 2)

Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.

Cycles: 3
States: 10
Addressing: immediate
Flags: none

Jcondition addr (Conditional jump)

If (CCC),

(PC) — (byte 3) (byte 2)

If the specified condition is true, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction; otherwise, control continues sequentially.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>low-order addr</td>
<td>high-order addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycles: 2/3
States: 7/10
Addressing: immediate
Flags: none

CALL addr (Call)

(SP) — 1 — (PCH)

(SP) — 2 — (PCL)

(SP) — (SP) — 2

(PC) — (byte 3) (byte 2)

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2. Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>low-order addr</td>
<td>high-order addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycles: 5
States: 18
Addressing: immediate/ reg. indirect
Flags: none
**THE INSTRUCTION SET**

**RRC (Rotate right)**

\[(A_n) - (A_{n-1}); (A_7) - (A_0)\]

(CY) \[\rightarrow (A_0)\]

The content of the accumulator is rotated right one position. The high order bit and the CY flag are both set to the value shifted out of the low order bit position. Only the CY flag is affected.

![RRC Table](image)

- Cycles: 1
- States: 4
- Flags: CY

**RAL (Rotate left through carry)**

\[(A_{n-1}) - (A_n); (CY) - (A_7)\]

\[(A_0) - (CY)\]

The content of the accumulator is rotated left one position through the CY flag. The low order bit is set equal to the CY flag and the CY flag is set to the value shifted out of the high order bit. Only the CY flag is affected.

![RAL Table](image)

- Cycles: 1
- States: 4
- Flags: CY

**CMA (Complement accumulator)**

\[(A) \rightarrow (\overline{A})\]

The contents of the accumulator are complemented (zero bits become 1, one bits become 0). No flags are affected.

![CMA Table](image)

- Cycles: 1
- States: 4
- Flags: none

**CMC (Complement carry)**

\[(CY) \rightarrow (CY)\]

The CY flag is complemented. No other flags are affected.

![CMC Table](image)

- Cycles: 1
- States: 4
- Flags: CY

**STC (Set carry)**

\[(CY) \rightarrow 1\]

The CY flag is set to 1. No other flags are affected.

![STC Table](image)

- Cycles: 1
- States: 4
- Flags: CY
THE INSTRUCTION SET

**SBB M**  
(Subtract memory with borrow)  
(A) − (A) − ((H) (L)) − (CY)  
The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.

```
1 0 0 1 1 1 1 0
```

Cycles: 2  
States: 7  
Addressing: reg. indirect  
Flags: Z,S,P,CY,AC

**INR M**  
(Increment memory)  
((H) (L)) − ((H) (L)) + 1  
The content of the memory location whose address is contained in the H and L registers is incremented by one. Note: All condition flags except CY are affected.

```
0 0 1 1 0 1 0 0
```

Cycles: 3  
States: 10  
Addressing: reg. indirect  
Flags: Z,S,P,AC

**SBI data**  
(Subtract immediate with borrow)  
(A) − (A) − (byte 2) − (CY)  
The contents of the second byte of the instruction and the contents of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.

```
1 1 0 1 1 1 1 0
```

```
data
```

Cycles: 2  
States: 7  
Addressing: immediate  
Flags: Z,S,P,CY,AC

**DCR r**  
(Decrement Register)  
(r) − (r) − 1  
The content of register r is decremented by one. Note: All condition flags except CY are affected.

```
0 0 D D D 1 0 1
```

Cycles: 1  
States: 4  
Addressing: register  
Flags: Z,S,P,AC

**INR r**  
(Increment Register)  
(r) − (r) + 1  
The content of register r is incremented by one. Note: All condition flags except CY are affected.

```
0 0 D D D 1 0 0
```

Cycles: 1  
States: 4  
Addressing: register  
Flags: Z,S,P,AC

**DCR M**  
(Decrement memory)  
((H) (L)) − ((H) (L)) − 1  
The content of the memory location whose address is contained in the H and L registers is decremented by one. Note: All condition flags except CY are affected.

```
0 0 1 1 0 1 0 1
```

Cycles: 3  
States: 10  
Addressing: reg. indirect  
Flags: Z,S,P,AC
THE INSTRUCTION SET

**ADC M**  
(Add memory with carry)  
\((A) - (A) + ((H) (L)) + (CY)\)  
The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are added to the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>Addressing:</th>
<th>Flags:</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg. indirect</td>
<td>Z,S,P,CY,AC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycles:</th>
<th>States:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**SUB M**  
(Subtract memory)  
\((A) - (A) - ((H) (L))\)  
The content of the memory location whose address is contained in the H and L registers is subtracted from the content of the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>Addressing:</th>
<th>Flags:</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg. indirect</td>
<td>Z,S,P,CY,AC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycles:</th>
<th>States:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**ACI data**  
(Add immediate with carry)  
\((A) - (A) + (\text{byte 2}) + (CY)\)  
The content of the second byte of the instruction and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>Addressing:</th>
<th>Flags:</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate</td>
<td>Z,S,P,CY,AC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycles:</th>
<th>States:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**SUI data**  
(Subtract immediate)  
\((A) - (A) - (\text{byte 2})\)  
The content of the second byte of the instruction is subtracted from the content of the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>Addressing:</th>
<th>Flags:</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate</td>
<td>Z,S,P,CY,AC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycles:</th>
<th>States:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**SUB r**  
(Subtract Register)  
\((A) - (A) - (r)\)  
The content of register r is subtracted from the content of the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>Addressing:</th>
<th>Flags:</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>Z,S,P,CY,AC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycles:</th>
<th>States:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

**SBB r**  
(Subtract Register with borrow)  
\((A) - (A) - (r) - (CY)\)  
The content of register r and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.

<table>
<thead>
<tr>
<th>Addressing:</th>
<th>Flags:</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>Z,S,P,CY,AC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycles:</th>
<th>States:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
APPENDIX C

EXCERPTS FROM THE INTEL 8086 USER GUIDE
INSTRUCTION SET

Mnemonic: MOV
Description: MOV performs a byte or word transfer from the source operand to the destination operand.

Encoding:
Memory or Register Operand to/from Register Operand:

\[
\begin{align*}
\text{Source register} & : \text{reg} \quad \text{Data} \\
\text{Destination register} & : \text{reg} \quad \text{Data}
\end{align*}
\]

Immediate Operand to Memory or Register Operand:

\[
\begin{align*}
\text{Source register} & : \text{REG} \\
\text{Destination register} & : \text{EA}
\end{align*}
\]

Immediate Operand to Register:

\[
\begin{align*}
\text{Source register} & : \text{REG} \\
\text{Data} & : \text{EA}
\end{align*}
\]

Segment Register to Memory or Register Operand:

\[
\begin{align*}
\text{Source register} & : \text{REG} \\
\text{Destination register} & : \text{EA}
\end{align*}
\]

Operation:

(DEST) \iff (SRC)

Flags Affected:

None

Mnemonic: PUSH
Description: PUSH decrements the SP register by two and then transfers a word from the source operand to the stack element currently addressed by SP.

Encoding:
Memory or Register Operand:

\[
\begin{align*}
\text{Source register} & : \text{EA} \\
\text{Register} & : \text{10}
\end{align*}
\]

Immediate Operand to Register:

\[
\begin{align*}
\text{Source register} & : \text{REG} \\
\text{data} & : \text{EA}
\end{align*}
\]

Segment Register to Memory or Register Operand:

\[
\begin{align*}
\text{Source register} & : \text{REG} \\
\text{Destination register} & : \text{EA}
\end{align*}
\]

Operation:

(DEST) \iff (SRC)

Flags Affected:

None
INSTRUCTION SET

Mnemonic: IN and INW
Description: IN or INW transfers a byte (or word) from an input port to the AL register or AX register for INW. The port is specified either with an inline data byte, allowing fixed access to ports 0 through 255, or with a port number in the DX register, allowing variable access to 64K input ports.
Encoding:
Fixed Port:
```
1 1 1 0 0 1 0 w
```
if \( w = 0 \) then SRC = port, DEST = AL
else SRC = port+1, port, DEST = AX
Timing: 10 clocks
Variable Port:
```
1 1 1 0 1 1 1 w
```
if \( w = 0 \) then SRC = (DX), DEST = AL
else SRC = (DX) + 1, DX, DEST = AX
Timing: 8 clocks
Operation:
\( \text{DEST} \leftarrow \text{SRC} \)
Flags Affected:
None

Mnemonic: OUT and OUTW
Description: OUT or OUTW transfers a byte (or word) from the AL register or AX register for OUTW to an output port. The port is specified either with an inline data byte, allowing fixed access to ports 0 through 255, or with a port number in the DX register, allowing variable access to 64K output ports.
Encoding:
Fixed Port:
```
1 1 1 0 0 1 1 w
```
if \( w = 0 \) then SRC = AL, DEST = port
else SRC = AX, DEST = port+1, port
Timing: 10 clocks
Variable Port:
```
1 1 1 0 1 1 1 w
```
if \( w = 0 \) then SRC = AL, DEST = DX
else SRC = AX, DEST = DX + 1, DX
Timing: 8 clocks
Operation:
\( \text{DEST} \leftarrow \text{SRC} \)
Flags Affected:
None

Mnemonic: XLAT
Description: XLAT performs a table lookup byte translation. The AL register is used as an index into a 256-byte table addressed by the BX register. The byte operand so addressed is transferred to AL.
Encoding:
```
1 1 0 0 1 1 1
```
Timing: 11 clocks
Operation:
\( \text{AL} \leftarrow (\text{BX}) \)
Flags Affected:
None

Three Address-Object transfer operations are provided:

- Address Object Transfers
  - LEA Load Effective Address
  - LDS Load Pointer into DS
  - LES Load Pointer into ES

Mnemonic: LEA
Description: LEA (Load Effective Address) transfers the offset address of the source operand to the destination operand. The source operand must be a memory operand and the destination operand can be any 16-bit general, pointer, or index register.
Encoding:
```
1 0 0 0 1 1 0 1 m
```
if \( \text{mod} = 1 \) then undefined operation
Timing: 2 + \( \text{EA} \) clocks
Operation:
\( \text{REG} \leftarrow \text{EA} \)
Flags Affected:
None

Mnemonic: LDS
Description: LDS (Load Pointer into DS) transfers a "pointer-object" i.e., a 32-bit object containing an offset address and a segment address from the source operand (which must be a memory operand) to a pair of destination registers. The segment address is transferred to the DS segment register. The offset address may be transferred to any 16-bit general, pointer, or index register.
Encoding:
```
1 1 0 0 1 1 0 1 m
```
if \( \text{mod} = 1 \) then undefined operation
Timing: 16 + \( \text{EA} \) clocks
Operation:
\( \text{REG} \leftarrow (\text{EA}) \)
\( \text{DS} \leftarrow (\text{EA} + 2) \)
Flags Affected:
None
INSTRUCTION SET

Mnemonic: POPF
Description: POPF (pop flags) transfers specific bits of the stack element addressed by the SP register to the flag registers and then increments SP by two.
Encoding:

```
10011101
```
Timing: 8 clocks

Operation:
Flags <= (SP) + 1: (SP)
(SP) <= (SP) + 2
Flags Affected:
All

4.4.4 Arithmetic

The 8086 provides the four basic mathematical operations in a number of different varieties. Both 8- and 16-bit operations and both signed and unsigned arithmetic are provided. Standard two's complement representation of signed values is used. The addition and subtraction operations serve as both signed and unsigned operations. In these cases, the flag settings allow the distinction between signed and unsigned operations to be made (see Conditional Transfer). Correction operations are provided to allow arithmetic to be performed directly on unpacked decimal digits or on packed decimal representations.

Six flag bits are set or cleared by most arithmetic operations to reflect certain properties of the result of the operation. They generally follow these rules:

- CF is set if the operation resulted in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; otherwise CF is cleared.
- AF is set if the operation resulted in a carry out of (from addition) or a borrow into (from subtraction) the low-order four bits of the result; otherwise AF is cleared.
- ZF is set if the result of the operation is zero; otherwise ZF is cleared.
- SF is set if the high-order bit of the result of the operation is set; otherwise SF is cleared.
- PF is set if the modulo 2 sum of the low-order eight bits of the result of the operation is 0 (even parity); otherwise PF is cleared (odd parity).
- OF is set if the operation resulted in a carry into the high-order bit of the result but not a carry out of the high-order bit, or vice versa; otherwise OF is cleared.

Five addition operations are provided:
ADD Add
ADC Add with Carry
INC Increment
AAA Unpacked BCD ASCII Adjust for Addition
DAA Decimal Adjust for Addition

Mnemonic: ADD
Description: ADD performs an addition of the two source operands and returns the result to one of the operands.
Encoding:

Memory or Register Operand with Register Operand:
```
0000000w | mod reg r/m
```
if \( d = 1 \) then \( LSRC = REG, RSRC = EA, DEST = REG \)
else \( LSRC = EA, RSRC = REG, DEST = EA \)
Timing (clocks): register to register 3
memory to register 9 = EA
register to memory 16 = EA

Immediate Operand to Memory or Register Operand:
```
1000000w | mod 000 r/m data
```
if \( w = 0 \) then \( LSRC = AL, RSRC = data, DEST = AL \)
else \( LSRC = AX, RSRC = data, DEST = AX \)
Timing: 4 clocks

Immediate Operand to Accumulator:
```
00000110 w data data if w=1
```
if \( w = 0 \) then \( LSRC = AL, RSRC = data, DEST = AL \)
else \( LSRC = AX, RSRC = data, DEST = AX \)
Timing: 3 clocks

Operation:
\( (DEST) <= (LSRC) + (RSRC) \)
Flags Affected:
AF, CF, OF, PF, SF, ZF

Mnemonic: ADC
Description: ADC (add with carry) performs an addition of the two source operands, adds one if the CF flag is set, and returns the result to one of the operands.
Encoding:

Memory or Register Operand with Register Operand:
```
0001000w | mod reg r/m
```
if \( d = 1 \) then \( LSRC = REG, RSRC = EA, DEST = REG \)
else \( LSRC = EA, RSRC = REG, DEST = EA \)
Timing (clocks): register to register 3
memory to register 9 = EA
register to memory 16 = EA
INSTRUCTION SET

Mnemonic: SUB
Description: SUB performs a subtraction of the two source operands and returns the result to one of the operands.
Encoding:
Memory or Register Operand and Register Operand:

\[ \text{0010} \text{d.w} \text{mod reg r/m} \]

- if \( d = 1 \) then LSRC = REG, RSRC = EA, DEST = REG
- else LSRC = EA, RSRC = REG, DEST = EA
Timing (clocks): register from register 3
memory from register 9 = EA
register from memory 16 = EA
Immediate Operand from Memory or Register Operand:

\[ \text{100000} \text{s w} \text{ mod 01 r/m} \]

LSRC = EA, RSRC = data, DEST = EA
Timing (clocks): immediate from register 4
immediate from memory 17 = EA
Immediate Operand from Accumulator:

\[ \text{0011} \text{d.w} \text{ mod 01 r/m} \]

- if \( w = 0 \) then LSRC = AL, RSRC = data, DEST = AL
- else LSRC = AX, RSRC = data, DEST = AX
Timing (clocks): immediate from register 4
Operation:

\( (\text{DEST}) <= (\text{LSRC}) - (\text{RSRC}) \)
Flags Affected:
AF, CF, OF, PF, SF, ZF

Mnemonic: DEC
Description: DEC (decrement) performs a subtraction of one from the source operand and returns the result to the operand.
Encoding:
Memory or Register Operand:

\[ \text{0001} \text{s w} \text{ mod 01 r/m} \]

DEST = EA
Timing (clocks): register 2
memory 15 = EA
Register Operand:

\[ \text{0100} \text{ reg} \]
DEST = REG
Timing: 2 clocks
Operation:

\( (\text{DEST}) <= (\text{DEST}) - 1 \)
Flags Affected:
AF, OF, PF, SF, ZF

Mnemonic: SBB
Description: SBB (subtract with borrow) performs a subtraction of the two source operands, subtracts one if the CF flag is set, and returns the result to one of the operands.
Encoding:
Memory or Register Operand and Register Operand:

\[ \text{0011} \text{d.w} \text{ mod reg r/m} \]

- if \( d = 1 \) then LSRC = REG, RSRC = EA, DEST = REG
- else LSRC = EA, RSRC = REG, DEST = EA
Timing (clocks): register from register 3
memory from register 9 = EA
register from memory 16 = EA
Immediate Operand from Memory or Register Operand:

\[ \text{100000} \text{s w} \text{ mod 01 r/m} \]

LSRC = EA, RSRC = data, DEST = EA
Timing (clocks): immediate from register 4
immediate from memory 17 = EA
Immediate Operand from Accumulator:

\[ \text{0001} \text{d.w} \]
data 4

- if \( w = 0 \) then LSRC = AL, RSRC = data, DEST = AL
- else LSRC = AX, RSRC = data, DEST = AX
Timing (clocks): immediate from register 4
Operation:

\( (\text{DEST}) <= (\text{LSRC}) - (\text{RSRC}) \)
Flags Affected:
AF, CF, OF, PF, SF, ZF
INSTRUCTION SET

Mnemonic: **MUL**
Description: MUL (multiply) performs an unsigned multiplication of the accumulator (AL or AX) and the source operand, returning a double-length result to the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation). CF and OF are set if the top half of the result is nonzero.

Encoding:

```
 1 1 1 1 1 0 1 1 w | mod 1 0 0 r / m
```

if \( w = 0 \)
then \( \text{LSRC} = \text{AL}, \text{RSRC} = \text{EA}, \text{DEST} = \text{AX}, \text{EXT} = \text{AH} \)
else \( \text{LSRC} = \text{AX}, \text{RSRC} = \text{EA}, \text{DEST} = \text{DX}:\text{AX}, \text{EXT} = \text{DX} \)
Timing (clocks): 8-bit \( 71=\text{EA} \)
16-bit \( 124=\text{EA} \)

Operation:
\( \text{DEST} <= \text{LSRC} \times \text{RSRC} \), where \( \times \) is unsigned multiply
if \( \text{EXT} = 0 \)
then \( \text{CF} <= 0 \)
else \( \text{CF} <= 1 \)
\( \text{OF} <= \text{CF} \)

Flags Affected:
CF, OF
AF, PF, SF, ZF undefined

Mnemonic: **IMUL**
Description: IMUL (integer multiply) performs a signed multiplication of the accumulator (AL or AX) and the source operand, returning a double-length result to the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation). CF and OF are set if the top half of the result is not the sign-extension of the low half of the result.

Encoding:

```
 1 1 1 1 0 1 1 w | mod 1 0 0 r / m
```

if \( w = 0 \)
then \( \text{LSRC} = \text{AL}, \text{RSRC} = \text{EA}, \text{DEST} = \text{AX}, \text{EXT} = \text{AH}, \text{LOW} = \text{AL} \)
else \( \text{LSRC} = \text{AX}, \text{RSRC} = \text{EA}, \text{DEST} = \text{DX}:\text{AX}, \text{EXT} = \text{DX}, \text{LOW} = \text{AX} \)
Timing (clocks): 8-bit \( 90=\text{EA} \)
16-bit \( 144=\text{EA} \)

Operation:
\( \text{DEST} <= \text{LSRC} \times \text{RSRC} \), where \( \times \) is signed multiply
if \( \text{EXT} = \text{sign-extension of LOW} \) then \( \text{CF} <= 0 \)
else \( \text{CF} <= 1 \)
\( \text{OF} <= \text{CF} \)

Flags Affected:
CF, OF
AF, PF, SF, ZF undefined

Mnemonic: **AAM**
Description: AAM (Unpacked BCD ASCII adjust for multiply) performs a correction of the result in AX of multiplying two unpacked decimal operands, yielding an unpacked decimal product.

Encoding:

```
 1 1 0 1 0 1 0 0 | 0 0 0 0 0 1 0 1 0
```

Timing: 83 clocks

Operation:
\( \text{AH} <= (\text{AL} \div \text{OA}) \)
\( \text{AL} <= (\text{AL} \div \text{OA}) \)

Flags Affected:
PF, SF, ZF
AF, CF, OF undefined

Three division operations are provided, as well as two sign-extension operations which support signed division:
DIV Divide
IDIV Integer Divide
AAD Unpacked BCD ASCII Adjust for Division
CBW Convert Byte to Word
CWD Convert Word to Double Word
4.4.7 Control Transfer

Four classes of control transfer operations may be distinguished: calls, jumps, and returns; conditional transfers; iteration control; and interrupts.

All control transfer operations cause, perhaps upon a certain condition, the program execution to continue at some new location in memory, possibly in a new code segment.

**NOTE:** Queue reinitialization is not included in the timing information for transfer operations. To account for queue loading, add 4 clocks to timing numbers.

Calls, Jumps, and Returns. Two basic varieties of calls, jumps, and returns are provided — those which transfer control within the current code segment, and those which transfer control to an arbitrary code segment, which then becomes the current code segment. Both direct and indirect transfers are supported; indirect transfers make use of the standard addressing modes described in Section 4.4. Intra-segment direct calls and jumps specify a self-relative direct displacement, thus allowing position independent code. A shortened jump instruction is available for transfers within ±128 bytes from the instruction using less code.

The three transfer operations are:

- CALL Call
- JMP Jump
- RET Return

**Mnemonic: CALL**

Description: CALL pushes the offset address of the next instruction onto the stack in the case of an inter-segment transfer the CS segment register is pushed first; and then transfers control to the target operand.

Encoding:
- Intra-segment Direct:
  - 11110000 disp-low disp-high
  - DEST = [EA]
  - Timing: 13+EA clocks
- Intra-Segment Indirect:
  - 11111111 mod 010 r/m
  - DEST = [IP] disp
  - Timing: 11 clocks
- Inter-Segment Direct:
  - 10010101 offset-low offset-high
  - SEG-low SEG-high
  - DEST = offset, SEG = seg
  - Timing: 20 clocks

Inter-Segment Indirect:

```
1 1 1 1 1 1 1 1 mod 0 1 1 r/m
```

DEST = [EA], SEG = [EA + 2]

Timing: 29+EA clocks

**Operation:**

if Inter-Segment then

- (SP) <= (SP) - 2
- (IP) <= (CS)
- (CS) <= SEG
- (SP) <= (IP) - 2
- (IP) <= DEST

Flags Affected:
None

**Mnemonic: JMP**

Description: JMP transfers control to the target operand.

Encoding:
- Intra-Segment Direct:
  - 1110001 disp-low disp-high
  - DEST = [IP] disp
  - Timing: 7 clocks
- Intra-Segment Direct Short:
  - 1110000
  - DEST = [IP] disp
  - Timing: 2 clocks

Intra-Segment Indirect:

```
1 1 1 1 1 1 1 1 mod 1 0 0 r/m
```

DEST = [EA]

Timing: 7+EA clocks

Inter-Segment Direct:

```
1 1 1 0 1 0 1 0 offset-low offset-high
```

DEST = offset, SEG = seg

Timing: 7 clocks
REFERENCES


