MACHINE LANGUAGE PROGRAMMING

FOR THE '8 0 0 8'

(AND SIMILAR MICROCOMPUTERS)

AUTHOR: NAT WADSWORTH

SCELBI COMPUTER CONSULTING, INC.
1322 REAR - BOSTON POST ROAD
MILFORD, CT. 06460

MACHINE LANGUAGE PROGRAMMING

FOR THE '8 0 0 8'

(AND SIMILAR MICROCOMPUTERS)

****

TABLE OF CONTENTS

****

INTRODUCTION

THIS MANUAL IS ON MACHINE LANGUAGE PROGRAMMING METHODS AND TECHNIQUES FOR 8008 BASED COMPUTERS. WHILE MACHINE LANGUAGE PROGRAMMING IS THE MOST FUNDAMENTAL TYPE OF COMPUTER PROGRAMMING POSSIBLE. IT IS BY FAR THE MOST EFFICIENT METHOD, IN TERMS OF UTILIZATION OF THE MACHINES'S CAPABILITIES, WITH WHICH TO PROGRAM OR SET UP A 8008 SYSTEM TO PERFORM A JOB. MACHINE LANGUAGE PROGRAMMING IS, ON THE OTHER HAND, THE MOST DEMANDING METHOD OF COMPUTER PROGRAMMING IN TERMS OF HUMAN ENDEAVOR AND SKILL. HOWEVER, THE FUNDAMENTAL SKILLS AND TECHNIQUES NECESSARY FOR MACHINE LANGUAGE PROGRAMMING CAN BE APPLIED TO VIRTUALLY ANY LEVEL OF COMPUTER PROGRAMMING. A CLEAR UNDERSTANDING OF MACHINE LANGUAGE PROGRAMMING WILL GIVE ONE GREAT INSIGHT INTO ANY HIGHER LEVEL LANGUAGE PROGRAMMING.

MACHINE LANGUAGE PROGRAMMING IS THE ACTUAL STEP-BY-STEP PROGRAMMING OF THE COMPUTER USING THE MACHINE CODES AND MEMORY ADDRESSES THAT WILL BE USED BY THE COMPUTER DIRECTLY. IT IS CONSIDERABLY MORE DETAILED THAN PROGRAMMING IN A HIGH LEVEL LANGUAGE SUCH AS FORTRAN (RTM) OR BASIC (RTM) - IT IS IN FACT, THE LEVEL OF PROGRAMMING FROM WHICH THOSE HIGH LEVEL LANGUAGES ARE DEVELOPED. IN FACT, IF ONE KNOWS HOW TO DEVELOP PROGRAMS IN MACHINE LANGUAGE, ONE WILL HAVE THE BASIC SKILLS NECESSARY FOR DEVELOPING A HIGHER LEVEL LANGUAGE. (THAT IS A TREMENDOUS ASSETT OVER ONE WHO ONLY KNOWS HOW TO PROGRAM IN HIGHER LEVEL LANGUAGES.)

THE PRIMARY REASON FOR HAVING A MANUAL DEVOTED TO MACHINE LANGUAGE PROGRAMMING FOR THE 8008 IS BECAUSE THIS METHOD IS BY FAR THE MOST EFFICIENT METHOD FOR PACKING A PROGRAM INTO A SMALL AMOUNT OF MEMORY. AS USER'S KNOW, MEMORY ELEMENTS COST A GOOD AMOUNT OF MONEY, AND THE MORE ONE CAN PROGRAM INTO A GIVEN AMOUNT OF MEMORY, THE LESS MEMORY REQUIRED FOR A GIVEN TASK - AND THE MORE ONE CAN DO WITH A LOW COST MACHINE. HIGH LEVEL LANGUAGES REQUIRE MUCH MORE MEMORY BECAUSE OF TWO MAJOR REASONS. FIRST, A LARGE AMOUNT OF MEMORY MUST BE USED BY THE HIGH LEVEL LANGUAGE ITSELF. SECOND, HIGHER LEVEL LANGUAGES MUST CONVERT USER STATEMENTS OR COMMANDS TO MACHINE LANGUAGE CODES AND THEY GENERALLY CANNOT DO THIS ANY WHERE NEAR AS EFFICIENTLY (MEMORY USAGE - WISE) AS A TRAINED HUMAN PROGRAMMER!

ANOTHER REASON FOR DISCUSSING MACHINE LANGUAGE PROGRAMMING AT LENGTH IS BECAUSE IT IS THE ONLY METHOD WHERE-BY MANY CAPABILITIES OF THE MACHINE CAN BE TAKEN ADVANTAGE OF - THIS IS PARTICULARLY TRUE FOR "REAL-TIME" AND I/O OPERATIONS. MANY USERS WILL WANT TO UTILIZE THEIR 8008 MINICOMPUTERS FOR UNIQUE APPLICATIONS AND THE CONTENTS OF THIS MANUAL WILL PRESENT MANY IDEAS AND CONCEPTS FOR THESE PEOPLE TO APPLY TO THEIR INDIVIDUAL APPLICATIONS.

MACHINE LANGUAGE PROGRAMMING IN GENERAL IS NOWHERE AS DIFFICULT TO LEARN AS MANY PEOPLE MIGHT TEND TO THINK WHEN FIRST INTRODUCED TO THE SUBJECT. THIS IS ESPECIALLY TRUE FOR THE 8008 TYPE MINICOMPUTER. THERE ARE MANY FUNDAMENTAL CONCEPTS THAT CAN BE READILY LEARNED AND ONCE THIS HAS BEEN ACCOMPLISHED THE NOVICE IS ON THE WAY TO DEVELOPING ORIGINAL SOLUTIONS TO PROGRAMMING PROBLEMS THAT MAY BE OF UNIQUE INTEREST TO THE INDIVIDUAL.

COMPUTER PROGRAMMING, AND MACHINE LANGUAGE PROGRAMMING IN PARTICULAR, IS IN MANY RESPECTS AN ART, AND IN OTHER RESPECTS A VERY RIGID SCIENCE. THE FUN PART, AND WHAT CAN BE CONSIDERED ARTISTIC, IS THAT INDIVIDUALS CAN TAILOR OR FASHION SERIES OF INSTRUCTIONS TO ACCOMPLISH A PARTICULAR TASK IN A VARIETY OF WAYS. THE SCIENTIFIC PART OF PROGRAM-

MING INVOLVES ACQUIRING SOME BASIC SKILLS AND KNOWLEDGE ABOUT WHAT CAN AND CANNOT BE DONE, AND AT A HIGHER LEVEL, PERHAPS, AN UNDERSTANDING OF BASIC MATHEMATIC ALGORITHMS AND PROCEDURES THAT CAN BE READILY APPLIED USING COMPUTER TECHNIQUES. SOME OF THE BASIC SKILLS INCLUDE KNOWING JUST WHAT THE AVAILABLE MACHINE INSTRUCTIONS ARE, AND SOME OF THE MOST FREQUENTLY USED COMBINATIONS OF INSTRUCTIONS THAT WILL PERFORM FREQUENTLY REQUIRED TASKS - THESE SKILLS ARE AS FUNDAMENTAL AS A PAINTER KNOWING THE PRIMARY COLORS AND HOW TO COMBINE THEM TO CREATE THE COMMONLY USED SECONDARY COLORS. HOWEVER, LIKE THE PAINTER WHO COMBINES THE BASIC PIGMENTS, BEYOND A CERTAIN POINT THE TASK OF COMPUTER PROGRAMMING BECOMES A HIGHLY CREATIVE INDIVIDUALISTIC ART. AND, IT IS AN ART IN WHICH ONE CAN CONSTANTLY GAIN NEW SKILLS AND ABILITY. A HIGH SCHOOL STUDENT OR A COLLEGE PROFESSOR CAN BOTH FIND EQUALLY REWARDING CHALLENGES IN COMPUTER PROGRAMMING. THERE ARE OFTEN MANY DIFFERENT WAYS TO PROGRAM A COMPUTER TO PERFORM A GIVEN TASK AND MANY "TRADE - OFFS" TO CONSIDER WHEN DEVELOPING A PROGRAM (SUCH AS HOW MUCH MEMORY TO USE, WHAT FUNCTIONS HAVE PRIORITY, HOW MUCH BURDEN TO PLACE ON THE HUMAN OPERATOR WHEN THE PROGRAM IS OPERATING). EACH INDIVIDUAL SOON LEARNS TO CAPITALIZE ON THE ASPECTS CONSIDERED MOST IMPORTANT FOR THE SPECIFIC APPLICATION AT HAND AND WILL DEVELOP THEIR OWN PERSONAL METHODS FOR HANDLING VARIOUS TYPES OF PROGRAMMING TASKS.

REMEMBER AS YOU READ THIS MANUAL THAT THERE ARE MANY OTHER WAYS OF PROGRAMMING A COMPUTER TO PERFORM MANY OF THE EXAMPLE PROGRAMS ILLUSTRATED. DON'T BE AFRAID TO DEVELOP YOUR OWN SOLUTIONS FOR PRACTICE AS YOU GO THROUGH THE MATERIAL. TRY OUT YOUR SOLUTIONS - SEE IF THEY WORK AS PLANNED - PRACTICE BEING A "CREATIVE PROGRAMMER!" BY THE TIME YOU HAVE COMPLETED ABSORBING AND UNDERSTANDING THE CONTENTS OF THIS PUBLICATION YOU SHOULD BE WELL EQUIPPED TO DEVELOP PROGRAMS OF YOUR OWN AND THUS BE IN A POSITION TO REAP EVEN GREATER BENEFITS FROM YOUR 8008 BASED MICROCOMPUTER THAN JUST BEING ABLE TO OPERATE PROGRAMS THAT OTHER PEOPLE HAVE PREPARED.

THE FIRST CHAPTER OF THIS MANUAL CONTAINS A DETAILED PRESENTATION OF THE INSTRUCTION SET THAT THE 8008 CPU IS CAPABLE OF PERFORMING. IT GOES ALMOST WITHOUT SAYING, THAT THE FIRST STEP TOWARDS BECOMING A PROFICIENT MACHINE LANGUAGE PROGRAMMER IS TO BECOME THOROUGHLY FAMILIAR WITH ALL THE TYPES OF INSTRUCTIONS THAT THE MACHINE CAN EXECUTE AND ESPECIALLY TO LEARN ABOUT ANY SPECIAL CONDITIONS THAT APPLY TO THE EXECUTION OF SPECIFIC TYPES OF COMMANDS. THE LEAD-OFF CHAPTER PRESENTS A COMPREHENSIVE EXPLANATION OF ALL THE INSTRUCTIONS IN THE 8008 REPERTOIRE ALONG WITH THE MNEMONICS AND MACHINE CODES. THE READER SHOULD BECOME QUITE FAMILIAR WITH THE INFORMATION PRESENTED THERE BEFORE GOING FURTHER IN THIS MANUAL. (AT LEAST TO THE POINT WHERE ONE CAN RAPIDLY LOCATE ANY CLASS OF INSTRUCTIONS IN THE CHAPTER IN ORDER TO REFRESH ONE'S MEMORY ON JUST HOW AN INSTRUCTION OPERATES AND TO BE ABLE TO RAPIDLY LOCATE THE "MACHINE CODES" WHEN ONE IS PREPARING A PROGRAM)!

THE '8008' CPU INSTRUCTION SET

THIS MINI-COMPUTER HAS QUITE A COMPREHENSIVE INSTRUCTION SET THAT CONSISTS OF 48 BASIC INSTRUCTIONS, WHICH, WHEN THE POSSIBLE PERMUTATIONS ARE CONSIDERED, RESULT IN A TOTAL SET OF ABOUT 170 INSTRUCTIONS.

THE INSTRUCTION SET ALLOWS THE USER TO DIRECT THE COMPUTER TO PERFORM OPERATIONS WITH MEMORY, WITH THE 7 BASIC REGISTERS IN THE CPU, AND WITH INPUT AND OUTPUT PORTS.

IT SHOULD BE POINTED OUT THAT THE 7 BASIC REGISTERS IN THE CPU CONSIST OF ONE ACCUMULATOR - THAT IS A REGISTER THAT CAN PERFORM MATHEMATICAL AND LOGIC OPERATIONS, AND AN ADDITIONAL 6 REGISTERS WHICH WHILE NOT HAVING THE FULL CAPABILITY OF THE ACCUMULATOR, CAN PERFORM CERTAIN OPERATIONS (INCREMENT AND DECREMENT), CAN STORE DATA, AND CAN OPERATE WITH THE ACCUMULATOR. TWO OF THE SIX REGISTERS HAVE SPECIAL SIGNIFICANCE BECAUSE THEY MAY BE USED TO "POINT" TO AN ADDRESS IN MEMORY.

THE SEVEN CPU REGISTERS HAVE ARBITRARILY BEEN GIVEN SYMBOLS SO THAT WE MAY REFER TO THEM IN A COMMON LANGUAGE. THE FIRST REGISTER IS DESIGNATED BY THE SYMBOL "A" IN THE FOLLOWING DISCUSSION AND WILL BE CONSIDERED THE ACCUMULATOR REGISTER. THE NEXT FOUR REGISTERS WILL BE REFERRED TO AS THE "B," "C," "D," AND "E," REGISTERS, AND THE REMAINING TWO SPECIAL MEMORY POINTING REGISTERS SHALL BE DESIGNATED THE "H" (FOR THE HIGH PORTION OF A MEMORY ADDRESS) AND THE "L" (FOR THE LOW PORTION OF A MEMORY ADDRESS) REGISTERS.

THE CPU ALSO HAS SEVERAL FLIP-FLOPS WHICH SHALL BE REFERRED TO AS "FLAGS." THESE FLIP-FLOPS ARE SET AS THE RESULT OF CERTAIN OPERATIONS AND ARE IMPORTANT BECAUSE THEY CAN BE "TESTED" BY MANY OF THE INSTRUCTIONS AND THE INSTRUCTION'S MEANING CHANGED AS A CONSEQUENCE OF THE FLAGS PARTICULAR STATUS AT THE TIME IT IS TESTED. THERE ARE FOUR BASIC FLAGS WHICH WILL BE REFERRED TO IN THIS MANUAL DESIGNATED AS FOLLOWS:

> THE "C" FLAG REFERS TO THE CARRY BIT STATUS. THE CARRY BIT IS A 1 UNIT REGISTER WHICH CHANGES STATE WHEN THE ACCUMULATOR OVER-FLOWS OR UNDER-FLOWS. THIS BIT CAN ALSO BE SET TO A KNOWN CONDITION BY CERTAIN TYPES OF INSTRUCTIONS. THIS IS IMPORTANT TO REMEMBER WHEN DEVELOPING A PROGRAM BECAUSE QUITE OFTEN A PROGRAM WILL HAVE A LONG STRING OF INSTRUCTIONS WHICH DO NOT UTILIZE THE CARRY BIT OR CARE ABOUT ITS STATUS, BUT WHICH WILL BE CAUSING THE CARRY BIT TO CHANGE ITS STATUS FROM TIME TO TIME. THUS, WHEN ONE PREPARES TO DO A SERIES OF OPERATIONS THAT WILL RELY ON THE CARRY BIT, ONE OFTEN DESIRES TO SET THE CARRY BIT TO A KNOWN STATE.
>
> THE "Z" FOR ZERO FLAG REFERS TO A 1 UNIT REGISTER THAT WHEN DESIRED WILL INDICATE WHETHER THE VALUE OF THE ACCUMULATOR IS EXACTLY EQUAL TO ZERO. IN ADDITION, IMMEDIATELY AFTER AN INCREMENT OR DECREMENT OF THE B, C, D, E, H OR L REGISTERS, THIS FLAG WILL ALSO INDICATE WHETHER THE INCREMENT OR DECREMENT CAUSED THAT PARTICULAR REGISTER TO GO TO ZERO.
>
> THE "S" FOR SIGN FLAG REFERS TO A 1 UNIT REGISTER THAT INDICATES WHETHER THE VALUE IN THE ACCUMULATOR IS A POSITIVE OR NEGATIVE VALUE (BASED ON TWO'S COMPLEMENT NOMENCLATURE). ESSENTIALLY, THIS FLAG MONITORS THE MOST SIGNIFICANT BIT IN THE ACCUMULATOR AND IS "SET" WHEN IT IS A ONE.

THE "P" FLAG REFERS TO THE LAST FLAG IN THE GROUP WHICH IS FOR INDICATING WHEN THE ACCUMULATOR CONTAINS A VALUE WHICH HAS EVEN PARITY. PARITY IS USEFUL FOR A NUMBER OF REASONS AND IS USUALLY USED IN CONJUNCTION WITH TESTING FOR ERROR CONDITIONS ON WORDS OF DATA PARTICULARLY WHEN INPUTTING DATA FROM EXTERNAL SOURCES. EVEN PARITY OCCURS WHEN THE NUMBER OF BITS THAT ARE A "1" IN THE ACCUMULATOR (OUT OF THE EIGHT POSSIBLE) IS AN EVEN VALUE, I.E., 2, 4, 6, OR 8; REGARDLESS OF WHAT ORDER THEY MAY BE IN THE ACCUMULATOR REGISTER.

IT IS IMPORTANT TO NOTE THAT THE "Z," "S," AND "P" FLAGS (AS WELL AS THE PREVIOUSLY MENTIONED "C" FLAG) CAN ALL BE SET TO KNOWN STATES BY CERTAIN INSTRUCTIONS. IT IS ALSO IMPORTANT TO NOTE THAT SOME INSTRUCTIONS DO NOT RESULT IN THE FLAGS BEING SET SO THAT IF THE PROGRAMMER DESIRES TO HAVE THE PROGRAM MAKE "DECISIONS" BASED ON THE STATUS OF FLAGS, THE PROGRAMMER SHOULD ENSURE THAT THE PROPER INSTRUCTION, OR SEQUENCE OF INSTRUCTIONS IS UTILIZED. IT IS PARTICULARLY IMPORTANT TO NOTE THAT "LOAD REGISTER" INSTRUCTIONS DO NOT BY THEMSELVES SET THE FLAGS. SINCE IT IS OFTEN DESIRABLE TO OBTAIN A DATA WORD (I.E. LOAD IT INTO THE ACCUMULATOR) AND TEST ITS STATUS FOR SUCH PARAMETERS AS WHETHER OR NOT THE VALUE IS ZERO, OR A NEGATIVE NUMBER ETC., THE PROGRAMMER MUST REMEMBER TO FOLLOW A LOAD INSTRUCTION BY A LOGICAL INSTRUCTION (SUCH AS THE NDA - "AND THE ACCUMULATOR") IN ORDER TO SET THE FLAGS BEFORE USING AN INSTRUCTION THAT IS CONDITIONAL IN REGARDS TO THE FLAG STATUS.

THE DESCRIPTION OF THE VARIOUS TYPES OF INSTRUCTIONS AVAILABLE WITH AN 8008 CPU UNIT WHICH FOLLOWS WILL PROVIDE BOTH THE MACHINE LANGUAGE CODE FOR THE INSTRUCTION GIVEN AS 3 OCTAL DIGITS, AND ALSO A MNEMONIC NAME SUITABLE FOR WRITING PROGRAMS IN SYMBOLIC TYPE LANGUAGE WHICH IS USUALLY EASIER THAN TRYING TO REMEMBER OCTAL CODES! IT MAY BE NOTED THAT THE SYMBOLIC LANGUAGE USED IS THE SAME AS THAT SUGGESTED BY INTEL CORPORATION WHICH ORIGINALLY DEVELOPED THE 8008 "CPU-ON-A-CHIP" WHICH IS AT THE HEART OF 8008 SYSTEMS, AND HENCE USERS WHO MAY ALREADY BE FAMILIAR WITH THE SUGGESTED MNEMONICS WILL NOT HAVE ANY "RELEARNING" PROBLEMS AND THOSE LEARNING THE MNEMONICS FOR THE FIRST TIME WILL HAVE PLENTY OF "GOOD COMPANY." IF THE PROGRAMMER IS NOT ALREADY AWARE OF IT, THE USE OF MNEMONICS FACILITATES WORKING WITH AN "ASSEMBLER" PROGRAM WHEN IT IS DESIRED TO DEVELOP RELATIVELY LARGE AND COMPLEX PROGRAMS. THUS THE PROGRAMMER IS URGED TO CONCENTRATE ON LEARNING THE MNEMONICS FOR THE INSTRUCTIONS AND NOT WASTE TIME MEMORIZING THE OCTAL CODES. AFTER A PROGRAM HAS BEEN WRITTEN USING THE MNEMONIC CODES, THE PROGRAMMER CAN ALWAYS USE A LOOKUP TABLE TO CONVERT TO THE MACHINE CODE IF AN ASSEMBLER PROGRAM IS NOT AVAILABLE. ITS A LOT EASIER TECHNIQUE (AND LESS SUBJECT TO ERROR) THAN TRYING TO MEMORIZE THE 170 OR SO 3 DIGIT COMBINATIONS WHICH MAKE UP THE MACHINE INSTRUCTION CODE SET!

THE PROGRAMMER MUST ALSO BE AWARE, THAT IN THIS MACHINE, SOME INSTRUCTIONS REQUIRE MORE THAN ONE "WORD" IN MEMORY. "IMMEDIATE" TYPE COMMANDS REQUIRE TWO CONSECUTIVE WORDS AND JUMP AND CALL COMMANDS REQUIRE THREE CONSECUTIVE WORDS. THE REMAINING TYPES OF INSTRUCTIONS ONLY REQUIRE ONE WORD. THIS WILL BE PRESENTED IN DETAIL IN THE DESCRIPTION FOR EACH TYPE OF INSTRUCTION.

THE FIRST GROUP OF INSTRUCTIONS TO BE PRESENTED ARE THOSE THAT ARE USED TO "LOAD" DATA FROM ONE CPU REGISTER TO ANOTHER, OR FROM A CPU REGISTER TO A WORD IN MEMORY, OR VICE-VERSA. THIS GROUP OF INSTRUCTIONS REQUIRES JUST ONE WORD OF MEMORY. IT IS IMPORTANT TO NOTE THAT NONE OF THE INSTRUCTIONS IN THIS GROUP AFFECT THE "FLAGS."

LOAD DATA FROM ONE CPU REGISTER TO ANOTHER CPU REGISTER

MNEMONIC	MACHINE CODE
LAA	3 0 0
LBA	3 1 0
.	.
.	.
LAB	3 0 1

THE LOAD REGISTER GROUP OF INSTRUCTIONS ALLOWS THE PROGRAMMER TO MOVE THE CONTENTS OF ONE CPU REGISTER INTO ANOTHER CPU REGISTER. THE CONTENTS OF THE ORIGINATING (FROM) REGISTER IS NOT CHANGED. THE CONTENTS OF THE DESTINATION (TO) REGISTER BECOMES THE SAME AS THE ORIGINATING REGISTER. ANY CPU REGISTER CAN BE LOADED INTO ANY CPU REGISTER. NOTE THAT FOR INSTANCE LOADING REGISTER "A" INTO REGISTER "A" IS ESSENTIALLY A "NOP" (NO OPERATION) COMMAND. WHEN USING MNEMONICS THE LOAD SYMBOL IS THE LETTER "L" FOLLOWED BY THE "TO" REGISTER AND THEN THE "FROM" REGISTER. THE MNEMONIC "LBA" MEANS THE THE CONTENTS OF REGISTER "A" (THE ACCUMULATOR) IS TO BE LOADED INTO REGISTER "B." THE MNEMONIC "LAB" STATES THAT REGISTER "B" IS TO HAVE ITS CONTENTS LOADED INTO REGISTER "A." IT CAN BE SEEN THAT THIS BASIC INSTRUCTION HAS MANY VARIATIONS. THE MACHINE LANGUAGE CODING FOR THIS INSTRUCTION IS IN THE SAME FORMAT AS THE MNEMONIC CODE EXCEPT THAT THE LETTERS USED TO REPRESENT THE REGISTERS ARE REPLACED BY NUMBERS THAT THE MACHINE CAN USE. USING OCTAL CODE, THE 7 CPU REGISTERS ARE CODED AS FOLLOWS:

REG "A" = 0
REG "B" = 1
REG "C" = 2
REG "D" = 3
REG "E" = 4
REG "H" = 5
REG "L" = 6

ALSO SINCE THE MACHINE CAN ONLY UTILIZE NUMBERS, THE OCTAL NUMBER 3 IN THE MOST SIGNIFICANT LOCATION OF A WORD SIGNIFIES THAT THE COMPUTER IS TO PERFORM A "LOAD" OPERATION. THUS, IN MACHINE CODING, THE INSTRUCTION FOR LOADING REGISTER "B" WITH THE CONTENTS OF REGISTER "A" BECOMES: 3 1 0 (IN OCTAL FORM) OR, IF ONE WANTED TO GET VERY DETAILED, THE ACTUAL BINARY CODING FOR THE 8 BITS OF INFORMATION IN THE INSTRUCTION WORD WOULD BE: 1 1 0 0 1 0 0 0. IT IS IMPORTANT TO NOTE THAT THE LOAD INSTRUCTIONS DO NOT AFFECT ANY OF THE "FLAGS."

LOAD DATA FROM ANY CPU REGISTER TO A LOCATION IN MEMORY

LMA	3 7 0
LMB	3 7 1
LMC	3 7 2
LMD	3 7 3
LME	3 7 4
LMH	3 7 5
LML	3 7 6

THIS INSTRUCTION IS VERY SIMILAR TO THE PREVIOUS GROUP OF INSTRUCTIONS EXCEPT THAT NOW THE CONTENTS OF A CPU REGISTER WILL BE LOADED INTO A SPECIFIED MEMORY LOCATION. THE MEMORY LOCATION THAT WILL RECEIVE THE CONTENTS OF THE PARTICULAR CPU REGISTER IS THAT WHOSE ADDRESS IS SPECIFIED BY THE CONTENTS OF THE CPU "H" AND "L" REGISTERS AT THE TIME THE INSTRUCTION IS EXECUTED. THE "H" CPU REGISTER SPECIFIES THE "HIGH" PORTION OF THE ADDRESS DESIRED, AND THE "L" CPU REGISTER SPECIFIES THE "LOW" PORTION OF THE ADDRESS

INTO WHICH DATA FROM THE SELECTED CPU REGISTER IS TO BE LOADED. NOTE THAT THERE ARE 7 DIFFERENT INSTRUCTIONS IN THIS GROUP AS ANY CPU REGISTER CAN HAVE ITS CONTENTS LOADED INTO ANY LOCATION IN MEMORY. THIS GROUP OF INSTRUCTIONS DOES NOT AFFECT ANY OF THE "FLAGS."

LOAD DATA FROM A MEMORY LOCATION TO ANY CPU REGISTER

LAM	3 0 7
LBM	3 1 7
LCM	3 2 7
LDM	3 3 7
LEM	3 4 7
LHM	3 5 7
LLM	3 6 7

THIS GROUP OF INSTRUCTIONS CAN BE CONSIDERED THE OPPOSITE OF THE PREVIOUS GROUP. NOW, THE CONTENTS OF THE WORD IN MEMORY WHOSE ADDRESS IS SPECIFIED BY THE "H" (FOR THE HIGH PORTION OF THE ADDRESS) AND "L" (LOW PORTION OF THE ADDRESS) REGISTERS WILL BE LOADED INTO THE CPU REGISTER SPECIFIED BY THE INSTRUCTION. ONCE AGAIN, THIS GROUP OF INSTRUCTIONS HAS NO AFFECT ON THE STATUS OF THE "FLAGS."

LOAD "IMMEDIATE" DATA INTO A CPU REGISTER

LAI	0 0 6
LBI	0 1 6
LCI	0 2 6
LDI	0 3 6
LEI	0 4 6
LHI	0 5 6
LLI	0 6 6

AN "IMMEDIATE" TYPE OF INSTRUCTION REQUIRES TWO WORDS IN ORDER TO BE COMPLETELY SPECIFIED. THE FIRST WORD IS THE INSTRUCTION ITSELF, THE SECOND WORD, OR "IMMEDIATELY FOLLOWING" WORD, MUST CONTAIN THE DATA UPON WHICH IMMEDIATE ACTION IS TAKEN. THUS, A LOAD "IMMEDIATE" INSTRUCTION IN THIS GROUP MEANS THAT THE CONTENTS OF THE WORD IMMEDIATELY FOLLOWING THE INSTRUCTION WORD IS TO BE LOADED INTO THE SPECIFIED REGISTER. FOR EXAMPLE, A TYPICAL LOAD IMMEDIATE INSTRUCTION WOULD BE: LAI 001. THIS WOULD RESULT IN THE VALUE 001 BEING PLACED IN THE "A" REGISTER WHEN THE INSTRUCTION WAS EXECUTED. IT IS IMPORTANT TO REMEMBER THAT ALL "IMMEDIATE" TYPE INSTRUCTIONS MUST BE FOLLOWED BY A DATA WORD. AN INSTRUCTION SUCH AS LDI ALONE WOULD RESULT IN IMPROPER OPERATION BECAUSE THE COMPUTER WOULD ASSUME THE NEXT WORD CONTAINED DATA, AND IF THE PROGRAMMER HAS MISTAKENLY LEFT OUT THE DATA WORD, AND IN ITS PLACE HAD ANOTHER INSTRUCTION, THE COMPUTER WOULD NOT REALIZE THE OPERATORS "MISTAKE" AND HENCE THE PROGRAM WOULD BE "FOULED-UP!" NOTE TOO, THAT THE LOAD "IMMEDIATE" GROUP OF INSTRUCTIONS DOES NOT AFFECT THE "FLAGS."

LOAD "IMMEDIATE" DATA INTO A MEMORY LOCATION

LMI	0 7 6

THIS INSTRUCTION IS ESSENTIALLY THE SAME AS THE LOAD IMMEDIATE INTO THE CPU REGISTER GROUP EXCEPT THAT NOW, USING THE CONTENTS OF

THE "H" AND "L" REGISTERS AS "POINTERS" TO THE DESIRED ADDRESS IN MEMORY, THE CONTENTS OF THE "IMMEDIATELY FOLLOWING WORD" WILL BE PLACED IN THE MEMORY LOCATION SPECIFIED. THIS INSTRUCTION DOES NOT AFFECT THE STATUS OF THE "FLAGS."

THE ABOVE RATHER LARGE GROUP OF "LOAD" INSTRUCTIONS PERMIT THE PROGRAMMER TO DIRECT THE COMPUTER TO MOVE DATA ABOUT. THEY ARE USED TO BRING IN DATA FROM MEMORY WHERE IT CAN BE OPERATED ON BY THE CPU, OR TO TEMPORARILY STORE INTERMEDIATE RESULTS IN THE CPU REGISTER DURING COMPLICATED AND EXTENDED CALCULATIONS, AND OF COURSE ALLOW DATA, SUCH AS RESULTS, TO BE PLACED BACK INTO MEMORY FOR LONG TERM STORAGE. SINCE NONE OF THEM WILL ALTER THE CONTENTS OF THE FOUR CPU FLAGS, THESE INSTRUCTIONS CAN BE CALLED UPON TO, FOR EXAMPLE, SET UP DATA, BEFORE INSTRUCTIONS THAT MAY AFFECT OR UTILIZE THE FLAGS' STATUS ARE EXECUTED. THE PROGRAMMER WILL USE INSTRUCTIONS FROM THIS SET FREQUENTLY. THE MNEMONIC NAMES FOR THE INSTRUCTIONS ARE EASY TO REMEMBER AS THEY ARE WELL ORDERED. THE MOST IMPORTANT ITEM TO REMEMBER ABOUT THE MNEMONICS IS THAT THE "TO" REGISTER IS ALWAYS INDICATED FIRST IN THE MNEMONIC, AND THEN THE "FROM" REGISTER. THUS "LBA" = "LOAD TO REGISTER "B" FROM REGISTER "A."

INCREMENT THE VALUE OF A CPU REGISTER BY 1

INB	0 1 0
INC	0 2 0
IND	0 3 0
INE	0 4 0
INH	0 5 0
INL	0 6 0

THIS GROUP OF INSTRUCTIONS ALLOWS THE PROGRAMMER TO "ADD 1" TO THE PRESENT VALUE OF ANY OF THE CPU REGISTERS EXCEPT THE ACCUMULATOR. (NOTE CAREFULLY THAT THE ACCUMULATOR CAN NOT BE INCREMENTED BY THIS TYPE OF INSTRUCTION. IN ORDER TO "ADD 1" TO THE ACCUMULATOR A MATHEMATICAL ADDITION INSTRUCTION, DESCRIBED LATER, MUST BE USED). THIS INSTRUCTION FOR INCREMENTING THE DEFINED CPU REGISTERS IS VERY VALUABLE IN A NUMBER OF APPLICATIONS. FOR ONE THING, IT IS AN EASY WAY TO HAVE THE "L" REGISTER SUCCESSIVELY "POINT" TO A STRING OF LOCATIONS IN MEMORY. A FEATURE THAT MAKES THIS TYPE OF INSTRUCTION EVEN MORE POWERFUL, IS THAT THE RESULT OF THE INCREMENTED REGISTER WILL AFFECT THE "Z," "S," AND "P" FLAGS. (IT WILL NOT CHANGE THE "C" OR "CARRY" FLAG). THUS, AFTER A CPU REGISTER HAS BEEN INCREMENTED BY THIS INSTRUCTION, ONE CAN UTILIZE A "FLAG TEST" INSTRUCTION (SUCH AS THE JUMP AND CALL INSTRUCTIONS TO BE DESCRIBED LATER) TO DETERMINE WHETHER THAT PARTICULAR REGISTER HAS A VALUE OF ZERO ("Z" FLAG), OR IF IT IS A NEGATIVE NUMBER ("S" FLAG), OR EVEN PARITY ("P" FLAG). IT IS IMPORTANT TO NOTE THAT THIS GROUP OF INSTRUCTIONS, AND THE DECREMENT GROUP (DESCRIBED IN THE NEXT PARAGRAPH) ARE THE ONLY INSTRUCTIONS WHICH ALLOW THE "FLAGS" TO BE MANIPULATED BY OPERATIONS THAT ARE NOT CONCERNED WITH THE ACCUMULATOR ("A") REGISTER.

DECREMENT THE VALUE OF A CPU REGISTER BY 1

DCB	0 1 1
DCC	0 2 1
DCD	0 3 1
DCE	0 4 1
DCH	0 5 1
DCL	0 6 1

THE DECREMENT GROUP OF INSTRUCTIONS IS SIMILAR TO THE INCREMENT GROUP EXCEPT THAT NOW THE VALUE 1 WILL BE SUBTRACTED FROM THE SPECIFIED CPU REGISTER. THIS INSTRUCTION WILL NOT AFFECT THE "C" FLAG BUT IT DOES AFFECT THE "Z," "S," AND "P" FLAGS. IT SHOULD ALSO BE NOTED THAT THIS GROUP, AS WITH THE INCREMENT GROUP, DOES NOT INCLUDE THE ACCUMULATOR REGISTER. A SEPARATE MATHEMATICAL INSTRUCTION MUST BE USED TO SUBTRACT 1 FROM THE ACCUMULATOR.

ARITHMETIC INSTRUCTIONS USING THE ACCUMULATOR

THE FOLLOWING GROUP OF INSTRUCTIONS ALLOW THE PROGRAMMER TO DIRECT THE COMPUTER TO PERFORM ARITHMETIC OPERATIONS BETWEEN OTHER CPU REGISTERS AND THE ACCUMULATOR, OR BETWEEN THE CONTENTS OF WORDS IN MEMORY AND THE ACCUMULATOR. ALL OF THE OPERATIONS FOR THE DESCRIBED ADDITION, SUBTRACTION, AND COMPARE INSTRUCTIONS AFFECT THE STATUS OF THE "FLAGS."

ADD THE CONTENTS OF A CPU REGISTER TO THE ACCUMULATOR

ADA	2 0 0
ADB	2 0 1
ADC	2 0 2
ADD	2 0 3
ADE	2 0 4
ADH	2 0 5
ADL	2 0 6

THIS GROUP OF INSTRUCTIONS WILL SIMPLY ADD THE PRESENT CONTENTS OF THE ACCUMULATOR REGISTER TO THE PRESENT VALUE OF THE SPECIFIED CPU REGISTER AND LEAVE THE RESULT IN THE ACCUMULATOR. THE VALUE OF THE SPECIFIED REGISTER IS UNCHANGED EXCEPT IN THE CASE OF THE "ADA" INSTRUCTION. NOTE THAT THE "ADA" INSTRUCTION ESSENTIALLY ALLOWS THE PROGRAMMER TO DOUBLE THE VALUE OF THE ACCUMULATOR (WHICH IS THE "A" REGISTER)! IF THE ADDITION CAUSES AN "OVER-FLOW" OR "UNDER-FLOW" THEN THE "CARRY" ("C" FLAG) WILL BE AFFECTED.

ADD THE CONTENTS OF A CPU REGISTER PLUS THE VALUE OF THE CARRY FLAG TO THE ACCUMULATOR

ACA	2 1 0
ACB	2 1 1
ACC	2 1 2
ACD	2 1 3
ACE	2 1 4
ACH	2 1 5
ACL	2 1 6

THIS GROUP IS IDENTICAL TO THE PREVIOUS GROUP EXCEPT THAT NOW THE CONTENT OF THE CARRY FLAG IS CONSIDERED AS AN ADDITIONAL BIT (MSB) IN THE SPECIFIED CPU REGISTER AND THE COMBINED VALUE OF THE CARRY BIT PLUS THE CONTENTS OF THE SPECIFIED CPU REGISTER ARE ADDED TO THE VALUE IN THE ACCUMULATOR. THE RESULTS ARE LEFT IN THE ACCUMULATOR. AGAIN, WITH THE EXCEPTION OF THE "ACA" INSTRUCTION, THE CONTENTS OF THE SPECIFIED CPU REGISTER IS LEFT UNCHANGED. AGAIN TOO, THE CARRY BIT ("C" FLAG) WILL BE AFFECTED BY THE RESULTS OF THE OPERATION.

SUBTRACT THE CONTENTS OF A CPU REGISTER FROM THE ACCUMULATOR

SUA	2 2 Ø
SUB	2 2 1
SUC	2 2 2
SUD	2 2 3
SUE	2 2 4
SUH	2 2 5
SUL	2 2 6

THIS GROUP OF INSTRUCTIONS WILL CAUSE THE PRESENT VALUE OF THE SPECIFIED CPU REGISTER TO BE SUBTRACTED FROM THE VALUE IN THE ACCUMULATOR. THE VALUE OF THE SPECIFIED REGISTER IS NOT CHANGED EXCEPT IN THE CASE OF THE "SUA" INSTRUCTION. (NOTE THAT THE "SUA" INSTRUCTION IS A CONVENIENT INSTRUCTION WITH WHICH TO "CLEAR" THE ACCUMULATOR). THE CARRY FLAG WILL BE AFFECTED BY THE RESULTS OF A SUBTRACT INSTRUCTION.

SUBTRACT THE CONTENTS OF A CPU REGISTER AND THE VALUE OF THE CARRY FLAG FROM THE ACCUMULATOR

SBA	2 3 Ø
SBB	2 3 1
SBC	2 3 2
SBD	2 3 3
SBE	2 3 4
SBH	2 3 5
SBL	2 3 6

THIS GROUP IS IDENTICAL TO THE PREVIOUS GROUP EXCEPT THAT NOW THE CONTENT OF THE CARRY FLAG IS CONSIDERED AS AN ADDITIONAL BIT (MSB) IN THE SPECIFIED CPU REGISTER AND THE COMBINED VALUE OF THE CARRY BIT PLUS THE CONTENTS OF THE SPECIFIED CPU REGISTER ARE SUBTRACTED FROM THE VALUE IN THE ACCUMULATOR. THE RESULTS ARE LEFT IN THE ACCUMULATOR, AND THE CARRY BIT ("C" FLAG) IS AFFECTED BY THE RESULT OF THE OPERATION. WITH THE EXCEPTION OF THE "SBA" INSTRUCTION THE CONTENTS OF THE SPECIFIED CPU REGISTER IS LEFT UNCHANGED.

COMPARE THE VALUE IN THE ACCUMULATOR AGAINST THE CONTENTS OF A CPU REGISTER

CPA	2 7 Ø
CPB	2 7 1
CPC	2 7 2
CPD	2 7 3
CPE	2 7 4
CPH	2 7 5
CPL	2 7 6

THE "COMPARE" GROUP OF INSTRUCTIONS ARE A VERY POWERFUL AND SOMEWHAT UNIQUE SET OF INSTRUCTIONS. THEY DIRECT THE COMPUTER TO COMPARE THE CONTENTS OF THE ACCUMULATOR AGAINST ANOTHER REGISTER AND TO SET THE "FLAGS" AS A RESULT OF THE COMPARING OPERATION. IT IS ESSENTIALLY A SUBTRACTION OPERATION WITH THE VALUE OF THE SPECIFIED REGISTER BEING SUBTRACTED FROM THE VALUE OF THE ACCUMULATOR EXCEPT THAT THE VALUE OF THE ACCUMULATOR IS NOT ACTUALLY ALTERED BY THE OPERATION. HOWEVER, THE "FLAGS" ARE SET IN THE SAME MANNER AS THOUGH AN ACTUAL SUBTRACTION OPERATION HAD OCCURED. THUS, BY SUBSEQUENTLY TESTING THE STATUS OF THE VARIOUS FLAGS AFTER A COM-

PARE INSTRUCTION HAS BEEN EXECUTED, THE PROGRAM CAN DETERMINE WHETHER THE "COMPARE" OPERATION RESULTED IN A MATCH, OR NON-MATCH, AND IN THE CASE OF A NON-MATCH WHETHER THE COMPARED REGISTER CONTAINED A VALUE GREATER OR LESS THAN THAT IN THE ACCUMULATOR. THIS WOULD BE ACCOMPLISHED BY TESTING THE "Z" FLAG AND "C" FLAG RESPECTIVELY UTILIZING A "JUMP" OR "CALL" FLAG TESTING INSTRUCTION (WHICH WILL BE DESCRIBED LATER).

ADDITION, SUBTRACTION, AND COMPARE INSTRUCTIONS THAT USE WORDS IN MEMORY AS OPERANDS

THE FIVE TYPES OF MATHEMATICAL OPERATIONS: ADD, ADD WITH CARRY, SUBTRACT, SUBTRACT WITH CARRY, AND THE COMPARE; WHICH HAVE JUST BEEN PRESENTED FOR PERFORMING THE OPERATIONS WITH THE CONTENTS OF THE CPU REGISTERS, CAN ALL ALSO BE PERFORMED WITH WORDS THAT ARE IN MEMORY. AS WITH THE "LOAD" INSTRUCTIONS WITH MEMORY, THE "H" AND "L" REGISTERS MUST CONTAIN THE ADDRESS OF THE WORD IN MEMORY THAT IT IS DESIRED TO ADD, SUBTRACT, OR COMPARE TO THE ACCUMULATOR. THE SAME CONDITIONS FOR THE OPERATIONS AS WAS DETAILED WHEN USING THE CPU REGISTERS APPLY. THUS, FOR MATHEMATICAL OPERATIONS WITH A WORD IN MEMORY, THE FOLLOWING INSTRUCTIONS ARE USED:

ADD THE CONTENTS OF A MEMORY WORD TO THE ACCUMULATOR

ADM 2 0 7

ADD THE CONTENTS OF A MEMORY WORD PLUS THE VALUE OF THE CARRY FLAG TO THE ACCUMULATOR

ACM 2 1 7

SUBTRACT THE CONTENTS OF A MEMORY WORD FROM THE ACCUMULATOR

SUM 2 2 7

SUBTRACT THE CONTENTS OF A MEMORY WORD AND THE VALUE OF THE CARRY FLAG FROM THE ACCUMULATOR

SBM 2 3 7

COMPARE THE VALUE IN THE ACCUMULATOR AGAINST THE CONTENTS OF A MEMORY WORD

CPM 2 7 7

"IMMEDIATE" TYPE ADDITIONS, SUBTRACTIONS, AND COMPARE INSTRUCTIONS

THE 5 TYPES OF MATHEMATICAL OPERATIONS DISCUSSED CAN ALSO BE PERFORMED WITH THE OPERAND BEING THE WORD OF DATA IMMEDIATELY AFTER THE INSTRUCTION. THIS GROUP OF INSTRUCTIONS IS SIMILAR IN FORMAT TO THE PREVIOUSLY DESCRIBED "LOAD IMMEDIATE" INSTRUCTIONS. THE SAME CONDITIONS FOR THE MATHEMATIC OPERATIONS AS DISCUSSED FOR THE OPERATIONS WITH THE CPU REGISTERS APPLY.

ADD "IMMEDIATE"

ADI Ø Ø 4

ADD WITH CARRY "IMMEDIATE"

ACI Ø 1 4

SUBTRACT "IMMEDIATE"

SUI Ø 2 4

SUBTRACT WITH CARRY "IMMEDIATE"

SBI Ø 3 4

COMPARE "IMMEDIATE"

CPI Ø 7 4

LOGICAL INSTRUCTIONS WITH THE ACCUMULATOR

THERE ARE SEVERAL GROUPS OF INSTRUCTIONS WHICH ALLOW BOOLEAN LOGIC OPERATIONS TO BE PERFORMED BETWEEN THE CONTENTS OF THE CPU REGISTERS AND THE "A" OR ACCUMULATOR REGISTER, AS WELL AS BETWEEN CONTENTS OF LOCATIONS IN MEMORY AND THE "A" REGISTER. IN ADDITION THERE ARE LOGIC "IMMEDIATE" TYPE INSTRUCTIONS. THE BOOLEAN LOGIC OPERATIONS ARE VALUABLE IN A NUMBER OF PROGRAMMING APPLICATIONS. THE INSTRUCTION SET ALLOWS THREE BASIC BOOLEAN OPERATIONS TO BE PERFORMED. THESE ARE THE: "LOGICAL AND;" "LOGICAL OR;" AND "EXCLUSIVE OR" OPERATIONS. EACH TYPE OF LOGIC OPERATION IS PERFORMED ON A "BIT-BY-BIT" BASIS BETWEEN THE ACCUMULATOR REGISTER AND THE CPU REGISTER OR MEMORY LOCATION SPECIFIED BY THE INSTRUCTION. A DETAILED EXPLANATION OF EACH TYPE OF LOGIC OPERATION, AND THE APPROPRIATE INSTRUCTIONS FOR EACH TYPE IS PRESENTED BELOW. THE LOGIC INSTRUCTION SET IS ALSO VALUABLE BECAUSE ALL OF THEM WILL CAUSE THE CARRY ("C") FLAG TO BE SET TO THE "Ø" CONDITION. THIS IS IMPORTANT IF ONE IS GOING TO PERFORM A SEQUENCE OF INSTRUCTIONS THAT WILL EVENTUALLY USE THE STATUS OF THE "C" FLAG TO ARRIVE AT A DECISION AS IT ALLOWS THE PROGRAMMER TO SET THE "C" FLAG TO A KNOWN STATE AT THE START OF THE SEQUENCE. ALL OTHER "FLAGS" ARE SET IN ACCORDANCE WITH RESULT OF THE LOGIC OPERATION AND HENCE THE GROUP OFTEN HAS VALUE WHEN THE PROGRAMMER DESIRES TO DETERMINE THE CONTENTS OF A REGISTER THAT HAS JUST BEEN "LOADED" INTO A REGISTER (SINCE THE "LOAD" INSTRUCTIONS DO NOT AFFECT THE STATE OF THE "FLAGS").

THE BOOLEAN "AND" OPERATION AND INSTRUCTION SET

WHEN THE BOOLEAN "AND" INSTRUCTION IS EXECUTED, EACH BIT OF THE ACCUMULATOR WILL BE COMPARED WITH THE CORRESPONDING BIT IN THE REGISTER OR MEMORY LOCATION SPECIFIED BY THE INSTRUCTION. AS EACH BIT IS COMPARED A LOGIC RESULT WILL BE PLACED IN THE ACCUMULATOR FOR EACH BIT COMPARISON. THE LOGIC RESULT IS DETERMINED AS FOLLOWS: IF BOTH THE BIT IN THE ACCUMULATOR AND THE BIT IN THE REGISTER WITH WHICH THE OPERATION IS BEING PERFORMED ARE A "1," THEN THE ACCUMULATOR BIT

WILL BE LEFT AS A "1." FOR ALL OTHER POSSIBLE COMBINATIONS (I.E., THE ACCUMULATOR BIT = 0 AND THE OTHER REGISTER'S BIT = 1, OR IF THE ACCUMULATOR BIT = 1 AND THE OTHER REGISTER'S BIT = 0, OR IF BOTH THE ACCUMULATOR AND THE OTHER REGISTER HAVE THE PARTICULAR BIT = 0), THEN THE ACCUMULATOR BIT WILL BE SET TO "0." AN EXAMPLE WILL ILLUSTRATE THE LOGICAL "AND" OPERATION:

INITIAL STATE OF THE ACCUMULATOR: 1 0 1 0 1 0 1 0

CONTENTS OF OPERAND REGISTER: 1 1 0 0 1 1 0 0

FINAL STATE OF THE ACCUMULATOR: 1 0 0 0 1 0 0 0

THERE ARE 7 LOGICAL "AND" INSTRUCTIONS THAT ALLOW ANY CPU REGISTER TO BE USED AS THE "AND" OPERAND. THEY ARE AS FOLLOWS:

NDA	2 4 0
NDB	2 4 1
NDC	2 4 2
NDD	2 4 3
NDE	2 4 4
NDH	2 4 5
NDL	2 4 6

THE CONTENTS OF THE OPERAND REGISTER IS NOT ALTERED BY AN "AND" LOGICAL INSTRUCTION.

THERE IS ALSO A LOGICAL "AND" INSTRUCTION THAT ALLOWS A WORD IN MEMORY TO BE USED AS AN OPERAND. THE ADDRESS OF THE WORD IN MEMORY THAT WILL BE USED IS "POINTED TO" BY THE CONTENTS OF THE "H" AND "L" CPU REGISTERS.

NDM 2 4 7

AND FINALLY THERE IS ALSO A LOGICAL "AND" "IMMEDIATE" TYPE OF INSTRUCTION THAT WILL USE THE CONTENTS OF THE WORD IMMEDIATELY FOLLOWING THE INSTRUCTION AS THE OPERAND.

NDI 0 4 4

THE NEXT GROUP OF BOOLEAN LOGIC INSTRUCTIONS DIRECT THE COMPUTER TO PERFORM THE LOGICAL "OR" OPERATION ON A "BIT-BY-BIT" BASIS WITH THE ACCUMULATOR AND THE CONTENTS OF A CPU REGISTER OR A WORD IN MEMORY. THE LOGICAL "OR" OPERATION WILL RESULT IN THE ACCUMULATOR HAVING A BIT SET TO "1" IF EITHER THAT BIT IN THE ACCUMULATOR, OR THE CORRESPONDING BIT IN THE OPERAND REGISTER IS A "1." SINCE THE CASE WHERE BOTH THE ACCUMULATOR BIT AND THE OPERAND BIT IS A "1" ALSO SATISFIES THE RELATIONSHIP, THAT CONDITION WILL ALSO RESULT IN THE ACCUMULATOR BIT BEING A "1." IF NEITHER REGISTER HAS A ONE IN THE BIT POSITION, THEN THE ACCUMULATOR BIT REMAINS "0." AN EXAMPLE ILLUSTRATES THE RESULTS OF A LOGICAL "OR" OPERATION:

INITIAL STATE OF THE ACCUMULATOR: 1 0 1 0 1 0 1 0

CONTENTS OF THE OPERAND REGISTER: 1 1 0 0 1 1 0 0

FINAL STATE OF THE ACCUMULATOR: 1 1 1 0 1 1 1 0

THERE ARE 7 LOGICAL "OR" INSTRUCTIONS THAT ALLOW ANY CPU REGISTER TO BE USED AS THE "OR" OPERAND. THEY ARE:

```
ORA     2 6 0
ORB     2 6 1
ORC     2 6 2
ORD     2 6 3
ORE     2 6 4
ORH     2 6 5
ORL     2 6 6
```

AND, BY USING THE "H" AND "L" REGISTERS AS "POINTERS" ONE CAN ALSO USE A WORD IN MEMORY AS AN "OR" OPERAND:

```
ORM     2 6 7
```

THERE IS ALSO THE LOGICAL "OR" "IMMEDIATE" INSTRUCTION:

```
ORI     0 6 4
```

AS WITH THE LOGICAL "AND" GROUP OF INSTRUCTIONS, THE LOGICAL "OR" INSTRUCTION DOES NOT ALTER THE CONTENTS OF THE OPERAND REGISTER.

THE LAST GROUP OF BOOLEAN LOGIC INSTRUCTIONS IS A VARIATION OF THE LOGIC "OR." THE VARIATION IS TERMED THE LOGICAL "EXCLUSIVE OR." THE "EXCLUSIVE OR" OPERATION IS SIMILAR TO THE "OR" EXCEPT THAT WHEN THE CORRESPONDING BITS IN BOTH THE ACCUMULATOR AND THE OPERAND REGISTER ARE A "1" THEN THE ACCUMULATOR BIT WILL BE SET TO "0." THUS, THE ACCUMULATOR BIT WILL BE A "1" AFTER THE OPERATION ONLY IF JUST ONE OF THE REGISTERS (ACCUMULATOR REGISTER OR OPERAND REGISTER) HAS A "1" IN THE BIT POSITION. (AGAIN, THE OPERATION IS PERFORMED ON A BIT-BY-BIT BASIS). AN EXAMPLE PROVIDES CLARIFICATION:

```
INITIAL STATE OF THE ACCUMULATOR:     1 0 1 0 1 0 1 0

CONTENTS OF THE OPERAND REGISTER:     1 1 0 0 1 1 0 0

FINAL STATE OF THE ACCUMULATOR:       0 1 1 0 0 1 1 0
```

THE 7 INSTRUCTIONS THAT ALLOW THE CPU REGISTERS TO BE USED AS OPERANDS ARE:

```
XRA     2 5 0
XRB     2 5 1
XRC     2 5 2
XRD     2 5 3
XRE     2 5 4
XRH     2 5 5
XRL     2 5 6
```

THE INSTRUCTION THAT USES REGISTERS "H" AND "L" AS POINTERS TO A MEMORY LOCATION IS:

```
XRM     2 5 7
```

AND THE "EXCLUSIVE OR" "IMMEDIATE" TYPE INSTRUCTION IS:

```
XRI     0 5 4
```

AS IN THE CASE OF THE LOGICAL "OR" OPERATION, THE OPERAND REGISTER IS NOT ALTERED EXCEPT FOR THE SPECIAL CASE WHEN THE "XRA" INSTRUCTION IS USED. THIS INSTRUCTION, WHICH DIRECTS THE COMPUTER TO "EXCLUSIVE OR" THE ACCUMULATOR (CPU REGISTER "A") WITH ITSELF, WILL CAUSE THE OPERAND REGISTER - SINCE IT IS ALSO THE ACCUMULATOR, TO HAVE ITS CONTENTS ALTERED (UNLESS IT IS ZERO AT THE TIME THE INSTRUCTION IS ISSUED). THIS IS BECAUSE, REGARDLESS OF WHAT VALUE IS IN THE ACCUMULATOR, IF IT IS "EXCLUSIVE-ORED" WITH ITSELF, THE RESULT WILL ALWAYS BE ZERO! THE EXAMPLE ILLUSTRATES:

ORIGINAL VALUE OF THE ACCUMULATOR:	1 0 1 0 1 0 1 0
"EXCLUSIVE OR" WITH ITSELF:	1 0 1 0 1 0 1 0
FINAL VALUE OF THE ACCUMULATOR:	0 0 0 0 0 0 0 0

THIS ONLY OCCURS WHEN THE LOGICAL "EXCLUSIVE OR" IS PERFORMED ON THE ACCUMULATOR ITSELF. IT CAN BE SHOWN THAT THE RESULTS OF PERFORMING THE LOGICAL "OR" OR LOGICAL "AND" BETWEEN THE ACCUMULATOR AND ITSELF WILL RESULT IN THE ORIGINAL ACCUMULATOR VALUE BEING RETAINED.

INSTRUCTIONS FOR ROTATING THE CONTENTS OF THE ACCUMULATOR

IT IS OFTEN DESIRABLE TO BE ABLE TO "SHIFT" THE CONTENTS OF THE ACCUMULATOR EITHER RIGHT OR LEFT. IN A FIXED LENGTH REGISTER, A SIMPLE SHIFT OPERATION WOULD RESULT IN SOME INFORMATION BEING LOST BECAUSE WHAT WAS IN THE MSB OR LSB (DEPENDING ON IN WHICH DIRECTION THE SHIFT OCCURED) WOULD JUST BE SHIFTED RIGHT OUT OF THE REGISTER! THEREFORE, INSTEAD OF JUST SHIFTING THE CONTENTS OF A REGISTER, AN OPERATION TERMED "ROTATING" IS UTILIZED. NOW, INSTEAD OF JUST SHIFTING A BIT OFF THE END OF THE REGISTER, THE BIT IS BROUGHT AROUND TO THE OTHER END OF THE REGISTER. FOR INSTANCE, IF THE REGISTER IS "ROTATED" TO THE RIGHT, THE LSB (LEAST SIGNIFICANT BIT) WOULD BE BROUGHT AROUND TO THE POSITION OF THE MSB (MOST SIGNIFICANT BIT) IN THE REGISTER WHICH WOULD HAVE BEEN VACATED BY THE SHIFTING OF ITS ORIGINAL CONTENTS TO THE RIGHT. OR, IN THE CASE OF A SHIFT TO THE LEFT, THE MSB WOULD BE BROUGHT AROUND TO THE POSITION OF THE LSB.

SINCE THE CARRY BIT (CARRY OR "C" FLAG) CAN BE CONSIDERED AS AN EXTENSION OF THE ACCUMULATOR REGISTER, IT IS OFTEN DESIRED THAT THE CARRY BIT BE CONSIDERED AS PART OF THE ACCUMULATOR (THE MSB) DURING A ROTATE OPERATION. THE INSTRUCTION SET FOR THIS MACHINE ALLOWS TWO TYPES OF ROTATE INSTRUCTIONS. ONE CONSIDERS THE CARRY BIT TO BE PART OF THE ACCUMULATOR REGISTER FOR THE ROTATE OPERATION, AND THE OTHER TYPE DOES NOT. IN ADDITION, EACH TYPE OF ROTATE CAN BE DONE EITHER TO THE RIGHT, OR TO THE LEFT.

IT SHOULD BE NOTED THAT THE ROTATE OPERATIONS ARE PARTICULARLY VALUABLE WHEN IT IS DESIRED TO MULTIPLY A NUMBER BECAUSE SHIFTING THE CONTENTS OF A REGISTER TO THE LEFT IS A QUICK WAY TO MULTIPLY A BINARY NUMBER BY POWERS OF TWO, AND SHIFTING TO THE RIGHT PROVIDES THE INVERSE OPERATION.

ROTATING THE ACCUMULATOR LEFT

RLC 0 0 2

ROTATATING THE ACCUMULATOR LEFT WITH THE "RLC" INSTRUCTION MEANS THE MSB OF THE ACCUMULATOR WILL BE BROUGHT AROUND TO THE LSB POSITION AND ALL OTHER BITS ARE SHIFTED ONE POSITION TO THE LEFT. WHILE THIS INSTRUCTION DOES NOT SHIFT THROUGH THE CARRY BIT, THE CARRY BIT WILL BE SET BY THE STATUS OF THE MSB OF THE ACCUMULATOR AT THE START OF THE ROTATE OPERATION. (THIS FEATURE ALLOWS THE PROGRAMMER TO DETERMINE WHAT THE MSB WAS PRIOR TO THE SHIFTING OPERATION BY TESTING THE "C" FLAG AFTER THE ROTATE INSTRUCTION HAS BEEN EXECUTED).

ROTATING THE ACCUMULATOR LEFT THROUGH THE CARRY BIT

RAL Ø 2 2

THE "RAL" INSTRUCTION WILL CAUSE THE MSB OF THE ACCUMULATOR TO GO INTO THE CARRY BIT. THE INITIAL VALUE OF THE CARRY BIT WILL BE SHIFTED AROUND TO THE LSB OF THE ACCUMULATOR. ALL OTHER BITS ARE SHIFTED ONE POSITION TO THE LEFT.

ROTATING THE ACCUMULATOR RIGHT

RRC Ø 1 2

THE "RRC" INSTRUCTION IS SIMILAR TO THE "RLC" INSTRUCTION EXCEPT THAT NOW THE LSB OF THE ACCUMULATOR IS PLACED IN THE MSB OF THE ACCUMULATOR AND ALL OTHER BITS ARE SHIFTED ONE POSITION TO THE RIGHT. ALSO, THE CARRY BIT WILL BE SET TO THE INITIAL VALUE OF THE LSB OF THE ACCUMULATOR AT THE START OF THE OPERATION.

ROTATING THE ACCUMULATOR RIGHT THROUGH THE CARRY BIT

RAR Ø 3 2

HERE, THE LSB OF THE ACCUMULATOR IS BROUGHT AROUND TO THE CARRY BIT AND THE INITIAL VALUE OF THE CARRY BIT IS SHIFTED TO THE MSB OF THE ACCUMULATOR. ALL OTHER BITS ARE SHIFTED A POSITION TO THE RIGHT.

IT SHOULD BY NOTED THAT THE "C" FLAG IS THE ONLY FLAG THAT CAN BE ALTERED BY A ROTATE INSTRUCTION. ALL OTHER FLAGS REMAIN UNCHANGED.

JUMP INSTRUCTIONS

THE INSTRUCTIONS DISCUSSED SO FAR HAVE ALL BEEN SORT OF "DIRECT ACTION" INSTRUCTIONS. THE PROGRAMMER ARRANGES A SEQUENCE OF THESE TYPES OF INSTRUCTIONS IN MEMORY AND WHEN THE PROGRAM IS STARTED THE COMPUTER PROCEEDS TO EXECUTE THE INSTRUCTIONS IN THE ORDER IN WHICH THEY ARE ENCOUNTERED. THE COMPUTER AUTOMATICALLY READS THE CONTENTS OF A MEMORY LOCATION, EXECUTES THE INSTRUCTION IT FINDS THERE, AND THEN AUTOMATICALLY INCREMENTS A SPECIAL ADDRESS REGISTER CALLED A "PROGRAM COUNTER" THAT WILL RESULT IN THE MACHINE READING THE INFORMATION CONTAINED IN THE NEXT SEQUENTIAL MEMORY LOCATION. HOWEVER, IT IS OFTEN DESIRABLE TO PERFORM A SERIES OF INSTRUCTIONS LOCATED IN ONE SECTION OF MEMORY, AND THEN SKIP OVER A GROUP OF MEMORY LOCATIONS AND START EXECUTING INSTRUCTIONS IN ANOTHER SECTION OF MEMORY. THIS ACTION CAN BE ACCOMPLISHED BY A GROUP OF INSTRUCTIONS THAT WILL CAUSE A NEW ADDRESS VALUE TO BE PLACED IN THE "PROGRAM COUNTER." THIS WILL CAUSE THE COMPUTER TO GO TO A NEW SECTION OF MEMORY AND TO CONTINUE EXECUTING INSTRUCTIONS SEQUENTIALLY FROM THE NEW MEMORY LOCATION.

THE "JUMP" INSTRUCTIONS IN THIS COMPUTER ADD CONSIDERABLE POWER TO THE MACHINE'S CAPABILITIES BECAUSE THERE ARE A SERIES OF "CONDITIONAL" JUMP INSTRUCTIONS AVAILABLE. THAT IS, THE COMPUTER CAN BE DIRECTED TO TEST THE STATUS OF A PARTICULAR FLAG ("C," "Z," "S," OR "P") AND IF THE STATUS OF THE FLAG IS THE DESIRED ONE, THEN A "JUMP" WILL BE PERFORMED. IF IT IS NOT, THE MACHINE WILL CONTINUE TO EXECUTE THE NEXT INSTRUCTION IN THE CURRENT SEQUENCE. THIS CAPABILITY PROVIDES A MEANS FOR THE COMPUTER TO "MAKE DECISIONS" AND TO MODIFY ITS OPERATION AS A FUNCTION OF THE STATUS OF THE VARIOUS FLAGS AT THE TIME THAT THE PROGRAM IS BEING EXECUTED.

IN A MANNER SIMILAR TO "IMMEDIATE" TYPES OF INSTRUCTIONS, THE "JUMP" INSTRUCTIONS REQUIRE MORE THAN ONE WORD OF MEMORY. A JUMP INSTRUCTION REQUIRES THREE WORDS TO BE PROPERLY DEFINED. (REMEMBER THAT "IMMEDIATE" TYPE INSTRUCTIONS REQUIRED TWO WORDS). THE "JUMP" INSTRUCTION ITSELF IS THE FIRST WORD. THE SECOND WORD MUST CONTAIN THE "LOW ADDRESS" PORTION OF THE ADDRESS OF THE WORD IN MEMORY THAT THE "PROGRAM COUNTER" IS TO BE SET FOR - IN OTHER WORDS, THE NEW LOCATION FROM WHICH THE NEXT INSTRUCTION IS TO BE TAKEN. THE THIRD WORD MUST CONTAIN THE "HIGH ADDRESS" (PAGE) OF THE MEMORY ADDRESS THAT THE "PROGRAM COUNTER" WILL BE SET TO, HENCE, THE "PAGE" OR HIGH ORDER PORTION OF THE ADDRESS THAT THE COMPUTER WILL "JUMP TO" TO OBTAIN ITS NEXT INSTRUCTION.

THE UNCONDITIONAL JUMP INSTRUCTION

JMP	1 X 4

NOTE: THE MACHINE CODE 1 X 4 INDICATES THAT ANY CODE FOR THE SECOND OCTAL DIGIT OF THE MACHINE CODE IS VALID. IT IS RECOMMENDED AS A STANDARD PRACTICE THAT THE CODE Ø BE USED THUS THE TYPICAL MACHINE CODE WOULD BE 1 Ø 4.

REMEMBER, THE JUMP INSTRUCTION MUST BE FOLLOWED BY TWO MORE WORDS WHICH CONTAIN THE LOW, AND THEN THE HIGH (PAGE) PORTION OF THE ADDRESS THAT THE PROGRAM IS TO "JUMP" TO!

JUMP IF THE DESIGNATED FLAG IS TRUE (CONDITIONAL JUMP)

JTC	1 4 Ø
JTZ	1 5 Ø
JTS	1 6 Ø
JTP	1 7 Ø

AS WITH THE UNCONDITIONAL JUMP INSTRUCTION, THE CONDITIONAL JUMP INSTRUCTIONS MUST BE FOLLOWED BY TWO WORDS OF INFORMATION - THE LOW PORTION, THEN THE HIGH PORTION, OF THE ADDRESS THAT PROGRAM EXECUTION IS TO CONTINUE FROM IF THE JUMP IS EXECUTED. THE "JUMP IF TRUE" GROUP OF INSTRUCTIONS WILL ONLY JUMP TO THE DESIGNATED ADDRESS IF THE CONDITION OF THE APPROPRIATE FLAG IS TRUE (LOGICAL "1"). THUS THE "JTC" INSTRUCTION STATES THAT IF THE CARRY FLAG ("C") IS A LOGICAL "1" (TRUE) THEN THE JUMP IS TO BE EXECUTED. IF IT IS A LOGICAL "Ø" (FALSE) THEN PROGRAM EXECUTION IS TO CONTINUE WITH THE NEXT INSTRUCTION IN THE CURRENT SEQUENCE OF INSTRUCTIONS. IN A SIMILAR MANNER THE "JTZ" INSTRUCTION STATES THAT IF THE ZERO FLAG IS TRUE THEN THE JUMP IS TO BE PERFORMED. OTHERWISE THE NEXT INSTRUCTION IN THE PRESENT SEQUENCE IS EXECUTED. LIKEWISE FOR THE "JTS" AND "JTP" INSTRUCTIONS.

JUMP IF THE DESIGNATED FLAG IS FALSE (CONDITIONAL JUMP)

JFC	1 Ø Ø
JFZ	1 1 Ø
JFS	1 2 Ø
JFP	1 3 Ø

AS WITH ALL JUMP INSTRUCTIONS THESE INSTRUCTIONS MUST BE FOLLOWED BY THE LOW ADDRESS THEN HIGH ADDRESS OF THE MEMORY LOCATION THAT PROGRAM EXECUTION IS TO CONTINUE FROM IF THE JUMP IS EXECUTED. THIS GROUP OF INSTRUCTIONS IS THE OPPOSITE OF THE JUMP IF THE FLAG IS TRUE GROUP. FOR INSTANCE THE "JFC" INSTRUCTION COMMANDS THE COMPUTER TO TEST THE STATUS OF THE CARRY ("C") FLAG. IF THE FLAG IS "FALSE," I.E. A LOGIC "Ø," THEN THE JUMP IS TO BE PERFORMED. IF IT IS "TRUE" THEN PROGRAM EXECUTION IS TO CONTINUE WITH THE NEXT INSTRUCTION IN THE CURRENT SEQUENCE OF INSTRUCTIONS. THE SAME PROCEDURE HOLDS FOR THE "JFZ," "JFS," AND "JFP" INSTRUCTIONS.

SUBROUTINE CALLING INSTRUCTIONS

QUITE OFTEN WHEN A PROGRAMMER IS DEVELOPING COMPUTER PROGRAMS THE PROGRAMMER WILL FIND THAT A PARTICULAR ALGORITHM (SEQEUNCE OF INSTRUCTIONS FOR PERFORMING A FUNCTION) CAN BE USED MANY TIMES IN DIFFERENT PARTS OF THE PROGRAM. RATHER THAN HAVE TO KEEP ENTERING THE SAME SEQUENCE OF INSTRUCTIONS AT DIFFERENT LOCATIONS IN MEMORY - WHICH WOULD NOT ONLY CONSUME THE TIME OF THE PROGRAMMER BUT WOULD ALSO RESULT IN A LOT OF MEMORY BEING USED TO PERFORM ONE PARTICULAR FUNCTION, IT IS DESIRABLE TO BE ABLE TO PUT AN OFTEN USED SEQUENCE OF COMMANDS IN ONE LOCATION IN MEMORY. THEN, WHENEVER THE PARTICULAR ALGORITHM IS REQUIRED BY ANOTHER PART OF THE PROGRAM, IT WOULD BE CONVENIENT TO "JUMP" TO THE SECTION THAT CONTAINED THE OFTEN USED ALGORITHM, PERFORM THE SEQUENCE OF INSTRUCTIONS, AND THEN RETURN BACK TO THE "MAIN" PART OF THE PROGRAM. THIS IS A STANDARD PRACTICE IN COMPUTER OPERATIONS. THE FREQUENTLY USED ALGORITHM CAN BE DESIGNATED AS A "SUBROUTINE." A SPECIAL SET OF INSTRUCTIONS ALLOWS THE PROGRAMMER TO "CALL" - IN OTHER WORDS SPECIFY A SPECIAL TYPE OF "JUMP TO," A SUBROUTINE. A SECOND TYPE OF INSTRUCTION IS USED TO TERMINATE A SEQUENCE OF INSTRUCTIONS THAT IS TO BE CONSIDERED A SUBROUTINE. THIS SPECIAL TERMINATOR WILL CAUSE THE PROGRAM OPERATION TO REVERT BACK TO THE NEXT SEQUENTIAL LOCATION IN MEMORY FOLLOWING THE INSTRUCTION THAT "CALLED" THE "SUBROUTINE." A GREAT DEAL OF COMPUTER POWER IS PROVIDED BY THE INSTRUCTION SET IN THIS MACHINE FOR "CALLING" AND "RETURNING" FROM SUBROUTINES. THIS IS BECAUSE, IN A MANNER SIMILAR TO THE CONDITIONAL JUMP INSTRUCTIONS, THERE ARE A NUMBER OF "CONDITIONAL CALLING" COMMANDS AND A NUMBER OF "CONDITIONAL RETURN" COMMANDS IN THE INSTRUCTION SET.

LIKE THE "JUMP" INSTRUCTIONS, THE "CALL" INSTRUCTIONS ALL REQUIRE THREE WORDS IN ORDER TO BE FULLY SPECIFIED. THE FIRST WORD IS THE "CALL" INSTRUCTION ITSELF. THE NEXT TWO WORDS MUST CONTAIN THE LOW AND HIGH PORTIONS OF THE STARTING ADDRESS OF THE SUBROUTINE THAT IS BEING "CALLED."

WHEN A "CALL" INSTRUCTION IS ENCOUNTERED BY THE COMPUTER, THE "CPU" WILL ACTUALLY SAVE THE CURRENT VALUE OF ITS PROGRAM COUNTER BY STORING IT IN A SPECIAL "PROGRAM COUNTER PUSH-DOWN STACK." THIS STACK IS CAPABLE OF HOLDING 7 ADDRESSES PLUS THE CURRENT OPERATING ADDRESS. WHAT THIS MEANS IS THAT THE MACHINE IS CAPABLE OF "NESTING" UP TO 7 SUBROUTINES AT ANY ONE TIME. THUS ONE CAN HAVE A SUBROUTINE, THAT IN TURN CALLS ANOTHER SUBROUTINE - THAT IN TURN CALLS ANOTHER ONE, UP TO 7 LEVELS AND THE MACHINE WILL BE ABLE TO "RETURN" TO THE INITIAL

LOCATION. THE PROGRAMMER MUST ENSURE THAT SUBROUTINES ARE NOT "NESTED" AT MORE THAN 7 LEVELS OTHERWISE THE "PROGRAM COUNTER PUSH-DOWN STACK" WILL "PUSH" THE ORIGINAL CALLING ADDRESS(ES) COMPLETELY OUT OF THE "PUSH-DOWN STACK" AND THE PROGRAM COULD NO LONGER AUTOMATICALLY RETURN TO THE INITIAL "CALLING" ROUTINE.

THE "RETURN" INSTRUCTION WHICH TERMINATES A SUBROUTINE ONLY REQUIRES ONE WORD. WHEN THE CPU ENCOUNTERS A "RETURN" INSTRUCTION IT CAUSES THE "PROGRAM COUNTER PUSH-DOWN STACK" TO "POP" UP ONE LEVEL. THIS EFFECTIVELY CAUSES THE ADDRESS "SAVED" IN THE STACK BY THE CALLING ROUTINE TO BE TAKEN AS THE NEW "PROGRAM COUNTER" AND HENCE PROGRAM EXECUTION RETURNS TO THE CALLING ROUTINE.

THE UNCONDITIONAL CALL INSTRUCTION

CAL 1 X 6

THIS INSTRUCTION FOLLOWED BY TWO WORDS CONTAINING THE LOW AND THEN THE HIGH ORDER OF THE STARTING ADDRESS OF THE SUBROUTINE THAT IS TO BE EXECUTED IS AN UNCONDITIONAL "CALL." THE SUBROUTINE WILL BE EXECUTED REGARDLESS OF THE STATUS OF THE "FLAGS." THE NEXT SEQUENTIAL ADDRESS AFTER THE "CAL" INSTRUCTION IS SAVED IN THE "PROGRAM COUNTER PUSH-DOWN STACK."

THE UNCONDITIONAL RETURN INSTRUCTION

RET Ø X 7

THIS INSTRUCTION DIRECTS THE CPU TO UNCONDITIONALLY "POP" THE "PROGRAM COUNTER PUSH-DOWN STACK" UP ONE LEVEL. THUS PROGRAM EXECUTION WILL CONTINUE FROM THE ADDRESS SAVED BY THE SUBROUTINE CALLING INSTRUCTION.

CALL A SUBROUTINE IF THE DESIGNATED FLAG IS TRUE

CTC	1 4 2
CTZ	1 5 2
CTS	1 6 2
CTP	1 7 2

IN A MANNER SIMILAR TO THE CONDITIONAL "JUMP IF TRUE" INSTRUCTIONS THESE INSTRUCTIONS (WHICH MUST ALL BE FOLLOWED BY THE LOW AND HIGH PORTIONS OF THE CALLED SUBROUTINE'S STARTING ADDRESS) WILL ONLY PERFORM THE "CALL" IF THE DESIGNATED FLAG IS IN THE TRUE (LOGICAL "1") STATE. IF THE DESIGNATED FLAG IS FALSE THEN THE "CALL" INSTRUCTION IS IGNORED AND PROGRAM EXECUTION CONTINUES WITH THE NEXT SEQUENTIAL INSTRUCTION.

RETURN FROM A SUBROUTINE IF THE DESIGNATED FLAG IS TRUE

RTC	Ø 4 3
RTZ	Ø 5 3
RTS	Ø 6 3
RTP	Ø 7 3

THESE ONE WORD INSTRUCTIONS WILL CAUSE A SUBROUTINE TO BE TERMINATED ONLY IF THE DESIGNATED FLAG IS IN THE LOGICAL "1" (TRUE) STATE.

CALL A SUBROUTINE IF THE DESIGNATED FLAG IS FALSE

CFC	1 0 2
CFZ	1 1 2
CFS	1 2 2
CFP	1 3 2

THESE INSTRUCTIONS ARE THE OPPOSITE OF THE PREVIOUS GROUP OF CALLING COMMANDS. THE SUBROUTINE IS CALLED ONLY IF THE DESIGNATED FLAG IS IN THE FALSE (LOGICAL 0) CONDITION. REMEMBER, THESE INSTRUCTIONS MUST BE FOLLOWED BY TWO WORDS WHICH CONTAIN THE LOW AND THEN HIGH PART OF THE STARTING ADDRESS OF THE SUBROUTINE THAT IS TO BE EXECUTED IF THE DESIGNATED FLAG IS FALSE. IF THE FLAG IS TRUE, THE SUBROUTINE WILL NOT BE CALLED AND PROGRAM OPERATION WILL CONTINUE WITH THE NEXT INSTRUCTION IN THE CURRENT SEQUENCE.

RETURN FROM A SUBROUTINE IF THE DESIGNATED FLAG IS FALSE

RFC	0 0 3
RFZ	0 1 3
RFS	0 2 3
RFP	0 3 3

THESE ONE WORD INSTRUCTIONS WILL TERMINATE A SUBROUTINE (POP THE "PROGRAM COUNTER STACK" UP ONE LEVEL) IF THE DESIGNATED FLAG IS FALSE. OTHERWISE THE INSTRUCTION IS IGNORED AND PROGRAM OPERATION IS CONTINUED WITH THE NEXT INSTRUCTION IN THE SUBROUTINE.

THE SPECIAL "RESTART" SUBROUTINE CALL INSTRUCTIONS

THERE IS A SPECIAL PURPOSE INSTRUCTION AVAILABLE THAT EFFECTIVELY SERVES AS A ONE WORD SUBROUTINE CALL (REMEMBER THAT IT NORMALLY REQUIRES THREE WORDS TO SPECIFY A SUBROUTINE CALL). THIS SPECIAL INSTRUCTION ALLOWS THE PROGRAMMER TO CALL A SUBROUTINE THAT STARTS AT ANY ONE OF EIGHT SPECIALLY DESIGNATED MEMORY LOCATIONS. THE EIGHT SPECIAL MEMORY LOCATIONS ARE AT LOCATIONS: 000, 010, 020, 030, 040, 050, 060 AND 070 ON PAGE ZERO. THERE ARE EIGHT VARIATIONS OF THE RESTART INSTRUCTION - ONE FOR EACH OF THE ABOVE ADDRESSES. THUS, THE ONE WORD INSTRUCTION CAN SERVE TO "CALL" A SUBROUTINE AT THE SPECIFIED STARTING LOCATION (INSTEAD OF HAVING TWO ADDITIONAL WORDS TO SPECIFY THE STARTING ADDRESS OF THE SUBROUTINE.) IT IS OFTEN CONVENIENT TO UTILIZE A RESTART COMMAND AS A QUICK "CALL" TO AN OFTEN USED SUBROUTINE, OR AS AN EASY WAY TO CALL SHORT "STARTING" ROUTINES FOR LARGE PROGRAMS - HENCE THE NAME FOR THE TYPE OF INSTRUCTION. THE EIGHT RESTART INSTRUCTIONS - ALONG WITH THE STARTING ADDRESS OF THE SUBROUTINE THAT EACH WILL AUTOMATICALLY "CALL" IS AS FOLLOWS:

INSTRUCTION (MNEMONIC)	MACHINE CODE	SUBROUTINE STARTING ADDRESS
RST 0	0 0 5	000 000
RST 1	0 1 5	000 010
RST 2	0 2 5	000 020
RST 3	0 3 5	000 030
RST 4	0 4 5	000 040
RST 5	0 5 5	000 050
RST 6	0 6 5	000 060
RST 7	0 7 5	000 070

INPUT INSTRUCTIONS

IN ORDER TO RECEIVE INFORMATION FROM AN EXTERNAL DEVICE THE COMPUTER MUST UTILIZE A GROUP OF SPECIAL SIGNAL LINES. THE TYPICAL 8008 COMPUTER IS DESIGNED TO HANDLE UP TO EIGHT GROUPS (EACH GROUP HAVING EIGHT SIGNAL LINES) OF INPUT SIGNALS. A GROUP OF SIGNALS IS ACCEPTED AT THE COMPUTER BY WHAT IS REFERRED TO AS AN "INPUT PORT." THE COMPUTER CONTROLS THE OPERATION OF THE "INPUT PORTS." UNDER PROGRAM CONTROL, THE COMPUTER CAN BE DIRECTED TO OBTAIN THE INFORMATION THAT IS ON THE GROUP OF LINES COMING IN TO ANY "INPUT PORT" AND BRING IT INTO THE ACCUMULATOR. VARIOUS TYPES OF EXTERNAL EQUIPMENT - SUCH AS A KEYBOARD - CAN BE CONNECTED TO THE INPUT PORT(S). WHEN IT IS DESIRED TO HAVE INFORMATION OBTAINED FROM A SPECIFIC "INPUT PORT" AN INPUT INSTRUCTION MUST BE USED. THE INPUT INSTRUCTION SIMPLY IDENTIFIES WHICH INPUT PORT IS TO BE OPERATED AND WHEN EXECUTED CAUSES THE SIGNAL LEVELS ON THE SELECTED INPUT PORT TO BE BROUGHT INTO THE "A" CPU REGISTER (ACCUMULATOR). UP TO 8 INPUT PORTS MAY BE PROVIDED ON A TYPICAL 8008 SYSTEM DESIGNATED PORTS 0 - 7. (NOTE THAT THE MACHINE CODE FOR AN INPUT PORT INCREASES BY A FACTOR OF TWO FOR EACH AVAILABLE PORT).

INP 0	1 0 1
INP 1	1 0 3
.	.
INP 6	1 1 5
INP 7	1 1 7

AN INPUT INSTRUCTION ONLY REQUIRES ONE MACHINE CODE WORD. IT IS ALSO IMPORTANT TO NOTE THAT AN INPUT INSTRUCTION - WHICH BRINGS NEW DATA INTO THE ACCUMULATOR - DOES NOT AFFECT THE STATUS OF ANY OF THE CPU FLAGS.

OUTPUT INSTRUCTIONS

IN ORDER TO OUTPUT INFORMATION TO AN EXTERNAL DEVICE THE COMPUTER UTILIZES ANOTHER GROUP OF SIGNAL LINES WHICH ARE REFERRED TO AS "OUTPUT PORTS." A TYPICAL 8008 SYSTEM MAY BE EQUIPPED TO SERVICE UP TO 24 "OUTPUT PORTS." (EACH OUTPUT PORT ACTUALLY CONSIST OF EIGHT SIGNAL LINES). AN OUTPUT INSTRUCTION CAUSES THE CONTENTS OF THE CPU "A" REGISTER (ACCUMULATOR) TO BE TRANSFERRED TO THE SIGNAL LINES OF THE DESIGNATED OUTPUT PORT. THE OUTPUT PORTS ARE NORMALLY DESIGNATED PORTS 10 - 37. (NOTE AGAIN THAT THE MACHINE CODE INCREASES BY A FACTOR OF TWO FOR EACH DESIGNATED PORT).

OUT 10	1 2 1
OUT 11	1 2 3
.	.
OUT 21	1 4 1
.	.
OUT 36	1 7 5
OUT 37	1 7 7

AN OUTPUT INSTRUCTION ONLY REQUIRES ONE MACHINE CODE WORD. IT DOES NOT AFFECT THE STATUS OF ANY OF THE CPU FLAGS. OUTPUT PORT(S) ARE CONNECTED TO EXTERNAL DEVICES - SUCH AS AN OSCILLOSCOPE DISPLAY SYSTEM, AND PROVIDE CAPABILITY FOR THE COMPUTER TO DISPLAY INFORMATION OR OTHERWISE CONTROL THE OPERATION OF EXTERNAL DEVICES.

THE HALT INSTRUCTION

THERE IS ONE MORE INSTRUCTION FOR THE COMPUTER'S INSTRUCTION SET. THIS INSTRUCTION DIRECTS THE CPU TO STOP ALL OPERATIONS AND TO REMAIN IN THAT STATE UNTIL AN "INTERRUPT" SIGNAL IS RECEIVED. IN A TYPICAL 8008 SYSTEM AN "INTERRUPT" SIGNAL MAY BE GENERATED BY AN OPERATOR PRESSING A SWITCH OR BY AN EXTERNAL PIECE OF EQUIPMENT. THIS INSTRUCTION IS NORMALLY USED WHEN THE PROGRAMMER DESIRES TO HAVE A PROGRAM BE TERMINATED, OR WHEN IT IS DESIRED TO HAVE THE MACHINE WAIT FOR AN OPERATOR TO SET UP EXTERNAL CONDITIONS. THERE ARE THREE MACHINE CODE INSTRUCTIONS THAT MAY BE USED FOR THE HALT COMMAND:

HLT	0 0 0
HLT	0 0 1
HLT	3 7 7

THE HALT INSTRUCTION DOES NOT AFFECT THE STATUS OF THE CPU FLAGS. IT IS A ONE WORD INSTRUCTION.

INFORMATION ON INSTRUCTION EXECUTION TIMES

WHEN PROGRAMMING FOR REAL-TIME APPLICATIONS IT IS IMPORTANT TO KNOW HOW MUCH TIME EACH TYPE OF INSTRUCTION REQUIRES TO BE EXECUTED. WITH THIS INFORMATION THE PROGRAMMER CAN DEVELOP "TIMING LOOPS" OR DETERMINE WITH SUBSTANTUAL ACCURACY HOW MUCH TIME IT TAKES TO PERFORM A PARTICULAR SERIES OF INSTRUCTIONS. THIS INFORMATION IS ESPECIALLY IMPORTANT WHEN DEALING WITH PROGRAMS THAT CONTROL THE OPERATION OF EXTERNAL DEVICES WHICH REQUIRE EVENTS TO OCCUR AT SPECIFIC TIMES.

THE FOLLOWING TABLE PROVIDES THE NOMINAL INSTRUCTION EXECUTION TIME FOR EACH CATEGORY OF INSTRUCTION USED IN A 8008 SYSTEM. THE PRECISE TIME NEEDED FOR EACH INSTRUCTION DEPENDS ON HOW CLOSE THE MASTER CLOCK HAS BEEN SET TO THE NOMINAL VALUE OF 500 KHZ. THE TABLE SHOWS THE NUMBER OF CYCLE STATES REQUIRED BY THE TYPE OF INSTRUCTION FOLLOWED BY THE NOMINAL TIME REQUIRED TO PERFORM THE ENTIRE INSTRUCTION. SINCE EACH STATE EXECUTES IN 4 MICROSECONDS (U'SECS) THE TOTAL TIME REQUIRED TO PERFORM THE INSTRUCTION AS SHOWN IN THE TABLE IS OBTAINED BY MULTIPLYING THE NUMBER OF STATES BY 4 MICROSECONDS. BY KNOWING THE NUMBER OF STATES REQUIRED FOR EACH INSTRUCTION THE PROGRAMMER CAN OFTEN REARRANGE AN ALGORITHM OR SUBSTITUTE DIFFERENT TYPES OF INSTRUCTIONS TO PROVIDE PROGRAMS THAT HAVE SPECIFIC EVENTS OCCURRING AT PRECISELY TIMED INTERVALS.

INSTRUCTION EXECUTION TIME TABLE

TYPE OF INSTRUCTION	# OF STATES	TOTAL EXECUTION TIME
LOAD DATA FROM ONE CPU REGISTER TO ANOTHER CPU REGISTER	5	20 U'SECS
LOAD DATA FROM A CPU REGISTER TO A LOCATION IN MEMORY	7	28 U'SECS

INSTRUCTION EXECUTION TIME TABLE

TYPE OF INSTRUCTION	# OF STATES	TOTAL EXECUTION TIME
LOAD DATA FROM A LOCATION IN MEMORY TO A CPU REGISTER	8	32 U'SECS
LOAD "IMMEDIATE" DATA INTO A CPU REGISTER	8	32 U'SECS
LOAD "IMMEDIATE" DATA INTO A LOCATION IN MEMORY	9	36 U'SECS
INCREMENT OR DECREMENT A CPU REGISTER	5	20 U'SECS
ARITHMETIC INSTRUCTION BETWEEN THE ACCUMULATOR AND A CPU REGISTER	5	20 U'SECS
COMPARE BETWEEN THE ACCUMULATOR AND A CPU REGISTER	5	20 U'SECS
ARITHMETIC OR COMPARE INSTRUCTION BETWEEN THE ACCUMULATOR AND A WORD IN MEMORY	8	32 U'SECS
"IMMEDIATE" TYPE ARITHMETIC AND COMPARE INSTRUCTIONS	8	32 U'SECS
BOOLEAN MATH OPERATIONS BETWEEN ACCUMULATOR AND CPU REGISTERS	5	20 U'SECS

INSTRUCTION EXECUTION TIME TABLE

TYPE OF INSTRUCTION	# OF STATES	TOTAL EXECUTION TIME
BOOLEAN MATH OPERATIONS BETWEEN ACCUMULATOR AND A LOCATION IN MEMORY	8	32 U'SECS
BOOLEAN "IMMEDIATE" INSTRUCTIONS	8	32 U'SECS
ACCUMULATOR ROTATE INSTRUCTIONS	5	20 U'SECS
UNCONDITIONAL JUMP OR CALL INSTRUCTIONS	11	44 U'SECS
CONDITIONAL JUMP OR CALL INSTRUCTIONS WHEN CONDITION IS NOT SATISFIED	9	36 U'SECS
AND CONDITIONAL JUMP OR CALL INSTRUCTIONS WHEN CONDITION IS SATISFIED	11	44 U'SECS
UNCONDITIONAL RETURN INSTRUCTION	5	20 U'SECS
CONDITIONAL RETURN INSTRUCTION WHEN CONDITION IS NOT SATISFIED	3	12 U'SECS
CONDITIONAL RETURN INSTRUCTION WHEN CONDITION IS SATISFIED	5	20 U'SECS
RESTART INSTRUCTION	5	20 U'SECS
OUTPUT INSTRUCTION	6	24 U'SECS
INPUT INSTRUCTION	8	32 U'SECS
HALT INSTRUCTION	4	16 U'SECS

INITIAL STEPS FOR DEVELOPING PROGRAMS

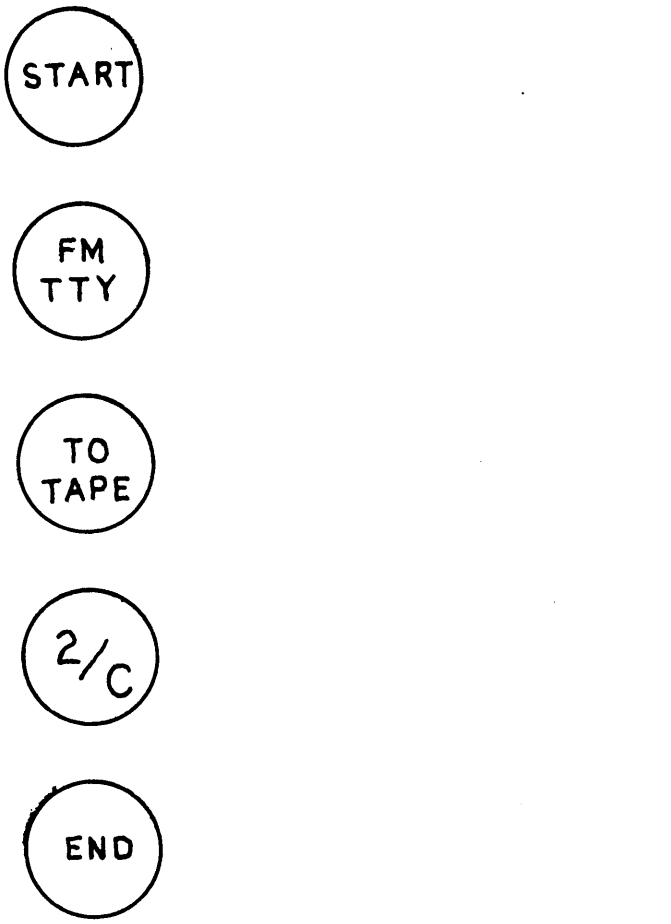
THE FIRST TASK THAT SHOULD BE DONE PRIOR TO STARTING TO WRITE THE INDIVIDUAL INSTRUCTIONS FOR A COMPUTER PROGRAM IS TO DECIDE EXACTLY WHAT IT IS THAT THE COMPUTER IS TO PERFORM AND TO WRITE THE GOAL(S) DOWN ON PAPER! WHILE THIS STATEMENT MIGHT SEEM UNNECESSARY TO SOME BECAUSE IT IS SUCH AN OBVIOUS ONE, IT IS STATED, AND WILL BE RESTATED BECAUSE THE MAJORITY OF PEOPLE LEARNING TO DEVELOP PROGRAMS WILL SOON COME TO REALIZE THE SIGNIFICANCE OF THE ABOVE STATEMENT WHEN THEY DISCOVER HALFWAY THROUGH THE WRITING OF THE MACHINE LANGUAGE INSTRUCTIONS THAT THEY LEFT OUT A VITAL STEP - AND OFTEN HAVE TO PRACTICALLY START WRITING THE PROGRAM ALL OVER. THE PRACTICE OF WRITING DOWN JUST WHAT TASKS A PRTICULAR PROGRAM IS TO PERFORM AND THE STEPS IN WHICH THEY ARE TO BE DONE WILL SAVE A LOT OF WORK IN THE LONG RUN. THE WRITTEN DESCRIPTION SHOULD BE AS COMPLETE AND DETAILED AS NECESSARY FOR THE INDIVIDUAL TO ENSURE THAT EXACTLY EACH STEP OF THE PROGRAM WILL BE CLEAR TO THE PERSON WHEN ACTUALLY WRITING THE PROGRAM IN MACHINE LANGUAGE. IT IS GENERALLY WISE FOR A NOVICE PROGRAMMER TO TAKE PAINS TO BE QUITE DETAILED IN THE INITIAL DESCRIPTION.

THE ACT OF ACTUALLY WRITING DOWN THE PROPOSED OPERATION OF THE PROGRAM SERVES SEVERAL VALUABLE PURPOSES. FIRST, IT FORCES ONE TO CAREFULLY REVIEW WHAT IS PLANNED AND OFTEN VIVIDLY REVEALS FLAWS IN ORIGINAL MENTAL IDEAS. SECONDLY, IT SERVES AS A GUIDE AND A CHECK LIST AS THE MACHINE LANGUAGE PROGRAM IS DEVELOPED. REMEMBER, IT WILL OFTEN TAKE A NUMBER OF HOURS TO COMPLETELY WRITE A FAIR SIZED PROGRAM - AND THESE HOURS MIGHT BE SPREAD OVER SEVERAL DAYS OR WEEKS. IN THIS PERIOD OF TIME THE HUMAN MIND CAN EASILY FORGET ORIGINAL INTENTIONS AND PLANS IF THE HUMAN "MEMORY" CANNOT BE REFRESHED BY WRITTEN NOTES. A PROGRAM THAT IS NOT KEPT CAREFULLY ORGANIZED AS IT IS DEVELOPED CAN BECOME A REAL MESS IF ONE KEEPS FORGETTING KEY CONCEPTS OR HAS TO CONSTANTLY ADD IN "FORGOTTEN" ROUTINES. THE TIME WASTED BY SUCH SLOPPY PROCEDURES CAN BE AVOIDED IF PROPER WORK HABITS ARE DEVELOPED RIGHT FROM THE BEGINNING.

ONCE ONE HAS WRITTEN A DESCRIPTION OF THE GENERAL TASK(S) TO BE PERFORMED, AND HAS ASCERTAINED THAT THERE ARE NO FLAWS TO THE OVER-ALL CONCEPTS OR IDEAS, IT IS A GOOD IDEA TO DRAW UP A SET OF "FLOW CHARTS" FOR THE PROPOSED PROGRAM. THE FLOW CHARTS ARE MORE DETAILED WRITTEN AND SYMBOLIC DESCRIPTIVE DIAGRAMS OF THE "FLOW" OF OPERATIONS THAT ARE TO OCCUR AS THE PROGRAM IS OPERATED. THEY ALSO SHOW THE INTER-RELATIONSHIPS BETWEEN DIFFERENT PORTIONS OF THE PROGRAM.

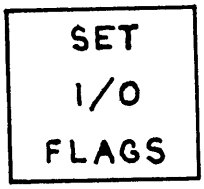
OVER THE YEARS A VARIETY OF SYMBOLS AND METHODS HAVE BEEN DEVELOPED FOR PRODUCING FLOW CHARTS. ALL OF THE VARIETIES HAVE THE SAME BASIC PURPOSE AND MOST OF THE DIFFERENCES ARE THE RESULT OF EDUCATIONALIST PUSHING THEIR OWN PREFERENCES. MOST PEOPLE CAN DO ADMIRABLY WELL USING JUST A FEW BASIC SYMBOLS TO DONOTE BASIC TYPES OF OPERATIONS IN A COMPUTER PROGRAM. THE SMALL GROUP TO BE PRESENTED HERE WILL ENABLE MOST 8008 PROGRAMMERS TO DEVELOP FLOW CHARTS RAPIDLY, WITH LITTLE CONFUSION, AND WITHOUT HAVING TO LEARN A HOST OF "SPECIAL" SYMBOLS.

A CIRCLE CAN BE USED AS A GENERAL PURPOSE SYMBOL TO SPECIFY THE ENTRY OR EXIT POINT TO A ROUTINE OR SUBROUTINE. INFORMATION MAY BE PRINTED INSIDE THE CIRCLE AND MIGHT DENOTE WHERE THE ROUTINE IS COMING FROM OR GOING TO (SUCH AS THE PAGE NUMBER AND LOCATION ON A PAGE FOR A PROGRAM THAT REQUIRES SEVERAL SHEETS OF PAPER TO BE FLOW CHARTED) OR IT CAN CONTAIN TRANSFER INFORMATION OR DENOTE STARTING OR STOPPING POINTS WITHIN A PROGRAM. SOME TYPICAL EXAMPLES OF THE CIRCLE SYMBOL ARE ILLUSTRATED ON THE NEXT PAGE.


START
FM TTY
TO TAPE
2/C
END

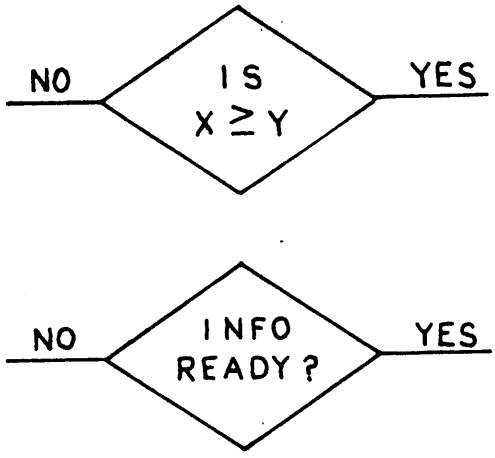

CLR THE ACC


STORE THE
INCOMING
MESSAGE


SET
I/O
FLAGS

A SQUARE OR RECTANGEL CAN BE USED TO DENOTE A GENERAL OR SPECIFIC OPERATION. THE TYPE OF OPERATION CAN BE DESCRIBED INSIDE THE BOXED AREA SUCH AS IN THE EXAMPLES ON THE LOWER HALF OF THE PREVIOUS PAGE.

A DIAMOND FORM MAY BE USED TO SYMBOLIZE A DECISION OR BRANCHING POINT IN A PROGRAM. THE DETERMINING FACTOR(S) FOR THE DECISION OR BRANCHING OPERATION MAY BE INDICATED INSIDE THE SYMBOL AND THE TWO SIDE POINTS OF THE TRIANGLE USED TO ILLUSTRATE THE PATH TAKEN WHEN A DECISION HAS BEEN MADE. THE DIAMOND SYMBOL IS ILLUSTRATED BELOW.



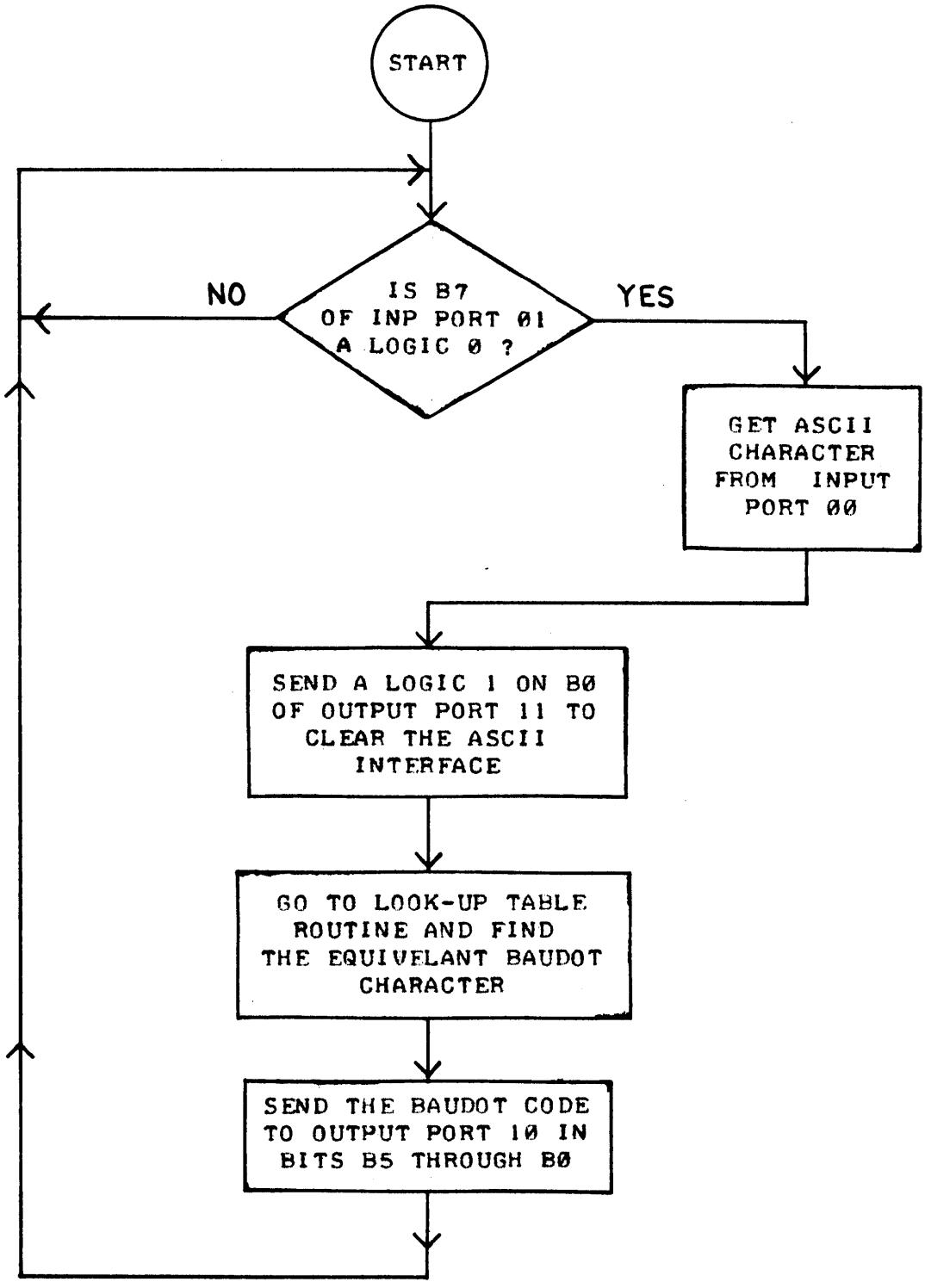
LINES WITH ARROWS MAY BE USED TO INTER-CONNECT THE THREE TYPES OF SYMBOLS JUST PRESENTED. IN THIS WAY, THE SYMBOLS MAY BE CONNECTED TO FORM READILY UNDERSTOOD "FLOW CHARTS" OF OPERATIONS THAT ARE TO OCCUR IN A PROGRAM AND TO SHOW HOW VARIOUS OPERATIONS RELATE TO EACH OTHER. FLOW CHARTS ARE EXTREMELY VALUABLE REFERENCES WHEN DEVELOPING PROGRAMS AS WELL AS WHEN WANTS TO UPDATE OR EXPAND A PROGRAM AND NEEDS TO QUICKLY REVIEW THE OPERATION OF A PARTICULAR PROGRAM.

BELOW IS AN EXAMPLE OF A FLOW CHART FOR A RELATIVELY SIMPLE PROGRAM THAT IS TO ACCEPT CHARACTERS FROM AN ASCII TELETYPE MACHINE AND SEND OUT THE EQUIVELANT CHARACTER TO A BAUDOT TELETYPE UNIT. IN THIS ILLUSTRATION IT IS ASSUMED THAT THE I/O INTERFACES TO THE TELETYPE MACHINES ARE "PARALLEL" INTERFACES (VERSUS BIT-SERIAL) SO THAT COMPLEX TIMING OPERATIONS DO NOT HAVE TO BE DISCUSSED IN THE EXAMPLE. A WRITTEN DESCRIPTION OF THE EXAMPLE PROGRAM COULD BE STATED AS FOLLOWS:

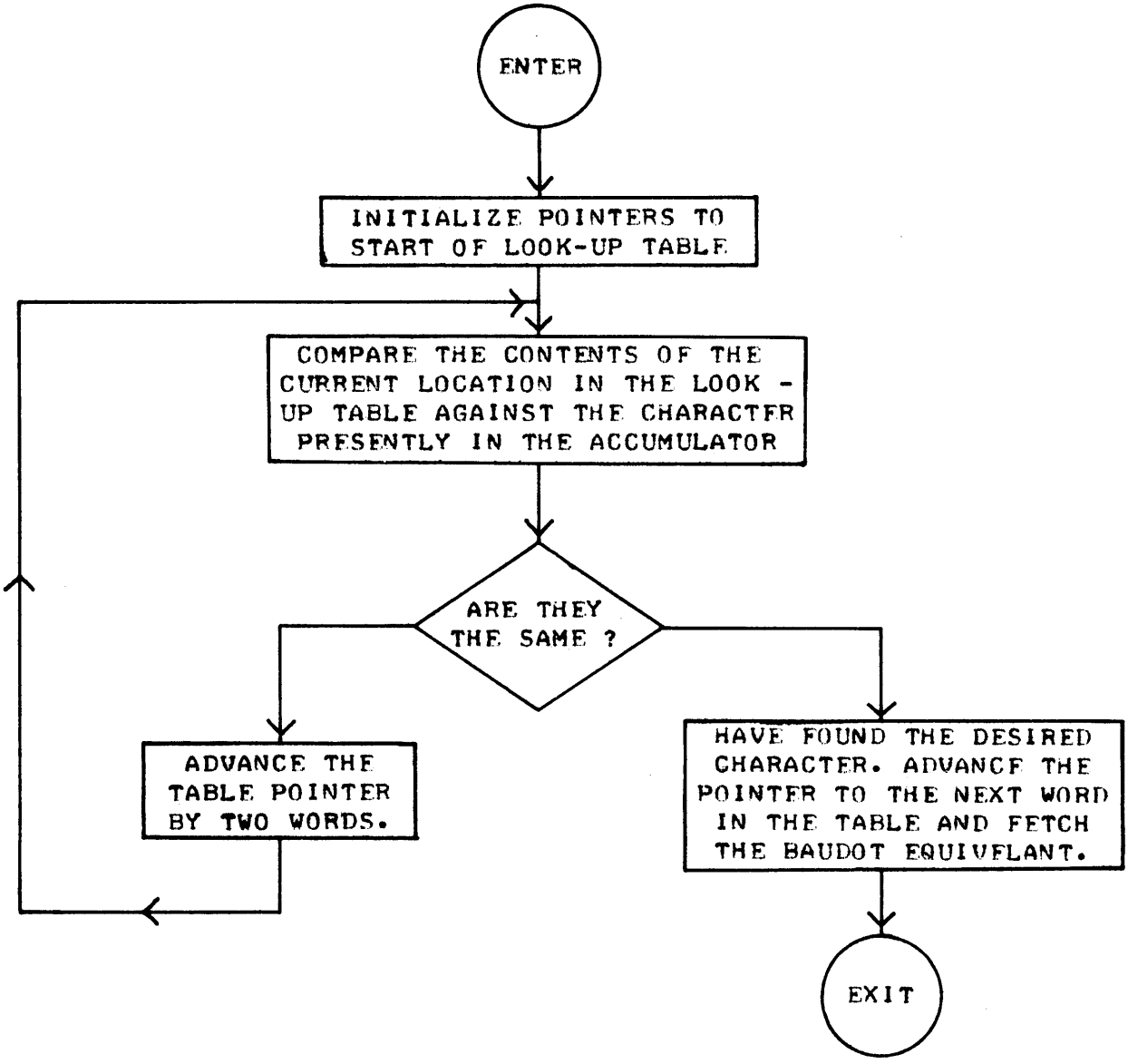
THE 8008 UNIT IS TO MONITOR BIT "B7" OF INPUT PORT 01, WHICH IS THE "CONTROL" PORT FOR AN INTERFACE TO AN ASCII TELETYPE MACHINE. WHENEVER BIT "B7" ON INPUT PORT 01 GOES LOW (LOGIC 0) IT INDICATES A NEW CHARACTER IS WAITING IN PARALLEL FORMAT FROM THE TELETYPE AT INPUT PORT 00. THE COMPUTER IS TO IMMEDIATELY OBTAIN THE CHARACTER THAT IS WAITING AT INPUT PORT 00 AND AS SOON AS IT HAS OBTAINED THE DATA IT IS TO SEND A LOGIC 1 (HIGH) SIGNAL TO BIT "B0" OF OUTPUT PORT 11 TO SIGNAL THE ASCII INTERFACE THAT THE CHARACTER HAS BEEN ACCEPTED BY THE COMPUTER. (THE RECEIPT OF THIS SIGNAL BY THE ASCII INTERFACE WILL THEN CAUSE THE ASCII INTERFACE TO RESTORE THE CONTROL SIGNAL ON BIT "B7" OF INPUT PORT 01 TO A HIGH (LOGIC 1) CONDITION).

WHENEVER A CHARACTER HAS BEEN RECEIVED FROM THE ASCII TELETYPE ON INPUT PORT 00, THE COMPUTER IS TO COMPARE THE CHARACTER JUST RECEIVED AGAINST AN ASCII TO BAUDOT "LOOK-UP" TABLE WHICH IS STORED IN THE COMPUTER'S MEMORY UNTIL IT FINDS A MATCH. WHEN IT FINDS A MATCH IT WILL THEN OBTAIN THE EQUIVELANT BAUDOT CHARACTER FROM THE CONVERSION TABLE AND SEND THE BAUDOT CODE FOR THE CHARACTER IN BIT POSITIONS "B5" THROUGH "B0" OF OUTPUT PORT 10. BIT "B5" WILL SERVE TO INDICATE TO THE BAUDOT

INTERFACE WHETHER THE CODE IN BITS "B4" THROUGH "B0" IS TO BE PROCESSED BY THE TELETYPE WHEN IT IS IN THE "LETTERS" OR "FIGURES" MODE. IT IS ASSUMED THAT THE CHARACTER RATE (BUT NOT NECESSARILY THE BAUD RATE) IS THE SAME FOR BOTH MACHINES SO THAT THE EXAMPLE MAY BE SIMPLIFIED BY ELIMINATING THE REQUIREMENT FOR CHARACTER BUFFERING OR STACKING" IN THE MEMORY OF THE COMPUTER. HOWEVER, IN PRACTICAL APPLICATIONS SUCH CAPABILITY MIGHT BE REQUIRED AND THE FEATURE COULD BE ADDED TO THE PROGRAM. BUT, FOR THIS CASE, AS SOON AS THE BAUDOT CODE HAS BEEN TRANSMITTED (IN PARALLEL FORMAT) TO THE BAUDOT INTERFACE, THE COMPUTER WILL SIMPLY GO BACK TO WAITING FOR THE NEXT CHARACTER TO COME IN FROM THE ASCII MACHINE. THE WRITTEN DESCRIPTION PROVIDED HERE COULD BE REPRESENTED QUITE CLEARLY BY THE FLOW CHART SHOWN BELOW.



THE FLOW CHART OF THE PROGRAM AS SHOWN ON THE PREVIOUS PAGE COULD BE CONSIDERED AS AN "OUTLINE" OF THE PROGRAM. PORTIONS OF THAT FLOW CHART COULD BE EXPANDED INTO MORE DETAILED FLOW CHARTS TO PRESENT A DETAILED VIEW OF SPECIAL OPERATIONS. FOR INSTANCE THE RECTANGLE LABELED "GO TO LOOK-UP TABLE ROUTINE AND FIND THE EQUIVALENT BAUDOT CHARACTER" REALLY REFERS TO A PORTION OF THE PROGRAM THAT CONSISTS OF A NUMBER OF OPERATIONS. THESE OPERATIONS COULD BE DESCRIBED IN A SEPARATE FLOW CHART AS ILLUSTRATED BELOW.



THE READER CAN SEE THAT THE ABOVE FLOW CHART READILY ILLUSTRATES THE OPERATION OF THE "TABLE LOOK-UP ROUTINE." WITH A LITTLE STUDY ONE COULD DISCERN THAT THE LOOK-UP TABLE CONSIST OF AN AREA IN MEMORY THAT HAS AN ASCII CHARACTER CODE IN ONE WORD, FOLLOWED IN THE NEXT WORD BY THE SAME CHARACTER IN THE BAUDOT CODE. THIS SEQUENCE CONTINUES FOR ALL THE POSSIBLE CHARACTERS AS SHOWN ON THE TOP OF THE NEXT PAGE. THE FLOW CHART ILLUSTRATES HOW THE DATA IN THE LOOK-UP TABLE IS SCANNED BY SKIPPING OVER EVERY OTHER MEMORY LOCATION (WHICH CONTAINS THE BAUDOT CODES) UNTIL THE PROPER ASCII CHARACTER IS LOCATED. WHEN THAT IS LOCATED, THE ROUTINE SIMPLY EXTRACTS THE PROPER BAUDOT CODE FROM THE NEXT MEMORY LOCATION IN THE TABLE. THE FLOW CHART MAKES THE SEQUENCE EASIER TO FOLLOW AND UNDERSTAND THAN A PURELY VERBAL EXPLANATION OF THE ROUTINE.

ADDRESS			MEMORY CONTENTS
PAGE: XX	LOC: Z		ASCII CODE FOR LETTER "A"
PAGE: XX	LOC: Z+1		BAUDOT CODE FOR LETTER "A"
PAGE: XX	LOC: Z+2		ASCII CODE FOR LETTER "B"
.	.		.
.	.		.
.	.		.
.	.		.
PAGE: XX	LOC: Z+3		BAUDOT CODE FOR LETTER "B"
PAGE: XX	LOC: Z+2(N-1)		ASCII CODE FOR "N"TH CHARACTER
PAGE: XX	LOC: Z+2(N-1)+1		BAUDOT CODE FOR "N"TH CHARACTER

ILLUSTRATION OF LOOK-UP TABLE ORGANIZATION FOR EXAMPLE PROGRAM

IT IS STRONGLY RECOMMENDED THAT BEGINNING PROGRAMMERS DEVELOP THE HABIT OF FIRST WRITING DOWN THE FUNCTION(S) OF THE DESIRED PROGRAM AND THEN DRAWING UP FLOW CHARTS AS DETAILED AS THE INDIVIDUAL FEELS IS NECESSARY TO CLEARLY SHOW THE INTENDED OPERATIONS OF THE PROGRAM THAT IS TO BE DEVELOPED. A NOVICE PROGRAMMER WILL BE WISE TO PREPARE QUITE DETAILED FLOW CHARTS. MORE EXPERIENCED PROGRAMMERS MAY PREFER TO LEAVE OUT DETAILS OF OPERATIONS THAT THEY THOROUGHLY UNDERSTAND. THE FLOW CHARTS SHOULD SERVE AS READY REFERENCES WHEN THE PROGRAMMER GOES ON TO ACTUALLY DEVELOP THE STEP-BY-STEP MACHINE LANGUAGE INSTRUCTION SEQUENCES FOR THE COMPUTER.

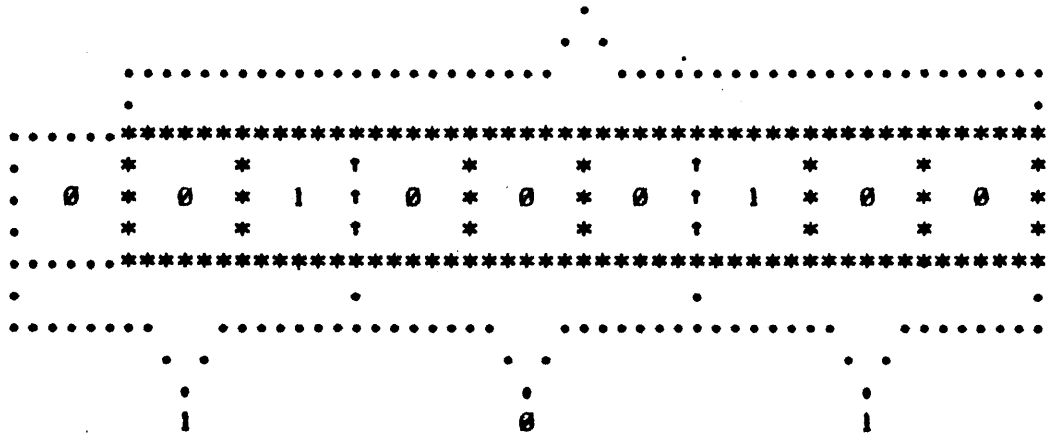
FLOW CHARTS ARE ALSO AN EXCELLENT METHOD FOR COMMUNICATING PROGRAMMING CONCEPTS TO FELLOW COMPUTER PROGRAMMERS. IT IS THE COMMON LANGUAGE OF COMPUTER TECHNOLOGISTS. (REMEMBER - GENERAL FLOW CHARTS DO NOT HAVE TO BE MACHINE SPECIFIC!) LEARNING HOW TO PREPARE AND READ FLOW CHARTS IS AN IMPORTANT (YET EASY) SKILL FOR ALL COMPUTER PROGRAMMERS TO ACQUIRE. IT CAN ALSO BE FUN AND A CREATIVE PROCESS AS ONE CAN VIEW THE OVER-ALL OPERATION OF A PROGRAM UNDER DEVELOPMENT AND GAIN NEW INSIGHTS INTO WHERE TO INTER-CONNECT ROUTINES, USE COMMON "LOOPS," TO SAVE MEMORY SPACE, OR OTHERWISE DETECT WAYS TO ENHANCE THE PROGRAM'S CAPABILITY.

BEFORE ONE CAN EFFECTIVELY DEVELOP MACHINE LANGUAGE PROGRAMS FOR THE 8008 CPU ONE MUST BE THOROUGHLY FAMILIAR WITH THE INSTRUCTION SET FOR THE MACHINE. IT IS ASSUMED FOR THE REMAINDER OF THIS MANUAL THAT THE READER HAS STUDIED THE DETAILED INFORMATION FOR THE INSTRUCTION SET OF THE 8008 CPU WHICH WAS PROVIDED IN THE FIRST CHAPTER. THE PROGRAMMER SHOULD BECOME INTIMATELY FAMILIAR WITH THE MNEMONICS (PRONOUNCED "KNEE-MONICS") FOR EACH TYPE OF INSTRUCTION. (MNEMONICS ARE EASILY REMEMBERED SYMBOLIC REPRESENTATIONS OF MACHINE LANGUAGE INSTRUCTIONS. THEY ARE FAR EASIER TO WORK WITH THAN THE ACTUAL NUMERIC CODES USED BY THE COMPUTER WHEN THE PROGRAMMER IS DEVELOPING A PROGRAM. THE MNEMONICS USED TO RE-PRESENT THE INSTRUCTIONS AVAILABLE IN AN 8008 SYSTEM WERE PRESENTED IN THE PREVIOUSLY MENTIONED FIRST CHAPTER OF THIS 8008 PROGRAMMERS MANUAL. WHILE THE PROGRAMMER WILL DEVELOP PROGRAMS AND "THINK" IN TERMS OF THE MNEMONICS, THE PROGRAMMER MUST EVENTUALLY CONVERT THE MNEMONICS TO THE MACHINE CODES USED BY THE COMPUTER. THIS, HOWEVER, IS ALMOST PURELY A "LOOK-UP" PROCEDURE, AND IN FACT, AS WILL BE SEEN SHORTLY, THIS TASK CAN ITSELF BE PERFORMED BY THE COMPUTER THROUGH THE USE OF AN "ASSEMBLER" PROGRAM.

MACHINE LANGUAGE PROGRAMMERS SHOULD ALSO BE FAMILIAR WITH MANIPU-LATING NUMBERS IN BINARY AND OCTAL FORM. IT IS ASSUMED THAT READERS ARE FAMILIAR WITH REPRESENTING NUMBERS AS BINARY VALUES, HOWEVER, THERE MAY BE A FEW READERS WHO ARE NOT USED TO THE CONVENTION OF REPRESENTING BIN-ARY NUMBERS BY THEIR OCTAL EQUIVALENT. THE TECHNIQUE IS QUITE SIMPLE AS IT CONSIST MERELY OF GROUPING BINARY DIGITS INTO GROUPS OF THREE AND REPRESENTING THEIR VALUE AS AN OCTAL NUMBER. THE OCTAL NUMBERING SYSTEM ONLY USES THE DIGITS 0 THROUGH 7, WHICH IS EXACTLY THE RANGE THAT A GROUP OF THREE BINARY DIGITS CAN REPRESENT. THE OCTAL NUMBERING SYSTEM MAKES IT A LOT EASIER TO MANIPULATE BINARY NUMBERS - FOR INSTANCE MOST PEOPLE FIND IT CONSIDERABLY MORE CONVENIENT TO REMEMBER AS THREE DIGIT NUMBER SUCH AS 1 0 4 THAN THE BINARY EQUIVALENT 0 1 0 0 0 1 0 0. AN OCTAL NUMBER IS EASILY EXPANDED TO A BINARY NUMBER BY SIMPLY PLACING THE OCTAL VALUE IN BINARY FORM USING THREE BINARY DIGITS.

THE INFORMATION IN AN EIGHT BIT BINARY REGISTER CAN BE EASILY CON-VERTED TO AN OCTAL NUMBER BY GROUPING THE BITS INTO GROUPS OF THREE STARTING WITH THE LEAST SIGNIFICANT BITS. THE TWO MOST SIGNIFICANT BITS IN THE REGISTER WHICH FORM THE LAST GROUP WILL ONLY BE ABLE TO REPRESENT THE OCTAL NUMBERS 0 TO 3. THE DIAGRAM BELOW ILLUSTRATES THE CONVENTION.

EIGHT CELL REGISTER



CONVERTING AN 8 BIT REGISTER FROM BINARY TO OCTAL NUMBERS

NOTE IN THE DIAGRAM HOW AN IMAGINARY ADDITIONAL BINARY DIGIT WITH A VALUE OF ZERO WAS ASSIGNED TO THE LEFT OF THE MOST SIGNIFICANT BIT SO THAT THE OCTAL CONVENTION FOR THE TWO MOST SIGNIFICANT BITS COULD BE MAINTAINED.

A TABLE ILLUSTRATING THE RELATIONSHIP BETWEEN THE BINARY AND OCTAL SYSTEMS IS PROVIDED FOR REFERENCE BELOW.

BINARY PATTERN	REPRESENTATIVE OCTAL #
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

A PERSON WHO DESIRES TO DEVELOP MACHINE LANGUAGE PROGRAMS FOR THE 8008 CPU SHOULD ALSO BECOME FAMILIAR WITH SOME STANDARD CONVENTIONS USED WHEN DEALING WITH "CLOSED" REGISTERS (GROUPS OF BINARY CELLS OF FIXED LENGTH SUCH AS A MEMORY WORD OR CPU REGISTER). ONE VERY SIMPLE POINT TO REMEMBER, AS A STARTER, IS THAT WHEN A GROUP OF CELLS IN A REGISTER IS IN THE ALL ONES CONDITION:

1 1 1 1 1 1 1 1

AND THE COUNT OF 1 IS ADDED TO THE REGISTER, THE REGISTER GOES TO THE VALUE:

0 0 0 0 0 0 0 0

OR, IF THE COUNT OF: 1 0 (BINARY) WAS ADDED TO A REGISTER THAT CONTAINED ALL ONES, THE NEW VALUE IN THE REGISTER WOULD BE AS SHOWN:

```
  1 1   1 1 1   1 1 1
+ 0 0   0 0 0   0 1 0
-------------------
  0 0   0 0 0   0 0 1
```

SIMILARLY, GOING THE OPPOSITE WAY, IF ONE SUBTRACTS A NUMBER SUCH AS 1 0 0 (BINARY) FROM A REGISTER THAT CONTAINS SOME LESSER VALUE, SUCH AS 0 1 0 (BINARY) THE THE REGISTER WOULD CONTAIN THE RESULT SHOWN IN THE FOLLOWING ILLUSTRATION:

```
  0 0   0 0 0   0 1 0
- 0 0   0 0 0   1 0 0
-------------------
  1 1   1 1 1   1 1 0
```

IT SHOULD BE NOTED THAT IF ONE USES ALL THE BITS IN A FIXED LENGTH REGISTER ONE CAN REPRESENT MATHEMATICAL VALUES OF AN ABSOLUTE MAGNITUDE FROM ZERO TO THE QUANTITY TWO TO THE NTH POWER MINUS ONE FROM THE QUANTITY (0 TO (2↑N - 1)) WHERE "N" IS THE NUMBER OF BITS IN THE REGISTER. HOWEVER, IF ALL THE BITS IN A REGISTER ARE USED TO REPRESENT THE MAGNITUDE OF A NUMBER, AND IT IS ALSO DESIRED TO REPRESENT THE MAGNITUDE AS BEING EITHER POSITIVE OR NEGATIVE IN SIGN, THEN SOME ADDITIONAL MEANS

MUST BE AVAILABLE TO RECORD THE SIGN OF THE MAGNITUDE. GENERALLY, THIS WOULD REQUIRE USING ANOTHER REGISTER OR MEMORY LOCATION SOLELY FOR THE PURPOSE OF KEEPING TRACK OF THE SIGN OF A NUMBER.

HOWEVER, IN MANY APPLICATIONS IT IS DESIRABLE TO ESTABLISH A CONVENTION THAT WILL ALLOW ONE TO MANIPULATE POSITIVE AND NEGATIVE NUMBERS WITHOUT HAVING TO USE ADDITIONAL REGISTER(S) TO MAINTAIN THE SIGN OF A NUMBER OR OTHERWISE PLACE RESTRICTIONS ON OPERATIONS. ONE WAY THIS MAY BE DONE IS TO SIMPLY ASSIGN THE MOST SIGNIFICANT BIT IN A REGISTER (OR THE MOST SIGNFICANT BIT IN A GROUP OF BITS) TO BE A "SIGN" INDICATOR. THE REMAINING BITS REPRESENT THE MAGNITUDE OF THE NUMBER REGARDLESS OF WHETHER IT IS POSITIVE OR NEGATIVE. NATURALLY, WHEN THIS IS DONE, THEN THE MAGNITUDE RANGE FOR AN "N" CELL REGISTER BECOMES 0 TO (2↑(N-1)-1) RATHER THAN 0 TO (2↑N) - 1. THE CONVENTION NORMALLY USED IS THAT IF THE MSB (MOST SIGNIFICANT BIT) IN THE REGISTER IS A ONE (1), THEN THE NUMBER REPRESENTED BY THE REMAINING BITS IS "NEGATIVE" IN SIGN. IF THE MSB IS ZERO (0) THEN THE REMAINING BITS SPECIFY THE MAGNITUDE OF A POSITIVE NUMBER. THIS CONVENTION ALLOWS COMPUTER PROGRAMMERS TO MANIPULATE MATHEMATICAL QUANTITIES IN A FASHION THAT MAKES IT EASY FOR THE COMPUTER TO KEEP TRACK OF THE SIGN OF A NUMBER. SOME EXAMPLES OF BINARY NUMBERS IN AN EIGHT BIT REGISTER SUCH AS THOSE USED IN AN 8008 SYSTEM ARE SHOWN BELOW.

BINARY REPRESENTATION	OCTAL	DECIMAL
0 0 0 0 0 1 0 0 0	0 1 0	+ 8
1 0 0 0 1 0 0 0	2 1 0	- 8
0 1 1 1 1 1 1 1	1 7 7	+127
1 1 1 1 1 1 1 1	3 7 7	-127
0 0 0 0 0 0 0 1	0 0 1	+ 1
1 0 0 0 0 0 0 1	2 0 1	- 1

WHILE THE SIGNED BIT CONVENTION ALLOWS THE SIGN OF A NUMBER TO BE STORED IN THE SAME REGISTER (OR WORD) AS THE MAGNITUDE, SIMPLY USING THE "SIGNED BIT" CONVENTION ALONE CAN STILL BE A SOMEWHAT CLUMSY METHOD TO USE IN A COMPUTER BECAUSE OF THE METHOD IN WHICH A COMPUTER MATHEMATICALLY ADDS THE CONTENTS OF TWO BINARY REGISTERS IN THE ACCUMULATOR. SUPPOSE FOR EXAMPLE THAT THE COMPUTER WAS TO ADD TOGETHER A POSITIVE AND A NEGATIVE NUMBER THAT WERE STORED IN REGISTERS IN THE SIGNED BIT FORMAT JUST DESCRIBED.

	0 0 0 0 1 0 0 0	(+ 8 DECIMAL)
PLUS	1 0 0 0 1 0 0 0	(- 8 DECIMAL)
EQUAL	1 0 0 1 0 0 0 0	(THIS IS NOT 0!)

THE RESULT OF THE OPERATION AS ILLUSTRATED WOULD NOT BE WHAT THE PROGRAMMER INTENDED! IN ORDER FOR THE OPERATION TO BE PERFORMED CORRECTLY IT IS NECESSARY TO ESTABLISH A METHOD OF PROCESSING THE NEGATIVE NUMBER CALLED THE "TWO'S COMPLEMENT" CONVENTION. IN THE "TWO'S COMPLEMENT" CONVENTION A NEGATIVE NUMBER IS REPRESENTED BY COMPLEMENTING WHAT THE VALUE FOR A POSITIVE NUMBER WOULD BE (COMPLEMENTING IS THE PROCESS OF REPLACING ALL BITS THAT ARE "0" WITH A "1" AND THOSE THAT ARE "1" WITH A "0") AND THEN ADDING THE VALUE ONE (1) TO THE COMPLEMENTED VALUE.

AS AN EXAMPLE, THE NUMBER MINUS EIGHT (- 8) DECIMAL WOULD BE DERIVED FROM THE NUMBER PLUS EIGHT (+ 8) BY THE FOLLOWING OPERATIONS.

```
 0 0   0 0 1   0 0 0   (ORIGINAL + 8)

 1 1   1 1 0   1 1 1   (COMPLEMENTED)
 0 0   0 0 0   0 0 1   (NOW ADD + 1)
-------------------
 1 1   1 1 1   0 0 0   (2'S COMPLEMENT FORM OF - 8)
```

SOME EXAMPLES OF NUMBERS EXPRESSED IN TWO'S COMPLEMENT NOTATION WITH THE "SIGNED BIT" CONVENTION RETAINED ARE SHOWN BELOW.

BINARY REPRESENTATION			OCTAL	DECIMAL
0 0	0 0 1	0 0 0	0 1 0	+ 8
1 1	1 1 1	0 0 0	3 7 0	- 8
0 1	1 1 1	1 1 1	1 7 7	+127
1 0	0 0 0	0 0 1	2 0 1	-127
0 0	0 0 0	0 0 1	0 0 1	+ 1
1 1	1 1 1	1 1 1	3 7 7	- 1
0 0	0 0 0	0 0 0	0 0 0	+ 0
1 0	0 0 0	0 0 0	2 0 0	-128

NOTE THAT WHEN USING THE TWO'S COMPLEMENT METHOD ONE MAY STILL RETAIN THE CONVENTION OF HAVING THE MSB IN THE REGISTER ESTABLISH THE "SIGN" NOTATION. IF THE MSB = 1, AS IN THE ABOVE ILLUSTRATION, THE NUMBER IS ASSUMED TO BE NEGATIVE. HOWEVER, SINCE THE NUMBER IS IN THE TWO'S COMPLEMENT FORM THE COMPUTER CAN READILY ADD A "POSITIVE" AND A "NEGATIVE" NUMBER AND COME UP WITH A RESULT THAT IS READILY INTERPRETED. LOOK!

```
       0 0   0 0 1   0 0 0   (+ 8 DECIMAL)
ADD    1 1   1 1 1   0 0 0   (- 8 DECIMAL AS 2'S COMPLEMENT)
      -------------------
       0 0   0 0 0   0 0 0   (CORRECT ANSWER OF ZERO!)
```

ANOTHER ESTABLISHED CONVENTION IN HANDLING NUMBERS WITH A COMPUTER IS TO ASSUME THAT "0" IS A "POSITIVE" VALUE. BECAUSE OF THIS CONVENTION THE MAGNITUDE OF THE LARGEST NEGATIVE NUMBER THAT CAN BE REPRESENTED IN A FIXED LENGTH REGISTER IS ONE MORE THAN THAT POSSIBLE FOR A POSITIVE NUMBER.

THE VARIOUS MEANS OF STORING AND MANIPULATING THE SIGNS OF NUMBERS AS JUST DISCUSSED HAVE ADVANTAGES AND DRAWBACKS AND THE METHOD USED DEPENDS ON THE SPECIFIC APPLICATION. HOWEVER, FOR MOST USER'S THE TWO'S COMPLEMENT METHOD COUPLED WITH THE "SIGNED BIT" CONVENTION WILL BE THE MOST CONVENIENT AND MOST OFTEN USED METHOD. THE PROSPECTIVE MACHINE LANGUAGE PROGRAMMER SHOULD MAKE SURE THAT THE CONVENTION IS WELL UNDERSTOOD.

ANOTHER AREA THAT THE MACHINE LANGUAGE PROGRAMMER MUST HAVE A THOR-

OUGH KNOWLEDGE OF IS THE CONVERSION OF NUMBERS BETWEEN THE DECIMAL NUMBERING SYSTEM THAT MOST PEOPLE WORK WITH ON A DAILY BASIS AND THE BINARY AND OCTAL NUMBERING SYSTEM UTILIZED BY COMPUTER TECHNOLOGISTS. PROGRAMMERS WORKING WITH THE 8008 CPU WILL GENERALLY FIND THE OCTAL NUMBERING SYSTEM MOST CONVENIENT BECAUSE THE CONVERSION FROM OCTAL TO BINARY IS SIMPLY A MATTER OF GROUPING BINARY BITS INTO GROUPS OF THREE AS DISCUSSED AT THE START OF THIS CHAPTER ON FUNDAMENTAL PROGRAMMING SKILLS. IT IS EASIER TO REMEMBER OCTAL CODES THAN LONG STRINGS OF BINARY DIGITS BUT ONE CAN READILY EXPAND THE OCTAL CODES INTO BINARY DIGIT STRINGS. OF COURSE, MOST PEOPLE ARE USED TO THINKING IN DECIMAL TERMS, WHICH THE COMPUTER DOES NOT USE AT THE MACHINE LANGUAGE LEVEL, AND SO IT IS NECESSARY TO BE ABLE TO CONVERT BACK AND FORTH BETWEEN THE VARIOUS NUMBERING SYSTEMS AS PROGRAMS ARE DEVELOPED.

THE CONVERSION PROCESS THAT IS GENERALLY MORE TROUBLESOME FOR PEOPLE TO LEARN (THAN THE OCTAL TO BINARY TRANSLATION) IS FROM DECIMAL TO BINARY OR DECIMAL TO OCTAL (AND VICE-VERSA)! IT IS PROBABLY A BIT EASIER FOR MOST PEOPLE TO LEARN TO CONVERT FROM DECIMAL TO OCTAL AND THEN USE THE SIMPLE OCTAL TO BINARY EXPANSION TECHNIQUE, THAN TO CONVERT DIRECTLY FROM DECIMAL TO BINARY AND SO THE EASIER METHOD WILL BE PRESENTED HERE. SINCE IT IS ASSUMED THAT THE READER IS ALREADY FAMILIAR WITH GOING FROM OCTAL TO BINARY (AND VICE-VERSA) ONLY THE CONVERSIONS BETWEEN DECIMAL AND OCTAL (AND THE REVERSE) WILL BE PRESENTED IN THESE PAGES.

A DECIMAL NUMBER MAY BE READILY CONVERTED TO ITS OCTAL EQUIVALENT BY THE FOLLOWING METHOD:

DIVIDE THE DECIMAL NUMBER BY 8. RECORD THE REMAINDER (NOTE THAT IS THE R E M A I N D E R !!) AS THE LEAST SIGNIFICANT DIGIT OF THE OCTAL NUMBER BEING DERIVED. TAKE THE QUOTIENT JUST OBTAINED AND USE IT AS THE NEW DIVIDEND. DIVIDE THE NEW DIVIDEND BY 8. THE REMAINDER FROM THIS OPERATION BECOMES THE NEXT SIGNIFICANT DIGIT OF THE OCTAL NUMBER. THE QUOTIENT IS AGAIN USED AS THE NEW DIVIDEND. THE PROCESS IS CONTINUED UNTIL THE QUOTIENT BECOMES 0. THE NUMBER OBTAINED FROM PLACING ALL THE REMAINDERS (FROM EACH DIVISION) IN INCREASING SIGNIFICANT ORDER (FIRST REMAINDER AS THE LEAST SIGNIFICANT DIGIT, LAST REMAINDER AS THE MOST SIGNIFICANT DIGIT) IS THE OCTAL NUMBER EQUIVALENT OF THE ORIGINAL DECIMAL NUMBER. THE PROCESS IS ILLUSTRATED BELOW FOR CLARITY.

THE OCTAL EQUIVALENT OF 1234 DECIMAL IS:

			QUOTIENT	REMAINDER
ORIGINAL NUMBER	1234 / 8	=	154	2
LAST QUOTIENT BECOMES NEW DIVIDEND	154 / 8	=	19	2
LAST QUOTIENT BECOMES NEW DIVIDEND	19 / 8	=	2	3
LAST QUOTIENT BECOMES NEW DIVIDEND	2 / 8	=	-	2

THUS THE OCTAL EQUIVALENT OF 1234 IS: 2 3 2 2

THE ABOVE METHOD IS QUITE EASY AND STRAIGHT FORWARD. SINCE A MAJ-

ORITY OF THE TIME THE USER WILL BE INTERESTED IN CONVERSIONS OF DECIMAL NUMBERS LESS THAN 255 (THE MAXIMUM DECIMAL NUMBER THAT CAN BE EXPRESSED IN AN EIGHT BIT REGISTER) ONLY A FEW DIVISIONS ARE NECESSARY:

THE OCTAL EQUIVALENT OF 255 DECIMAL IS:

				QUOTIENT	REMAINDER
ORIGINAL NUMBER	255 / 8	=	31		7
LAST QUOTIENT BECOMES NEW DIVIDEND	31 / 8	=	3		7
LAST QUOTIENT BECOMES NEW DIVIDENT	3 / 8	=	-		3

THUS THE OCTAL EQUIVALENT OF 255 IS: 3 7 7

FOR NUMBERS LESS THAN 63 DECIMAL (AND SUCH NUMBERS ARE USED FREQUENTLY TO SET COUNTERS IN "LOOP" ROUTINES) THE ABOVE METHOD REDUCES TO ONE DIVISION WITH THE REMAINDER BEING THE LSD AND THE QUOTIENT THE MSD. THIS IS A FEAT MOST PROGRAMMERS HAVE LITTLE DIFFICULTY DOING IN THEIR HEAD!

THE OCTAL EQUIVALENT OF 63 DECIMAL IS:

				QUOTIENT	REMAINDER
ORIGINAL NUMBER	63 / 8	=	7		7
LAST QUOTIENT BECOMES NEW DIVIDEND	7 / 8	=	-		7

THUS THE OCTAL EQUIVALENT OF 63 IS: 7 7

GOING FROM OCTAL TO DECIMAL IS QUITE EASY TOO. THE PROCESS CONSIST OF SIMPLY MULTIPLYING EACH OCTAL DIGIT BY THE NUMBER 8 RAISED TO ITS POSITIONAL (WEIGHTED) POWER AND THEN ADDING UP THE TOTAL OF EACH PRODUCT FOR ALL THE OCTAL DIGITS:

2 3 2 2 OCTAL =

2	X (8↑0)	= (2 X 1)	=	2
2	X (8↑1)	= (2 X 8)	=	16
3	X (8↑2)	= (3 X 64)	=	192
2	X (8↑3)	= (2 X 512)	=	1024

THUS THE DECIMAL EQUIVALENT OF 2322 OCTAL IS: 1 2 3 4

BESIDES THE BASIC MATHEMATICAL SKILLS INVOLVED WITH USING OCTAL AND BINARY NUMBERS, THERE ARE SOME PRACTICAL "BOOK KEEPING" CONSIDERATIONS THAT MACHINE LANGUAGE PROGRAMMERS MUST LEARN TO DEAL WITH AS THEY DEVELOP PROGRAMS. THESE "BOOK KEEPING" MATTERS HAVE TO DO WITH MEMORY USAGE AND ALLOCATION.

AS THE USER WHO HAS READ **CHAPTER ONE IN THIS MANUAL NOW KNOWS,** EACH TYPE OF INSTRUCTION USED IN THE 8008 CPU REQUIRES ONE, TWO OR THREE WORDS OF MEMORY. AS A GENERAL RULE SIMPLE REGISTER TO REGISTER OR REGISTER TO MEMORY COMMANDS REQUIRE BUT ONE MEMORY WORD. "IMMEDIATE" TYPE COMMANDS REQUIRE TWO MEMORY LOCATIONS (THE INSTRUCTION CODE FOLLOWED IMMEDIATELY BY THE "DATA" OR OPERAND). JUMP OR CALL INSTRUCTIONS REQUIRE THREE WORDS OF MEMORY STORAGE. ONE WORD FOR THE INSTRUCTION CODE AND TWO MORE WORDS FOR THE ADDRESS OF THE LOCATION SPECIFIED BY THE INSTRUCTION. THE FACT THAT DIFFERENT TYPES OF INSTRUCTIONS REQUIRE DIFFERENT AMOUNTS OF MEMORY IS IMPORTANT TO THE PROGRAMMER.

AS PROGRAMMERS WRITE A PROGRAM IT IS OFTEN NECESSARY FOR THEM TO KEEP TABS ON HOW MANY WORDS OF MEMORY THE ACTUAL OPERATING PORTION OF THE PROGRAM WILL REQUIRE (IN ADDITION TO CONTROLLING THE AREAS IN MEMORY THAT WILL BE USED FOR DATA STORAGE.) ONE REASON FOR MAINTAINING A COUNT OF THE NUMBER OF MEMORY WORDS A PROGRAM REQUIRES IS SIMPLY TO ENSURE THAT THE PROGRAM WILL "FIT" INTO THE AVAILABLE MEMORY SPACE.

OFTEN A PROGRAM THAT IS A LITTLE TOO LONG TO BE STORED IN AN AVAILABLE AMOUNT OF MEMORY WHEN FIRST DEVELOPED CAN BE RE-WRITTEN AFTER SOME **THOUGHT TO FIT IN THE AVAILABLE SPACE. GENERALLY, THE TRADE-OFF BETWEEN** WRITING "COMPACT" PROGRAMS VERSUS NOT-SO-COMPACT ROUTINES IS SIMPLY THE PROGRAMMER'S DEVELOPMENT TIME. HASTILY CONSTRUCTED PROGRAMS TEND TO REQUIRE MORE MEMORY STORAGE AREA BECAUSE THE PROGRAMMER DOES NOT TAKE THE TIME TO CONSIDER MEMORY CONSERVING INSTRUCTION COMBINATIONS.

HOWEVER, EVEN IF ONE IS NOT CONCERNED ABOUT CONSERVING THE AMOUNT OF MEMORY USED BY A PARTICULAR PROGRAM, ONE STILL OFTEN NEEDS TO KNOW HOW MUCH SPACE A GROUP OF INSTRUCTIONS WILL CONSUME IN MEMORY SO THAT ONE CAN TELL WHERE ANOTHER PROGRAM MIGHT BE PLACED WITHOUT INTERFERING WITH THE FIRST PROGRAM.

FOR THESE REASONS, PROGRAMMERS OFTEN FIND IT ADVANTAGEOUS TO DEVELOP THE HABIT OF WRITING DOWN THE NUMBER OF MEMORY WORDS UTILIZED BY EACH INSTRUCTION AS THEY WRITE THE MNEMONIC SEQUENCES FOR A ROUTINE, AND TO ALSO MAINTAIN A COLUMN SHOWING THE TOTAL NUMBER OF WORDS REQUIRED FOR STORAGE OF THE ROUTINE. AN EXAMPLE OF A WORK SHEET WITH THIS PRACTICE BEING FOLLOWED IS ILLUSTRATED HERE:

MEMORY WORDS THIS INSTR.	TOTAL WORDS THIS ROUTINE	MNEMONICS	COMMENTS
2	2	LAI 000	/PLACE 000 IN ACCUMULATOR
2	4	LHI 001	/SET REGISTER "H" TO 1
2	6	LLI 150	/AND REGIS "L" TO 150
1	7	ADM	/ADD THE CONTENTS OF MEMORY
1	8	INL	/LOCATIONS 150 & 151 ON PAGE 1
1	9	ADM	/ADDING SECOND NUMBER TO FIRST
1	10	RET	/END OF SUBROUTINE

IN THE EXAMPLE THE TOTAL NUMBER OF WORDS USED COLUMN WAS KEPT USING

DECIMAL NUMBERS. MANY PROGRAMMERS PREFER TO MAINTAIN THIS COLUMN USING OCTAL NUMBERS BECAUSE OF THE DIRECT CORRELATION BETWEEN THE TOTAL NUMBER OF WORDS USED AND THE ACTUAL MEMORY ADDRESSES USED BY THE 8008.

THE EXAMPLE JUST PRESENTED CAN BE USED TO INTRODUCE ANOTHER CONSIDERATION DURING PROGRAM DEVELOPMENT - MEMORY ALLOCATION AND THE DISTINCTION BETWEEN PROGRAM STORAGE AREAS IN MEMORY AND AREAS USED TO HOLD DATA THAT IS OPERATED ON BY THE PROGRAM. NOTE THAT THE SAMPLE SUBROUTINE IS DESIGNED TO HAVE THE COMPUTER ADD THE CONTENTS OF MEMORY LOCATIONS 150 AND 151 ON PAGE 01. THUS, THOSE TWO LOCATIONS MUST BE RESERVED FOR DATA. ONE MUST ENSURE THAT THOSE SPECIFIC MEMORY LOCATIONS ARE NOT INADVERTANTLY USED FOR SOME OTHER PURPOSE. IN A TYPICAL PROGRAM ONE MAY HAVE MANY LOCATIONS IN MEMORY ASSIGNED FOR HOLDING OR MANIPULATING DATA. IT IS IMPORTANT THAT ONE MAINTAIN SOME SORT OF SYSTEM OF RECORDING WHERE ONE PLANS TO STORE BLOCKS OF DATA AND WHERE VARIOUS OPERATING

PG	LOC	RTN	NOTES
01	00Ø	ADD	ADD # 5 @ 15Ø & 151 (ØØØ-Ø11)
	01Ø		
	02Ø		
	03Ø		
	04Ø		
	05Ø		
	06Ø		
	07Ø		
	10Ø		
	11Ø		
	12Ø		
	13Ø		
	14Ø		
	15Ø	# STORAGE	(15Ø, 151)
	16Ø		
	17Ø		
	20Ø		
	21Ø		
	22Ø		
	23Ø		
	24Ø		
	25Ø		
	26Ø		
	27Ø		
	30Ø		
	31Ø		
	32Ø		
	33Ø		
	34Ø		
	35Ø		
	36Ø		
↓	37Ø		

MEMORY USAGE MAP

ROUTINES WILL RESIDE AS A PROGRAM IS DEVELOPED. THIS CAN BE READILY ACCOMPLISHED BY SETTING UP AND USING "MEMORY USAGE MAPS" (OFTEN COMMONLY REFERRED TO ALSO AS "CORE MAPS"). AN EXAMPLE OF A MEMORY USAGE MAP BEING STARTED FOR THE SUBROUTINE JUST DISCUSSED IS SHOWN ON THE PREVIOUS PAGE.

THE SAME TYPE OF FORM MAY ALSO BE USED AS A PROGRAMMING DEVELOPMENT SHEET AS SHOWN BELOW. WHEN THEY ARE USED FOR THIS PURPOSE, THE "RTN" COLUMN MAY BE USED FOR THE "LABELS" OR NAMES OF ROUTINES, AND THE MNEMONICS AND COMMENTS PLACED IN THE "NOTES" COLUMN. THE READER SHOULD NOTICE HOW SPACES ARE LEFT BETWEEN INSTRUCTIONS THAT OCCUPY MORE THAN ONE WORD IN MEMORY SO THAT THE ACTUAL ADDRESSES USED CAN BE DETERMINED AS THE ROUTINE IS DEVELOPED.

PG	LOC	RTN	NOTES
01	000	ADD,	LAI 000 /000 → ACC
↓	1		
	2		LHI 001 /H → 1
	3		
	4		LLI 150 /L → 150
	5		
	6		ADM /M → A
	7		INL /Adv. PNTR
	010		ADM /A + M = A'
	11		RET /END SUBRTN
	12		
	13		
	14		
	15		
	16		
	17		
	020		
	21		
	22		
	23		
	24		
	25		
	26		
	27		
	030		
	31		
	32		
	33		
	34		
	35		
	36		
↓	37		

PROGRAM DEVELOPMENT WORK SHEET

MEMORY USAGE MAPS ARE EXTREMELY VALUABLE FOR KEEPING LARGE PROGRAMS ORGANIZED AS THEY ARE DEVELOPED OR FOR DISPLAYING THE LOCATIONS OF A VARIETY OF PROGRAMS THAT ONE MIGHT DESIRE TO HAVE RESIDING IN MEMORY AT THE SAME TIME. THE SAME FORM IS ALSO USEFUL AS A PROGRAM DEVELOPMENT WORK SHEET. IT IS SUGGESTED THAT THE PERSON INTENDING TO DO EVEN A MODERATE AMOUNT OF MACHINE LANGUAGE PROGRAMMING MAKE UP A SUPPLY OF SUCH FORMS (USING A DITTO OR MIMEOGRAPH MACHINE) TO HAVE ON HAND.

THERE ARE SOME IMPORTANT FACTORS ABOUT MACHINE LANGUAGE PROGRAMMING THAT SHOULD BE POINTED OUT AS THEY HAVE CONSIDERABLE IMPACT ON THE TOTAL EFFICIENCY AND SPEED AT WHICH ONE CAN DEVELOP SUCH PROGRAMS AND GET THEM OPERATING CORRECTLY. THE FACTORS RELATE TO ONE SIMPLE FACT - PEOPLE DEVELOPING MACHINE LANGUAGE PROGRAMS (ESPECIALLY BEGINNERS) ARE VERY PRONE TO MAKING PROGRAMMING MISTAKES! REGARDLESS OF HOW CAREFULLY ONE PROCEEDS, IT ALWAYS SEEMS THAT ANY FAIR SIZED PROGRAM NEEDS TO BE "REVISED" BEFORE A PROPERLY OPERATING PROGRAM IS ACHIEVED. THE IMPACT THAT CHANGES IN A PROGRAM HAVE ON THE DEVELOPMENT (OR REDEVELOPMENT) EFFORT VARY ACCORDING TO WHERE IN THE PROGRAM SUCH CHANGES MUST BE MADE. THE REASON FOR THE SERIOUSNESS OF THE PROBLEM IS BECAUSE PROGRAM CHANGES GENERALLY RESULT IN THE ADDRESSES OF THE INSTRUCTIONS IN MEMORY BEING ALTERED. REMEMBER, IF AN INSTRUCTION IS ADDED, OR DELETED, THEN ALL THE REMAINING INSTRUCTIONS IN THE ROUTINE BEING ALTERED MUST BE MOVED TO DIFFERENT LOCATIONS! THIS CAN HAVE "MULTIPLYING" EFFECTS IF THE INSTRUCTIONS THAT ARE MOVED ARE REFERRED TO BY OTHER ROUTINES (SUCH AS CALL AND JUMP COMMANDS) BECAUSE THEN THE ADDRESSES REFERRED TO BY THOSE TYPES OF COMMANDS MUST BE ALTERED TOO! TO ILLUSTRATE THE SITUATION, A CHANGE WILL BE MADE TO THE SAMPLE PROGRAM PRESENTED SEVERAL PAGES AGO. SUPPOSE IT WAS DECIDED THAT THE SUBROUTINE SHOULD PLACE THE RESULT OF THE ADDITION CALCULATION IN A WORD IN MEMORY BEFORE EXITING THE SUBROUTINE INSTEAD OF SIMPLY HAVING THE RESULT IN THE ACCUMULATOR. THE ORIGINAL PROGRAM, FOR EXAMPLE, COULD HAVE BEEN RESIDING IN THE LOCATIONS SHOWN ON THE PROGRAM DEVELOPMENT WORK SHEET ON THE PREVIOUS PAGE. CHANGING THE PROGRAM WOULD RESULT IN IT OCCUPYING THE FOLLOWING MEMORY LOCATIONS:

	PAGE	LOC	MEMORY CONTENTS	MNEMONICS	COMMENTS
	01	000	006	LAI 000	/PLACE 000 IN ACCUMULATOR
	01	001	000		
	01	002	056	LHI 001	/SET REG "H" TO 1
	01	003	001		
	01	004	066	LLI 150	/SET REG "L" TO 150
	01	005	150		
	01	006	207	ADM	/ADD CONTENTS OF MEMORY
	01	007	060	INL	/LOCATIONS 150 & 151
	01	010	207	ADM	/ADD 2ND TO 1ST
	01	011	066	LLI 160	/SET REG "L" TO 160
**	01	012	160		
**	01	013	370	LMA	/SAVE ANSWER @ 160
**	01	014	007	RET	/END OF SUBROUTINE

THE ** LOCATIONS DENOTE THE ADDITIONAL MEMORY LOCATIONS REQUIRED BY THE MODIFIED SUBROUTINE. IF THE PROGRAMMER HAD ALREADY DEVELOPED A ROUTINE THAT RESIDED IN LOCATIONS 012, 013 OR 014, THE CHANGE WOULD REQUIRE THAT IT BE MOVED!

IF ONE WAS USING A PROGRAM DEVELOPMENT WORK SHEET, ONE WOULD HAVE HAD TO ERASE THE ORIGINAL "RET" INSTRUCTION AT THE END OF THE ROUTINE AND THEN WRITTEN IN THE TWO NEW COMMANDS AND ADDED THE "RET" INSTRUCTION

AT THE END. THE EFFECTS WOULD NOT BE TOO DEVESTATING SINCE THE CHANGE WAS INSERTED AT THE END OF THE SUBROUTINE - BUT SUPPOSE A SIMILAR CHANGE WAS NECESSARY AT THE START OF A SUBROUTINE THAT HAD 50 INSTRUCTIONS IN IT? THE PROGRAMMER WOULD HAVE TO DO A LOT OF ERASING!

THE EFFECTS OF CHANGES IN PROGRAM SOURCE LISTINGS WAS RECOGNIZED EARLY AS A PROBLEM IN DEVELOPING PROGRAMS AND SO PEOPLE DEVELOPED PROGRAMS CALLED "EDITORS" THAT WOULD ENABLE THE COMPUTER TO ASSIST PEOPLE IN THE TASK OF CREATING AND MANIPULATING SOURCE LISTINGS FOR PROGRAMS. AN "EDITOR" IS A PROGRAM THAT WILL ALLOW A PERSON TO USE THE COMPUTER AS A "TEXT BUFFER." SOURCE LISTINGS CAN BE ENTERED FROM A KEYBOARD OR OTHER INPUT DEVICE AND STORED IN THE COMPUTER'S MEMORY. INFORMATION THAT IS PLACED IN THE "TEXT BUFFER" IS KEPT IN AN ORGANIZED FASHION, USUALLY BY "LINES" OF TEXT. AN EDITOR PROGRAM GENERALLY HAS A VARIETY OF COMMANDS AVAILABLE TO THE OPERATOR TO ALLOW THE INFORMATION IN THE TEXT BUFFER TO BE MANIPULATED. FOR INSTANCE, LINES OF INFORMATION STORED IN THE TEXT BUFFER MAY BE ADDED, DELETED, MOVED ABOUT OR INSERTED BEFORE OTHER LINES, AND SO FORTH. NATURALLY, THE INFORMATION IN THE BUFFER CAN BE DISPLAYED TO THE OPERATOR ON AN OUTPUT DEVICE SUCH AS A CATHODE RAY TUBE OR ELECTRIC TYPING MACHINE. USING THIS TYPE OF PROGRAM, A PROGRAMMER CAN RAPIDLY CREATE A SOURCE LISTING AND MODIFY IT AS NECESSARY. WHEN A PERMANENT COPY IS DESIRED, THE CONTENTS OF THE "TEXT BUFFER" CAN BE PUNCHED ONTO PAPER TAPE OR WRITTEN ONTO A MAGNETIC TAPE CASSETTE. IT TURNS OUT THAT THE COPY PLACED ON PAPER TAPE OR A CASSETTE CAN OFTEN BE FURTHER PROCESSED BY ANOTHER PROGRAM TO BE DISCUSSED SHORTLY WHICH IS TERMED AN ASSEMBLER. HOWEVER, AN IMPORTANT REASON FOR MAKING A COPY OF THE TEXT BUFFER ON PAPER TAPE OR MAGNETIC CASSETTE TAPE IS BECAUSE IF IT IS EVER NECESSARY TO MAKE CHANGES TO THE SOURCE LISTING, THEN THE OLD LISTING CAN BE QUICKLY RELOADED BACK INTO THE COMPUTER, CHANGES RAPIDLY IMPLEMENTED USING AN EDITOR PROGRAM, AND A NEW "CLEAN" LISTING OBTAINED IN A FRACTION OF THE TIME REQUIRED TO ERASE AND RE-WRITE A LARGE NUMBER OF LINES USING PENCIL AND PAPER!

RELATIVELY SMALL PROGRAMS CAN BE DEVELOPED USING MANUAL METHODS - THAT IS BY WRITING THE SOURCE LISTINGS WITH PENCIL AND PAPER - BUT ANYONE THAT IS PLANNING ON DOING EXTENSIVE PROGRAM DEVELOPMENT WORK SHOULD OBTAIN AN EDITOR PROGRAM IN ORDER TO SUBSTANTUALLY INCREASE THEIR OVERALL PROGRAM DEVELOPMENT EFFICIENCY. BESIDES, AN EDITOR PROGRAM CAN BE PUT TO A LOT OF GOOD USED BESIDES MAKING UP SOURCE LISTINGS! SUCH AS ENABLING ONE TO EDIT CORRESPONDENCE OR PREPARE WRITTEN DOCUMENTS THAT ARE NICE AND NEAT IN LESS THAN HALF THE TIME REQUIRED BY CONVENTIONAL METHODS.

CHANGES IN SOURCE LISTINGS NATURALLY RESULT IN CHANGES TO THE MACHINE CODES (WHICH THE MNEMONICS SIMPLY "SYMBOLIZE"). EVEN MORE IMPORTANTLY, THE ADDRESSES ASSOCIATED WITH INSTRUCTIONS OFTEN MUST BE CHANGED DUE TO ADDITIONS OR DELETIONS OF "WORDS" OF MACHINE CODE. FOR INSTANCE, IN THE EXAMPLE ROUTINE BEING USED IN THIS SECTION, MEMORY ADDRESS PAGE 01 LOCATION 011 ORIGINALLY CONTAINED THE CODE FOR A "RET" (RETURN) INSTRUCTION WHICH IS 007. WHEN THE SUBROUTINE WAS CHANGED BY ADDING SEVERAL MORE INSTRUCTIONS (SO THE ANSWER WOULD BE STORED IN A MEMORY LOCATION) THE "RET" INSTRUCTION WAS SHIFTED DOWN TO THE ADDRESS PAGE 01 LOCATION 014. THE ADDRESS WHERE IT FORMERLY RESIDED WAS CHANGED TO HOLD THE CODE FOR THE FIRST PART OF THE "LLI 160" INSTRUCTION WHICH IS 066. HAD CHANGES BEEN MADE EARLIER IN THE ROUTINE, THEN MANY MORE MEMORY LOCATIONS WOULD NEED TO BE ASSIGNED DIFFERENT MACHINE CODES. HOWEVER, THE CHANGES CAUSED BY ADDING ON TO THE SAMPLE PROGRAM PREVIOUSLY DISCUSSED ARE NOT QUITE AS FAR REACHING AS THOSE THAT WOULD OCCUR IF CHANGES WERE MADE TO A PROGRAM SUCH AS THE ONE PRESENTED ON THE FOLLOWING PAGE, WHERE THE CHANGES RESULT IN THE ADDRESSES OF SUBROUTINES REFERRED TO BY OTHER ROUTINES BEING CHANGED - SO THAT IT IS THEN NECESSARY TO GO BACK

AND MODIFY THE MACHINE CODES IN ALL OF THE ROUTINES THAT REFER TO THE SUBROUTINE THAT IS CHANGED!

PAGE	LOC	MEMORY CONTENTS	LABELS/MNEMONICS		COMMENTS
00	000	026	OVER,	LCI 100	/LOAD REG 'C' WITH 100
00	001	100			
00	002	106		CAL NEWONE	/CALL A NEW SUBROUTINE
00	003	013			
00	004	000			
00	005	106		CAL LOAD	/AND THEN ANOTHER
00	006	023			
00	007	000			
00	010	104		JMP OVER	/JUMP BACK & REPEAT SEQUENCE
00	011	000			
00	012	000			
00	013	056	NEWONE,	LHI 000	/LOAD REG 'H' WITH 0'S
00	014	000			
00	015	066		LLI 200	/AND 'L' WITH 200
00	016	200			
00	017	317		LBM	/FETCH MEMORY CONTENTS TO 'B'
00	020	010		INB	/INCREMENT THE VALUE IN 'B'
00	021	371		LMB	/PLACE 'B' BACK INTO MEMORY
00	022	007		RET	/EXIT SUBROUTINE
00	023	056	LOAD,	LHI 003	/SET 'H' TO 003 (PAGE)
00	024	003			
00	025	361		LLB	/PLACE REG 'B' INTO 'L'
00	026	370		LMA	/PLACE ACC INTO MEMORY
00	027	021		DCC	/DECREMENT VALUE IN REG 'C'
00	030	013		RFZ	/RETURN IF 'C' NOT = 000
00	031	000		HLT	/STOP IF 'C' = 000

SUPPOSE IT WAS DECIDED TO INSERT A SINGLE WORD INSTRUCTION RIGHT AFTER THE "LCI 100" COMMAND IN THE ABOVE PROGRAM. THE NEW PROGRAM WOULD APPEAR AS SHOWN BELOW.

PAGE	LOC	MEMORY CONTENTS	LABELS/MNEMONICS		COMMENTS
00	000	026	OVER,	LCI 100	/LOAD REG 'C' WITH 100
00	001	100			
00	002	250		XRA	/CLEAR THE ACCUMULATOR
*00	003	106		CAL NEWONE	/CALL A NEW SUBROUTINE
*00	004	**014			
*00	005	000			
*00	006	106		CAL LOAD	/AND THEN ANOTHER
*00	007	**024			
*00	010	000			
*00	011	104		JMP OVER	/JUMP BACK & REPEAT SEQUENCE
*00	012	000			
*00	013	000			
*00	014	056	NEWONE,	LHI 000	/LOAD REG 'H' WITH 0'S
*00	015	000			
*00	016	066		LLI 200	/AND 'L' WITH 200
*00	017	200			
*00	020	317		LBM	/FETCH MEMORY CONTENTS TO 'B'
*00	021	010		INB	/INCREMENT THE VALUE IN 'B'

```
*00    022    371              LMB        /PLACE 'B' BACK INTO MEMORY
*00    023    007              RET        /EXIT SUBROUTINE
*00    024    056    LOAD,     LHI 003    /SET 'H' TO 003 (PAGE)
*00    025    003
*00    026    361              LLB        /PLACE REG 'B' INTO 'L'
*00    027    370              LMA        /PLACE ACC INTO MEMORY
*00    030    021              DCC        /DECREMENT VALUE IN REG 'C'
*00    031    013              RFZ        /RETURN IF 'C' NOT = 000
*00    032    000              HLT        /STOP IF 'C' = 000
```

NOTE IN THE ILLUSTRATION HOW NOT ONLY THE ADDRESSES OF ALL THE INSTRUCTIONS BEYOND LOCATION 002 (DENOTED BY THE *) CHANGE, BUT EVEN MORE IMPORTANT, THAT PARTS OF THE INSTRUCTIONS THEMSELVES (THE ADDRESS PORTION OF THE "CAL" INSTRUCTIONS - DENOTED BY THE **) MUST NOW BE ALTERED. THE ESSENTIAL POINT BEING MADE HERE IS THAT IF THE STARTING ADDRESS OF A ROUTINE OR SUBROUTINE THAT IS REFERRED TO BY ANY OTHER PART OF THE PROGRAM IS CHANGED, THEN EACH AND EVERY REFERENCE TO THAT ROUTINE MUST BE LOCATED AND THE ADDRESS PORTION CORRECTED! THIS CAN BE AN EXTREMELY FORMIDABLE, TIME CONSUMING, TEDIOUS, AND DOWN RIGHT FRUSTRATING TASK IF ALL THE REFERENCES MUST BE FOUND AND CORRECTED BY MANUAL MEANS IN A LARGE PROGRAM!

FORTUNATELY, THIS TYPE OF PROBLEM BECAME VIVIDLY APPARENT TO EARLY COMPUTER TECHNOLOGIST AND THEY SOON FOUND A METHOD TO EASE THE TASK OF MAKING SUCH CORRECTIONS BY DEVELOPING A TYPE OF PROGRAM CALLED AN "ASSEMBLER" THAT WOULD UTILIZE THE COMPUTER TO DO SUCH TASKS. "ASSEMBLER" PROGRAMS ARE TYPES OF PROGRAMS THAT ARE ABLE TO PROCESS "SOURCE LISTINGS" WRITTEN IN MNEMONIC (SYMBOLIC) FORM AND THEN TRANSLATE THEM INTO THE "OBJECT" (ACTUAL MACHINE LANGUAGE) CODE THAT IS UTILIZED DIRECTLY BY THE COMPUTER. AN ASSEMBLER ALSO KEEPS TRACK OF ASSIGNING THE PROPER ADDRESSES TO REFERENCES TO ROUTINES (THROUGH A PROCESS INITIATED BY ASSIGNING "LABELS" TO ROUTINES IN THE SOURCE LISTING). ONE CAN NOW SEE THAT THE COMBINATION OF AN EDITOR AND AN ASSEMBLER PROGRAM CAN GREATLY EASE THE TASK OF DEVELOPING MACHINE LANGUAGE PROGRAMS OVER THAT OF THE PURELY MANUAL METHOD WHICH BECOMES UNWIELDY AND NEXT TO IMPOSSIBLE WHEN THE PROGRAM SIZE BECOMES LARGE. ONE REASON THE COMBINATION IS SO VALUABLE IS BECAUSE IF A MISTAKE IN PROGRAMMING IS MADE, ONE CAN USE THE RELATIVELY QUICK METHOD OF UTILIZING AN EDITOR PROGRAM TO REVISE THE SOURCE LISTING, AND THEN USE THE ASSEMBLER PROGRAM TO PROCESS THE CORRECTED SYMBOLIC LISTING AND PRODUCE A NEW VERSION OF THE MACHINE CODE ASSIGNED TO THE APPROPRIATE ADDRESSES.

FOR QUITE SMALL PROGRAMS - SAY LESS THAN 100 INSTRUCTIONS, THE USE OF EDITOR AND ASSEMBLER PROGRAMS ARE NOT MANDATORY. IN FACT, EVEN IF ONE USES THESE AIDS FOR SMALL PROGRAMS, ONE SHOULD KNOW HOW TO CONVERT MNEMONIC LISTINGS TO OBJECT (MACHINE CODE) AS IT WILL OCCASIONALLY BE BENEFICIAL TO BE ABLE TO MAKE MINOR PROGRAM CHANGES ("PATCHES") WITHOUT HAVING TO GO THROUGH THE PROCESS OF USING AN EDITOR AND ASSEMBLER. THIS IS PARTICULARLY TRUE WHEN ONE IS "DEBUGGING" LARGE PROGRAMS AND WANTS TO ASCERTAIN WHETHER A MINOR CORRECTION WILL OPERATE AS PLANNED. THE PROCESS OF CONVERTING FROM A MNEMONIC LISTING TO ACTUAL MACHINE CODE IS NOT DIFFICULT IN CONCEPT. MANY READERS WILL HAVE DISCERNED THE PROCESS FROM THE EXAMPLES ALREADY PROVIDED. HOWEVER, FOR ANY WHO ARE IN DOUBT THE PROCESS WILL BE REVIEWED FOR THE SAKE OF CLARITY AT THIS TIME.

SUPPOSE A PERSON DESIRED TO PRODUCE A SMALL PROGRAM THAT WOULD SET THE CONTENTS OF ALL THE WORDS IN PAGE 01 OF MEMORY TO 000 (OCTAL). THE PROGRAMMER WOULD FIRST DEVELOP THE ALGORITHM AND WRITE IT DOWN AS A MNEMONIC (SOURCE) LISTING. SUCH AN ALGORITHM MIGHT BE AS FOLLOWS.

MNEMONIC	COMMENTS
LHI 001	/SET THE HIGH ADDRESS REGISTER TO PAGE 1
LLI 000	/SET THE LOW ADDRESS REGISTER TO THE FIRST /LOCATION ON THE PAGE ASSIGNED BY REG. "H"
AGAIN, LMI 000	/LOAD THE CONTENTS OF THE MEMORY LOCATION /SPECIFIED BY REGISTERS "H" & "L" TO 000
INL	/ADVANCE REGISTER "L" TO THE NEXT MEMORY /LOCATION (BUT DO NOT CHANGE THE PAGE)
JFZ AGAIN	/IF THE VALUE OF REGISTER "L" IS NOT 000 /AFTER IT HAS BEEN INCREMENTED THEN JUMP /BACK TO THE PART OF THE PROGRAM DENOTED BY /THE LABEL "AGAIN" AND REPEAT THE PROCESS
HLT	/IF THE VALUE OF REGISTER "L" IS TRULY 000 /THEN HAVE THE PROGRAM STOP

TO CONVERT THE SOURCE LISTING TO MACHINE (OBJECT) CODE THE PROGRAMMER MUST FIRST DECIDE WHERE THE PROGRAM IS TO RESIDE IN MEMORY. IN THIS PARTICULAR CASE IT WOULD CERTAINLY NOT BE WISE TO PLACE THE PROGRAM ANYWHERE ON PAGE 01 AS THE PROGRAM WOULD SOON "SELF DESTRUCT!" HOWEVER, THE PROGRAM COULD SAFELY BE PLACED ANYWHERE ELSE AND FOR THE SAKE OF THE DEMONSTRATION LET US ASSUME THAT IT IS TO RESIDE ON PAGE 02 STARTING AT LOCATION 100. TO CONVERT THE SOURCE LISTING TO MACHINE CODE THE PROGRAMMER WOULD SIMPLY MAKE A LIST OF THE ADDRESSES TO BE OCCUPIED BY THE PROGRAM AND THEN SIMPLY LOOK UP THE MACHINE CODE CORRESPONDING TO THE MNEMONIC FOR EACH INSTRUCTION AND PLACE THIS NUMBER NEXT TO THE ADDRESS IN WHICH IT WILL RESIDE. THE MACHINE CODE FOR EACH MNEMONIC USED BY THE **8008 CPU IS PROVIDED IN THE FIRST CHAPTER AS THE READER WILL RECALL.** SINCE SOME INSTRUCTIONS ARE "LOCATION DEPENDENT" IN THAT THEY REQUIRE THE ADDRESS OF REFERENCED ROUTINES, IT IS OFTEN NECESSARY TO ASSIGN THE MACHINE CODE IN TWO PROCESSES. THE FIRST PROCESS CONSIST OF ASSIGNING THE MACHINE CODES TO SPECIFIC MEMORY ADDRESSES WHERE-EVER POSSIBLE. WHEN THE MACHINE CODE REQUIRES AN ADDRESS THAT HAS NOT YET BEEN DETERMINED, THE MEMORY LOCATION IS LEFT BLANK. THE SECOND PROCESS CONSIST OF GOING BACK AND FILLING IN ANY BLANKS ONCE THE ADDRESSES OF REFERENCED ROUTINES HAVE BEEN DETERMINED. IN THE EXAMPLE BEING ILLUSTRATED, ONLY ONE PROCESS IS REQUIRED BECAUSE THE ADDRESS SPECIFIED BY THE LABEL "AGAIN" IS DEFINED BEFORE THE LABEL (ADDRESS) IS REFERENCED BY THE "JFZ" INSTRUCTION. THE SAMPLE PROGRAM CONVERTED TO MACHINE LANGUAGE WOULD APPEAR AS FOLLOWS.

ORIGINAL MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LHI 001	02 100	056	/MACHINE CODE FOR "LHI"
	02 101	001	/"IMMEDIATE" PART OF "LHI"
LLI 000	02 102	066	/MACHINE CODE FOR "LLI"
	02 103	000	/"IMMEDIATE" PART OF "LLI"
AGAIN, LMI 000	02 104	076	/MACHINE CODE FOR "LMI" /NOTE THAT THE LABEL "AGAIN" /NOW DEFINES AN ADDRESS OF /LOCATION 104 ON PAGE 02
	02 105	000	/"IMMEDIATE" PART OF "LMI"
INL	02 106	060	/INCREMENT LOW ADDRESS
JFZ AGAIN	02 107	110	/MACHINE CODE FOR "JFZ"
	02 110	104	/LOW ADDRESS PORTION OF THE /CONDITIONAL JUMP INSTRUCTION /DEFINED BY LABEL "AGAIN"

```
                  02  111        002     /PAGE ADDRESS PORTION OF THE
                                         /CONDITIONAL JUMP INSTRUCTION
                                         /DEFINED BY LABEL "AGAIN"
HLT               02  112        377     /ALTERNATELY, THE CODE 000 OR
                                         /001 COULD HAVE BEEN USED HERE
                                         /FOR THE "STOP" INSTRUCTION
```

ONCE THE PROGRAM HAS BEEN PUT IN MACHINE LANGUAGE FORM THE ACTUAL MACHINE CODE MAY BE PLAGED IN THE ASSIGNED LOCATIONS IN MEMORY AND THE PROGRAMMER MAY PROCEED TO VERIFY THE ALGORITHM'S VALIDITY. FOR SMALL PROGRAMS SUCH AS THE EXAMPLE JUST ILLUSTRATED THE MACHINE CODE CAN SIMPLY BE LOADED INTO THE CORRECT MEMORY LOCATIONS USING MANUAL METHODS TYPICALLY PROVIDED ON 8008 SYSTEMS. SUCH SMALL PROGRAMS CAN THEN BE EASILY CHECKED OUT BY "STEPPING" THROUGH THE PROGRAM.

IF THE PROGRAM IS RELATIVELY LARGE THEN A SPECIAL LOADER PROGRAM WHICH IS TYPICALLY AVAILABLE WITH AN ASSEMBLER PROGRAM WOULD BE USED TO LOAD IN THE MACHINE CODE.

CHECKING OUT AND "DEBUGGING" LARGE PROGRAMS CAN SOMETIMES BE DIFFICULT IF A FEW SIMPLE RULES ARE NOT FOLLOWED. A GOOD RULE OF THUMB IS TO FIRST TEST OUT EACH SUBROUTINE INDEPENDENTLY. ONE CAN CHOOSE TO "STEP" THROUGH A SUBROUTINE, OR ELSE TO PLACE "HALT" INSTRUCTIONS AT THE END OF EACH SUBROUTINE AND VERIFY THAT DATA WAS MANIPULATED PROPERLY BY THAT SUBROUTINE BEFORE GOING ON TO THE NEXT SECTION. THE USE OF STRATEGICALLY LOCATED "HALT" INSTRUCTIONS IN A PROGRAM INITALLY BEING TRIED OUT IS AN IMPORTANT METHOD FOR THE USER TO REMEMBER. WHEN A HALT IS ENCOUNTERED THE USER CAN CHECK THE CONTENTS OF MEMORY LOCATIONS AND EXAMINE THE CONTENTS OF CPU REGISTERS TO DETERMINE IF THEY CONTAIN THE PROPER VALUES AT THAT POINT IN THE PROGRAM (USING THE MANUAL OPERATOR CONTROLS AND INDICATOR LAMPS TYPICALLY PROVIDED ON 8008 DEVELOPMENT OR GENERAL PURPOSE SYSTEMS). IF ALL IS WELL AT THE HALT CHECK POINT THEN THE PROGRAMMER CAN REPLACE THE HALT INSTRUCTION WITH THE ACTUAL INSTRUCTION FOR THAT POINT AND CONTINUE CHECKING THE OPERATION OF THE PROGRAM AFTER MAKING CERTAIN THAT ANY REGISTERS THAT WERE ALTERED BY THE EXAMINATION PROCEDURE (TYPICALLY "H" AND "L") HAVE BEEN RESET TO THE DESIRED VALUE IF THEY WILL EFFECT OPERATION OF THE PROGRAM AS IT CONTINUES!

IT IS OFTEN HELPFUL TO USE A UTILITY PROGRAM KNOWN AS A "MEMORY DUMP" PROGRAM TO CHECK THE CONTENTS OF MEMORY LOCATIONS WHEN CREATING A NEW PROGRAM. THE MEMORY DUMP PROGRAM IS A SMALL UTILITY PROGRAM THAT WILL ALLOW THE CONTENTS OF AREAS OF MEMORY TO BE DISPLAYED ON AN OUTPUT DEVICE. NATURALLY, THE MEMORY DUMP PROGRAM MUST BE PLACED IN AN AREA OF MEMORY OUTSIDE THAT BEING USED BY THE PROGRAM BEING DEVELOPED. BY USING THIS TYPE OF PROGRAM THE OPERATOR CAN EASILY VERIFY THE CONTENTS OF MEMORY LOCATIONS - SAY BEFORE AND AFTER A SPECIFIC OPERATION OCCURRED TO SEE IF THEIR CONTENTS ARE AS EXPECTED. A MEMORY DUMP PROGRAM IS ALSO A VALUABLE AID IN DETERMINING THAT A PROGRAM HAS BEEN PROPERLY LOADED OR THAT A PORTION OF A PROGRAM IS STILL PRESENT, PERHAPS AFTER A PROGRAM UNDER TEST HAS GONE ERRANT!

ONE WILL FIND THAT HAVING FLOW CHARTS AND MEMORY MAPS AT HAND DURING THE "DEBUGGING" PROCESS IS ALSO VERY HELPFUL AS A REFRESHER ON WHERE ROUTINES ARE SUPPOSED TO BE IN MEMORY AND WHAT THE ROUTINES ARE SUPPOSED TO BE DOING.

IF MINOR CORRECTIONS ARE NECESSARY OR DESIRED, THEN ONE CAN OFTEN MAKE PROGRAM CORRECTIONS - OR "PATCHES" AS THEY ARE COMMONLY REFERRED TO BY SOFTWARE PEOPLE, TO SEE IF THE CORRECTIONS BELIEVED NECESSARY WILL WORK AS PLANNED. AN EASY WAY TO MAKE A "PATCH" TO A PROGRAM IS TO RE-

PLACE A "CALL" OR "JUMP" INSTRUCTION WITH A "CALL" TO A NEW SUBROUTINE THAT CONTAINS THE NECESSARY CORRECTIONS (PLUS THE ORIGINAL "CALL" OR "JUMP" INSTRUCTION IF NECESSARY)! IF A "CALL" OF "JUMP" INSTRUCTION IS NOT AVAILABLE IN THE VICINITY OF THE AREA WHERE A CORRECTION MUST BE MADE THEN ONE CAN REPLACE THREE WORDS OF INSTRUCTIONS WITH A "CALL" PATCH PROVIDED THAT ONE IS VERY CAREFUL NOT TO SPLIT UP A MULTI-WORD INSTRUCTION, OR, IF THIS CANNOT BE AVOIDED, THAT THE REMAINING PORTION OF A SPLIT UP MULTI-WORD INSTRUCTION IS REPLACED WITH A "NO OPERATION" INSTRUCTIONS SUCH AS "LAA." ONE MUST ALSO MAKE CERTAIN THAT THE INSTRUCTIONS DISPLACED BY THE INSERTED "CALL" INSTRUCTION ARE PLACED IN THE "PATCHING" SUBROUTINE (PROVIDED THAT THEY ARE NOT BEING REMOVED PURPOSELY)! AN EXAMPLE OF SEVERAL PATCHES BEING MADE TO THE SMALL SAMPLE PROGRAM JUST DISCUSSED WILL BE ILLUSTRATED BELOW.

SUPPOSE, IN THE EXAMPLE JUST DISCUSSED, THAT THE OPERATOR DECIDED NOT TO CLEAR (SET TO 000) ALL THE WORDS IN PAGE 01 OF MEMORY, BUT RATHER TO ONLY CLEAR THE LOCATIONS 000 TO 177 ON THE PAGE. THE PROGRAM COULD BE MODIFIED BY REPLACING THE "JFZ AGAIN" INSTRUCTION STARTING AT LOCATION 107 OF PAGE 02 WITH THE COMMAND "CAL 000 003" (CALL THE SUBROUTINE STARTING AT LOCATION 000 ON PAGE 03 WHICH WILL BE THE "PATCH"). NOW AT LOCATION 000 ON PAGE 03 ONE COULD PUT:

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LAI 200	03 000	006	/PUT VALUE 200 INTO
	03 001	200	/THE ACCUMULATOR
			/NOTE VALUE OF 200 USED BE-
			/CAUSE CONTENTS OF REGISTER
			/"L" ALREADY INCREMENTED!
CPL	03 002	276	/COMPARE CONTENTS OF THE
			/ACCUMULATOR WITH THE CON-
			/TENTS OF REGISTER "L"
JFZ AGAIN	03 003	110	/IF ACCUMULATOR AND "L" DO
	03 004	104	/NOT MATCH THEN CONTINUE THE
	03 005	002	/ORIGINAL PROGRAM
RET	03 006	007	/END OF "PATCH" SUBROUTINE

SUPPOSE INSTEAD OF FILLING EVERY WORD ON PAGE 01 WITH 000 THE PROGRAMMER DECIDED TO FILL EVERY OTHER WORD? A PATCH COULD BE MADE BY REPLACING THE "LMI 000" COMMAND AT LOCATIONS 104 AND 105, PLUS THE "INL" COMMAND AT LOCATION 106 OF PAGE 02 AND AGAIN INSERTING A "CAL 000 003" TO A PATCH SUBROUTINE THAT MIGHT APPEAR AS:

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LMI 000	03 000	076	/KEEP THE "LMI" INSTRUC-
	03 001	000	/AS PART OF THE PATCH
INL	03 002	060	/ORIGINAL "INL"
INL	03 003	060	/PLUS ANOTHER TO SKIP
			/EVERY OTHER WORD
RET	03 004	007	/EXIT FROM PATCH

FINALLY, TO ILLUSTRATE A PATCH THAT SPLITS A MULTI-WORD COMMAND,

CONSIDER A HYPOTHETICAL CASE WHERE THE PROGRAMMER DECIDED THAT PRIOR TO DOING THE CLEARING ROUTINE, IT WOULD BE IMPORTANT TO SAVE THE CONTENTS OF REGISTER "H" BEFORE SETTING IT TO PAGE 01. IF A THREE WORD "CALL" ROUTINE IS PLACED STARTING AT LOCATION 100 ON PAGE 02 IN THE ORIGINAL ROUTINE TO SERVE AS A PATCH, IT CAN BE SEEN THAT THE SECOND HALF OF THE "LLI 000" INSTRUCTION WOULD CAUSE A PROBLEM WHEN THE PROGRAM RETURNED FROM THE PATCH. (THE VALUE OF 000 AT LOCATION 103 ON PAGE 02 IN THE EXAMPLE WOULD BE INTREPRETED AS A "HLT" COMMAND BY THE COMPUTER WHEN IT RETURNED FROM THE PATCH SUBROUTINE)! IN ORDER TO AVOID THIS PROBLEM THE PROGRAMMER COULD PLACE A "LAA" (EFFECTIVELY A "NO OPERATION" COMMAND) AT LOCATION 103 ON PAGE 02 AFTER PLACING THE "CAL 000 003" INSTRUCTION BEGINNING AT LOCATION 100 ON PAGE 02 TO SERVE AS THE PATCH. THE ACTUAL PATCH SUBROUTINE MIGHT APPEAR AS SHOWN:

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LEH	03 000	345	/SAVE "H" IN REGISTER "E"
LHI 001	03 001	056	/NOW SET REGISTER "H" TO
	03 002	001	/POINT TO PAGE 01
LLI 000	03 003	066	/AND SET THE LOW ADDRESS
	03 004	000	/POINTER TO LOCATION 000
RET	03 005	007	/END OF PATCH SUBROUTINE

IN THE BALANCE OF THIS MANUAL NUMEROUS TECHNIQUES FOR DEVELOPING MACHINE LANGUAGE PROGRAMS WILL BE PRESENTED AND DISCUSSED. MANY OF THE EXAMPLES USED WILL BE PRESENTED AS SUBROUTINES THAT THE READER CAN USE DIRECTLY WHEN DEVELOPING CUSTOM PROGRAMS. IT IS IMPORTANT FOR THE NEW PROGRAMMER TO LEARN TO THINK OF PROGRAMS IN TERMS OF ROUTINES OR SUBROUTINES AND THEN LEARN TO COMBINE SUBROUTINES INTO LARGER PROGRAMS. THIS PRACTICE MAKES IT EASIER FOR THE PROGRAMMER TO INITIALLY DEVELOP PROGRAMS AS IT IS GENERALLY MUCH EASIER TO CREATE SMALL ALGORITHMS AND THEN COMBINE THEM, IN THE FORM OF SUBROUTINES, INTO THE LARGER ROUTINES. REMEMBER, SUBROUTINES ARE SEQUENCES OF INSTRUCTIONS THAT CAN BE CALLED BY OTHER PARTS OF THE PROGRAM. THEY ARE TERMINATED BY "RET" OR CONDITIONAL RETURN COMMANDS. IT IS ALSO WISE WHEN DEVELOPING PROGRAMS TO LEAVE SOME ROOM IN MEMORY BETWEEN SUBROUTINES SO THAT PATCHES CAN BE INSERTED OR ROUTINES LENGTHENED WITHOUT HAVING TO RE-ARRANGE THE CONTENTS OF A LARGE AMOUNT OF MEMORY. FINALLY, WHILE SPEAKING OF SUBROUTINES, IT WILL BE POINTED OUT THAT THE USER WOULD BE WISE TO KEEP A NOTE BOOK OF SUBROUTINES THAT THE INDIVIDUAL DEVELOPS IN ORDER TO BUILD UP A REFERENCE "LIBRARY" OF PERTINENT ROUTINES. IT TAKES TIME TO THINK UP AND CHECK OUT ALGORITHMS - AND ITS AWFUL EASY TO FORGET JUST HOW ONE HAD SOLVED A PARTICULAR PROGRAMMING PROBLEM SIX MONTHS AFTER ONE INITIALLY ACCOMPLISHED THE GOAL. SAVE YOUR ACCRUED EFORTS - THE MORE ROUTINES YOU HAVE TO UTILIZE - THE MORE VALUABLE YOUR MACHINE BECOMES, BECAUSE THE POWER OF THE MACHINE IS ALL DETERMINED BY WHAT YOU PUT IN ITS MEMORY!

BEFORE GOING ON TO THE NEXT SECTION, THE ESSENTIAL STEPS IN THE PROCESS OF CREATING A PROGRAM WILL BE PRESENTED AS A SUMMARY FOR READY REFERENCE ON THE FOLLOWING PAGE.

REVIEW OF THE PROCESS OF CREATING A MACHINE LANGUAGE PROGRAM

1.) FIRST, THE PROGRAMMER SHOULD CLEARLY DEFINE AND WRITE DOWN ON PAPER EXACTLY WHAT THE PROGRAM IS TO ACCOMPLISH.

2.) NEXT, FLOW CHARTS TO AID IN THE COMPLEX TASK OF WRITING THE MNEMONIC (SOURCE) LISTINGS ARE PREPARED. THEY SHOULD BE AS DETAILED AS NECESSARY FOR THE PROGRAMMER'S LEVEL OF EXPERIENCE AND ABILITY.

3.) MEMORY MAPS SHOULD BE USED TO DISTRIBUTE AND KEEP TRACK OF PROGRAM STORAGE AREAS AND DATA MANIPULATING REGIONS IN AVAILABLE MEMORY.

4.) USING THE FLOW CHARTS AND MEMORY MAPS AS GUIDES, THE ACTUAL SOURCE LISTINGS OF THE ALGORITHMS ARE WRITTEN USING THE SYMBOLIC REPRESENTATIONS OF THE INSTRUCTIONS. AN EDITOR PROGRAM IS FREQUENTLY USED TO GOOD ADVANTAGE AT THIS TIME.

5.) THE MNEMONIC SOURCE LISTINGS ARE CONVERTED INTO THE ACTUAL MACHINE LANGUAGE NUMERICAL CODES ASSIGNED TO SPECIFIC ADDRESSES IN MEMORY. AN ASSEMBLER PROGRAM MAKES THIS TASK QUITE EASY AND SHOULD BE USED FOR ALL BUT THE SMALLEST PROGRAMS.

6.) THE PREPARED MACHINE CODE IS LOADED INTO THE APPROPRIATE ADDRESSES IN THE COMPUTER'S MEMORY AND OPERATION OF THE PROGRAM IS VERIFIED. OFTEN THE INITIAL CHECK OUT IS DONE USING THE "STEP" MODE OF OPERATION, OR BY EXERCISING INDIVIDUAL SUBROUTINES. THE JUDICIAL USE OF INSERTED "HALT" INSTRUCTIONS AT KEY LOCATIONS WILL OFTEN BE OF VALUE DURING THE INITIAL TESTING PHASE.

7.) IF THE PROGRAM IS NOT PERFORMING AS INTENDED THEN PROBLEM AREAS MUST BE ISOLATED. PROGRAM "PATCHES" MAY BE UTILIZED TO MAKE MINOR CORRECTIONS. IF SERIOUS PROBLEMS ARE FOUND IT MAY BE NECESSARY TO RETURN TO STEP #3, OR EVEN STEP #1.

BASIC PROGRAMMING TECHNIQUES

THE FIRST SECTION OF THIS CHAPTER WILL BE DEVOTED TO ILLUSTRATING A NUMBER OF SIMPLE INSTRUCTIONS AND SEQUENCES OF INSTRUCTIONS THAT MAY BE USED TO ACCOMPLISH COMMONLY REQUIRED FUNCTIONS. NOVICE PROGRAMMERS NEED TO BUILD UP A REPERTOIRE OF SUCH ROUTINES IN THEIR MIND SO THAT THEY CAN LEARN TO THINK IN TERMS OF THE FUNCTIONS THEY PERFORM AS THEY PREPARE TO DEVELOP PROGRAMS OF THEIR OWN. ALTERNATIVE WAYS OF PERFORMING FUNCTIONS WILL SOMETIMES BE PRESENTED TO ILLUSTRATE ADVANTAGES AND DISADVANTAGES OF ONE METHOD OVER ANOTHER. THERE WILL OFTEN BE MANY OTHER WAYS OF PERFORMING THE DESIRED FUNCTION OTHER THAN THAT PRESENTED AND THE READER SHOULD FEEL FREE TO THINK OF OTHER WAYS AND LOOK AT POSSIBLE ADVANTAGES AND NEGATIVE ASPECTS OF SUCH ALTERNATIVES.

CLEARING THE ACCUMULATOR

IT IS OFTEN DESIRABLE TO SET THE CONTENTS OF THE ACCUMULATOR (ACC FOR ABREVIATION IN THIS TEXT) TO ZERO BEFORE STARTING AN OPERATION, SUCH AS A MATHEMATICAL CALCULATION. ONE OBVIOUS WAY TO DO THIS IS TO USE AN "LAI 000" INSTRUCTION. A LESS OBVIOUS WAY IS TO USE AN "XRA" (EXCLUSIVE OR THE CONTENTS OF THE ACC WITH ITSELF)! THE "XRA" METHOD ONLY REQUIRES ONE WORD, WHEREAS THE "LAI 000" REQUIRES TWO. ALSO, THE "XRA" METHOD WILL SET ALL THE CPU "FLAGS" TO KNOWN STATES AS ANY BOOLEAN LOGIC INSTRUCTION CAUSES THE "Z," "S," AND "P" FLAGS TO BE AFFECTED AND THE "C" FLAG TO BE SET TO THE ZERO STATE. (WHENEVER NECESSARY THE READER SHOULD REFER TO THE APPROPRIATE SECTION IN CHAPTER ONE OF THIS 8008 PROGRAMMING MANUAL TO REVIEW THE DETAILED FUNCTION(S) OF EACH TYPE OF INSTRUCTION AVAILABLE IN AN 8008 BASED MINI-COMPUTER). SINCE THE "XRA" INSTRUCTION WILL SET THE ACC TO ALL 0'S, THEN THE "Z" AND "P" FLAGS WILL BE PLACED IN THE "1" CONDITION, AND THE "S" FLAG TO THE "0" STATE AT THE CONCLUSION OF THE INSTRUCTION'S EXECUTION. IT IS IMPORTANT TO REMEMBER THE TYPES OF INSTRUCTIONS THAT AFFECT THE OPERATION OF THE CPU FLAGS BECAUSE IT IS OFTEN NECESSARY TO USE THE STATUS OF A FLAG OR FLAGS TO CONTROL THE OPERATION OF A PROGRAM - OR TO SEE IF A FLAG'S STATUS HAS CHANGED - AND TO DO THIS, ONE MUST AT SOME TIME "KNOW" WHAT THE CONDITION OF A FLAG WAS - AND THAT IS OFTEN ACHIEVED BY USING AN INSTRUCTION SUCH AS THE "XRA" THAT WILL "FORCE" THEM TO DESIRED STATES. ON THE OTHER HAND, WHILE THE "LAI 000" METHOD OF CLEARING THE ACC REQUIRES TWO MEMORY WORDS, THE EXECUTION OF AN "LAI 000" INSTRUCTION DOES NOT AFFECT THE STATUS OF THE CPU FLAGS, AND THIS FACT SHOULD BE REMEMBERED BECAUSE THERE MAY BE TIMES WHEN IT IS DESIRABLE TO SET THE ACC TO THE 0'S CONDITION WITHOUT ALTERING THE CPU FLAGS!

SETTING THE ACCUMULATOR TO ALL 1'S

THIS FUNCTION CAN BE ACCOMPLISHED WITH SEVERAL TYPES OF INSTRUCTIONS SUCH AS THE "LAI 377" OR "ORI 377." WHILE BOTH THESE INSTRUCTIONS REQUIRE TWO WORDS OF MEMORY, IT SHOULD BE NOTED AGAIN THAT THE "LAI 377" TYPE WILL NOT AFFECT THE STATUS OF THE CPU FLAGS, WHILE THE "ORI 377" ONE WILL RESULT IN THE "C" AND "Z" FLAGS BEING SET TO THE "0" STATE AND THE "S" AND "P" FLAGS SET TO THE "1" CONDITION. IF A PARTICULAR PROGRAM REQUIRES THE ACCUMULATOR TO BE SET TO THE ALL 1'S STATE FREQUENTLY THEN IT MAY BE WORTHWHILE TO SET UP A CPU REGISTER TO CONTAIN 377 AND THEN USE A ONE WORD INSTRUCTION SUCH AS "LAX" (X = A CPU REGISTER) OR AN "ORX" DEPENDING ON WHETHER OR NOT ONE WANTS TO SAVE THE STATUS OF THE CPU FLAGS.

COMPLEMENTING THE ACCUMULATOR

OFTEN IT IS DESIRABLE TO "COMPLEMENT" THE VALUE IN THE ACCUMULATOR, THAT IS TO CHANGE ALL THE BITS SET TO A "1" TO BE "0" AND VICE-VERSA. THIS CAN BE READILY ACCOMPLISHED BY USING AN "XRI 377" INSTRUCTION. AGAIN, IF THE FUNCTION MUST BE PERFORMED OFTEN IN A ROUTINE IT MAY BE WORTHWHILE TO KEEP THE VALUE 377 IN A CPU REGISTER AND USE A "XRX" INSTRUCTION TO PERFORM THE OPERATION AND REDUCE THE COMMAND TO A ONE WORD INSTRUCTION. THE COMPLEMENT FUNCTION IS OFTEN UTILIZED WHEN PERFORMING MATHEMATICAL OPERATIONS USING "SIGNED NUMBERS" (AS EXPLAINED IN THE PREVIOUS CHAPTER) IN ORDER TO OBTAIN THE "TWO'S COMPLEMENT" FORM OF A NUMBER. THE "TWO'S COMPLEMENT" OF A NUMBER IS OBTAINED BY FIRST COMPLEMENTING THE VALUF AND THEN ADDING ONE TO THE COMPLEMENTED VALUE. THUS THIS FUNCTION COULD BE OBTAINED BY PERFORMING TWO KINDS OF INSTRUCTIONS IN SEQUENCE - FIRST AN "XRI 377" AND THEN AN "ADI 001" COMMAND.

FORMING BIT "MASKS"

WHEN UTILIZING A COMPUTER IT IS FREQUENTLY DESIRABLE NOT TO USE ALL THE BIT POSITIONS WITHIN A WORD - OR TO ISOLATE AND DETERMINE THF STATUS OF A PARTICULAR BIT WITHIN A REGISTER. THIS TECHNIQUE FOR EXAMPLE, CAN BE USED TO QUICKLY DETERMINE WHETHER A NUMBER IN A REGISTER IS ODD OR EVEN (BY EXAMINING JUST THE LEAST SIGNIFICANT BIT), OR WHETHER A NUMBER HAS REACHED A CERTAIN SIZE (BY SAMPLING THE MOST SIGNIFICANT BIT OF INTEREST), OR WHETHER PERHAPS, SOME PARTICULAR EXTERNAL EVENT HAS OCCURED (BY CHECKING A SPECIFIC BIT ON AN INPUT PORT).

THE PROCESS OF RIDDING A REGISTER OF UNWANTED DATA IN SELECTED BIT POSITIONS IS COMMONLY REFERRED TO BY COMPUTER TECHNOLOGISTS AS "MASKING." MASKING CAN BE ACCOMPLISHED IN SEVERAL WAYS DEPENDING ON WHAT THE PROGRAMMER DESIRES. SUPPOSE, FOR INSTANCE, THAT ONF DESIRED TO DETERMINE WHETHER A NUMBER IN THE ACCUMULATOR WAS ODD OR EVEN. ONE WAY TO DO THIS WOULD BE TO SIMPLY EXECUTE AN "NDI 001" INSTRUCTION AND THEN TEST TO SEE IF THE ACCUMULATOR WAS ZERO (USING A "JTZ" OR "JFZ" COMMAND). SUPPOSE THE ORIGINAL NUMBER IN THE ACCUMULATOR HAD BEEN 251 (REMEMBER THAT THIS TEXT IS USING OCTAL NUMBERS UNLESS OTHERWISE STATED!) THE RESULTS OF PERFORMING THE LOGIC AND OPERATION BETWEEN THE ACCUMULATOR CONTAINING 251 AND THE NUMBER 001 IS ILLUSTRATED BELOW.

```
            ACCUMULATOR  =  1 0  1 0 1  0 0 1  =  OCTAL 251
AND IMMEDIATE WITH 001   =  0 0  0 0 0  0 0 1  =  OCTAL 001
                            ----------------

    RESULT LEFT IN ACC   =  0 0  0 0 0  0 0 1  =  OCTAL 001
```

IT CAN BE OBSERVED THAT ALL THE BIT POSITIONS "ANDED" WITH A 0 WILL GO TO THE 0 CONDITION REGARDLESS OF WHETHER THEY ARE A "1" OR A "0." THUS, THE SEVEN MOST SIGNIFICANT BIT POSITIONS IN THE EXAMPLE HAVE BEEN EFFECTIVELY ELIMINATED. HOWEVER, A BIT POSITION "ANDED" AGAINST A "1" WILL BE A "1" IF, AND ONLY IF, THE POSITION UNDER TEST CONTAINS A "1." IN THE ABOVE CASE, A "1" WAS PRESENT IN THE "TEST" POSITION AND THUS THE RESULT WAS A "1." A "JTZ" INSTRUCTION WOULD QUICKLY DIRECT THE PROGRAM TO PROCEED ON THE BASIS THAT THE ORIGINAL NUMBER IN THE ACC HAD BEEN AN ODD NUMBER.

NOTE THAT THE ABOVE PARTICULAR MASKING METHOD WAS DESTRUCTIVE TO THE ORIGINAL VALUE IN THE ACCUMULATOR. HAD IT BEEN IMPORTANT, THE ORIGINAL NUMBER COULD HAVE BEEN SAVED IN A CPU REGISTER OR A MEMORY LOCATION.

A SLIGHTLY DIFFERENT APPROACH COULD HAVE BEEN TAKEN. THE NUMBER TO BE "MASKED" COULD BE PLACED IN A MEMORY LOCATION, OR A CPU REGISTER. THEN THE ACCUMULATOR COULD BE FILLED WITH THE APPROPRIATE "MASK." FINALLY, A SIMPLE ONE WORD "NDM" OR "NDX" INSTRUCTION COULD BE UTILIZED. THE RESULT OF THE MASKING OPERATION WOULD BE LEFT IN THE ACCUMULATOR AFTER THE EXECUTION OF THE INSTRUCTION AND THE ORIGINAL NUMBER WOULD BE AVAILABLE FOR FURTHER MANIPULATION. THIS DIFFERENT APPROACH IS POINTED OUT AS AN EXAMPLE OF HOW A PROGRAMMER SHOULD LOOK FOR THE BEST METHOD TO APPROACH A PARTICULAR PROBLEM. THE COMPUTER, WITH ITS VARIETY OF INSTRUCTIONS, PROVIDES MANY DIFFERENT METHODS TO CHOOSE FROM FOR SUCH PROBLEMS.

MASKING IS MOST EFFECTIVE WHEN THERE ARE SEVERAL BITS IN A REGISTER TO BE ISOLATED, OR WHEN A BIT OF INTEREST IS IN THE MIDDLE OF A WORD, OR WHEN IT MAY NOT BE EXPEDIENT TO BRING A PIECE OF DATA INTO THE ACCUMULATOR. FOR, IF ONE DESIRES TO EXAMINE THE STATUS OF A BIT IN THE ACC THAT IS AT EITHER END OF THE REGISTER, ONE CAN DO THIS BY USING A ROTATE INSTRUCTION SUCH AS "RAL" OR "RAR" TO PUT THE BIT OF INTEREST INTO THE "CARRY" POSITION OF THE ACC (REPRESENTED BY THE CARRY FLAG) AND THEN USE A "JTC" OR "JFC" INSTRUCTION TO DETERMINE THE STATUS OF THE BIT. NATURALLY, IF THE PROGRAMMER WANTED TO RETAIN THE ORIGINAL SETTING OF THE ACCUMULATOR AFTER THE TEST THE PROGRAM WOULD HAVE TO EXECUTE THE REVERSE ROTATE INSTRUCTION (TO THE ONE ORIGINALLY USED) TO BRING THE ACC BACK TO ITS ORIGINAL PATTERN.

SETTING UP POINTERS AND COUNTERS

IN MANY APPLICATIONS IT IS DESIRABLE TO PERFORM A PARTICULAR SEQUENCE OF OPERATIONS A PRECISE NUMBER OF TIMES. THE NUMBER OF TIMES AN OPERATION IS PERFORMED CAN BE CONTROLLED IN A ROUTINE BY FORMING A "PROGRAM LOOP." A PROGRAM LOOP IS ESTABLISHED BY SETTING UP A COUNTER SYSTEM THAT KEEPS TRACK OF HOW MANY TIMES AN OPERATION IS PERFORMED AND INCLUDING A PROGRAM TEST TO ASCERTAIN WHEN A PARTICULAR VALUE HAS BEEN REACHED SO THAT THE PROGRAM CONTROL CAN BE "BRANCHED" OUT OF THE "LOOP."

IN AN 8008 SYSTEM CPU REGISTERS MAKE HANDY LOOP COUNTERS AS THEY NOT ONLY CAN BE DIRECTLY INCREMENTED OR DECREMENTED BY ONE WORD COMMANDS, BUT THEY ALSO DIRECTLY AFFECT THE STATUS OF THE "Z," "S," AND "P" CPU FLAGS AFTER EACH INCREMENT OR DECREMENT, MAKING IT AN EASY MATTER TO USE ANY ONE OF THE CONDITIONAL TYPE INSTRUCTIONS IMMEDIATELY FOLLOWING A CPU REGISTER INCREMENT OR DECREMENT, TO SEE IF A CRITICAL VALUE HAS BEEN REACHED!

FOR INSTANCE, SUPPOSE REGISTER "B" IS INITIALLY SET TO THE VALUE 012 (10 DECIMAL) BY A "LBI 012" INSTRUCTION PRIOR TO EXECUTION OF THE FOLLOWING "PROGRAM LOOP."

```
MORE,   LMA          /LOAD CONTENTS OF ACC INTO MEMORY
        INL          /ADVANCE MEMORY POINTER
        DCB          /DECREMENT THE LOOP COUNTER
        JFZ MORE     /IF REG "B" IS NOT = 000, CONTINUE LOOP
DONE,   HLT          /EXIT SUBROUTINE WHEN COUNTER = 000
```

AS CAN BE OBSERVED, THE ABOVE SUBROUTINE WOULD "LOOP" UPON ITSELF AND LOAD DATA INTO CONSECUTIVE WORDS IN MEMORY UNTIL THE VALUE PLACED IN REGISTER "B" (PRIOR TO STARTING THE SUBROUTINE) REACHED ZERO. IN THE ABOVE EXAMPLE "B" WAS LOADED WITH 012 SO 12 OCTAL (10 DECIMAL) LOCATIONS IN MEMORY WOULD HAVE BEEN LOADED WITH DATA. (IT CAN BE ASSUMED THAT THE

"CALLING" ROUTINE SET UP REGISTERS "H" & "L" TO POINT TO THE PROPER MEMORY LOCATIONS AND PLACED THE CORRECT DATA INTO THE ACCUMULATOR!)

TO ILLUSTRATE HOW POWERFUL THE SIMPLE CONCEPT OF A PROGRAM LOOP IS, A SECOND EXAMPLE WILL BE USED TO ILLUSTRATE HOW SUCH A LOOP TECHNIQUE CAN BE USED TO PERFORM MULTIPLICATION OF SMALL NUMBERS. (THERE ARE MUCH MORE EFFICIENT PROGRAMMING TECHNIQUES AVAILABLE FOR USE WITH LARGE NUMBERS.) SINCE MULTIPLICATION IS REALLY JUST REPEATED ADDITION, ONE COULD MULTIPLY TWO NUMBERS, DESIGNATED "X" AND "Y," BY PERFORMING THE FOLLOWING OPERATIONS. ASSUME "X" IS THE MULTIPLICAND AND IT HAS BEEN LOADED INTO CPU REGISTER "C." THE NUMBER "Y" IS THE MULTIPLIER AND IT HAS BEEN PLACED IN REGISTER "B." THE FOLLOWING ROUTINE CONTAINING A PROGRAM LOOP WILL "MULTIPLY" THE TWO NUMBERS.

```
START,  XRA          /CLEAR THE ACCUMULATOR
CONTIN, ADC          /ADD CONTENTS OF REGISTER "C" TO ACC
        DCB          /DECREMENT VALUE OF THE MULTIPLIER
        JFZ CONTIN   /REPEAT ADDITION IF MULT. IS NOT = ZERO
EXIT,   RET          /EXIT SUBRTN WITH MULT. ANSWER IN ACC
```

AS READERS KNOW, THE CPU REGISTERS "H" AND "L" WHILE BEING ABLE TO SERVE AS ORDINARY CPU REGISTERS, ALSO HAVE THE SPECIAL FUNCTION OF BEING ABLE TO "POINT" TO ADDRESSES IN MEMORY WHENEVER "MEMORY REFERENCE" INSTRUCTIONS ARE USED. THE "H" REGISTER HOLDS THE HIGH ADDRESS OR "PAGE" PORTION OF THE POINTER AND THE "L" REGISTER HOLDS THE LOW ADDRESS OR LOCATION ON A PAGE. NATURALLY, WHEN ONE DESIRES TO OPERATE ON DATA AT A LOCATION IN MEMORY VIA A MEMORY REFERENCE COMMAND, ONE MUST FIRST SET UP THE "H" AND "L" REGISTERS TO CONTAIN THE DESIRED ADDRESS. THIS IS READILY DONE WITH A "LHI XXX" AND "LLI YYY" COMBINATION OF INSTRUCTIONS. HOWEVER, MANY TIMES IT IS DESIRABLE TO DO A WHOLE SEQUENCE OF OPERATIONS THAT OPERATE UPON SEQUENTIAL LOCATIONS IN MEMORY. IN THIS CASE, ONCE THE INITIAL STARTING ADDRESS HAS BEEN LOADED INTO THE MEMORY POINTER REGISTERS, ALL THAT IS NEEDED IS A SUBROUTINE THAT CAN BE REFERED TO, THAT WILL INCREMENT THE ADDRESS HELD IN THE TWO REGISTERS. A SIMPLE SUBROUTINE TO ACCOMPLISH THAT OBJECTIVE IS PRESENTED HERE.

```
ADV,    INL          /INCREASE VALUE OF REGISTER "L" BY 1
        RFZ          /EXIT SUBRTN IF NOT GOING TO NEW PAGE
        INH          /INCREMENT "H" BY 1 IF ON NEW PAGE
        RET          /EXIT SUBRTN
```

THE ABOVE SUBROUTINE TAKES CARE OF THE CASE WHERE THE ADDRESS CROSSES "PAGE" BOUNDARIES. EACH TIME REGISTER "L" IS ADVANCED, THE "RFZ" INSTRUCTION IS USED TO TEST WHETHER OR NOT REGISTER "L" WENT TO 000. THIS WOULD OCCUR IF THE LAST VALUE IN THE REGISTER HAD BEEN 377, WHICH IS THE LARGEST OCTAL ADDRESS THAT CAN BE REPRESENTED IN AN 8 BIT REGISTER, AND CONSEQUENTLY THE HIGHEST ADDRESS THAT CAN BE ASSIGNED ON A "PAGE" OF MEMORY. IF THE "RFZ" INSTRUCTION IS EXECUTED (BECAUSE THE CONTENTS OF "L" DID NOT GO TO 000) THEN THE ROUTINE IS IMMEDIATELY EXITED. HOWEVER, IF THE "RFZ" COMMAND IS NOT FOLLOWED, THEN THE SUBROUTINE CONTINUES TO ADVANCE THE CONTENTS OF REGISTER "H" TO UPDATE THE POINTER TO A NEW PAGE. IN SOME CASES, WHERE THE PROGRAMMER IS GOING TO LIMIT ALL THE MANIPULATIONS OF DATA TO JUST ONE PAGE OF MEMORY, THE ABOVE SUBROUTINE COULD BE SHORTENED TO JUST TWO INSTRUCTIONS - "INL" FOLLOWED BY A "RET" COMMAND.

FINE. BUT WHAT ABOUT THE OPPOSITE CASE WHEN A PROGRAMMER MIGHT DE-

SIRE TO PROCESS AREAS OF MEMORY IN DESCENDING ORDER? WELL, A SIMILAR SUBROUTINE TO DECREMENT THE MEMORY POINTER REGISTERS COULD BE USED BUT NOW THE PROGRAMMER WILL HAVE TO BE CAREFUL WHEN GOING TO A NEW PAGE. IN THE PREVIOUS CASE, WHEN THE "L" REGISTER WAS ADVANCED BEYOND LOCATION 377 TO 000, IT WAS AN EASY MATTER TO CHECK FOR THE 000 CONDITION TO SEE IF IT WAS NECESSARY TO ADVANCE THE "H" REGISTER TOO. NOW, HOWEVER, WHEN THE "L" REGISTER GOES FROM 000 TO 377 IT WILL BE NECESSARY TO DECREMENT THE "H" REGISTER TO THE NEXT LOWER PAGE. TESTING FOR THIS CONDITION IS NOT QUITE AS EASY. REMEMBER, THE STATUS OF THE CPU FLAGS ARE SET BY THE CONDITIONS IN THE REGISTER IMMEDIATELY AFTER THEY HAVE BEEN INCREMENTED OR DECREMENTED - NOT BEFORE. AND, WHILE ONE CAN USE A "JTZ" OR "RFZ" TYPE OF INSTRUCTION TO QUICKLY DETERMINE IF A REGISTER WENT TO 000, THE CASE WHERE IT DID NOT GO TO 000, DOES NOT MEAN IT IS NECESSARILY AT 377 - IT COULD BE AT ANY NON-ZERO VALUE. HOWEVER, THE CASE CAN BE HANDLED. ONE WAY TO HANDLE THE PROBLEM WOULD BE WITH THE SUBROUTINE SHOWN BELOW.

```
DEC,    XRA          /CLEAR ACC TO 000
        CPL          /COMPARE CONTENTS OF ACC WITH "L"
        JTZ DECH     /IF 000 NOW, THEN DECR BOTH "H" & "L"
        DCL          /OTHERWISE JUST DECREMENT "L"
        RET          /AND EXIT SUBROUTINE
DECH,   DCL          /FOR THIS CASE DECREMENT "L"
        DCH          /AND REGISTER "H"
        RET          /THEN EXIT SUBROUTINE
```

WHILE THE ABOVE SUBROUTINE WILL ACCOMPLISH THE OBJECTIVE, IT DOES HAVE SEVERAL MINOR FLAWS THAT THE PROGRAMMER MIGHT WANT TO CONSIDER. FIRST, IT ALTERS THE CONTENTS OF THE ACCUMULATOR. REMEMBER, THAT THE ABOVE SUBROUTINE MIGHT OFTEN BE USED IN A PROGRAM THAT IS MANIPULATING DATA BETWEEN THE ACCUMULATOR AND MEMORY. THE ABOVE SUBROUTINE WOULD REQUIRE THAT THE PROGRAMMER MAKE SURE ANY VALUABLE DATA IN THE ACCUMULATOR IS "SAVED" ELSEWHERE BEFORE THE SUBROUTINE IS CALLED. THIS IS ONE MORE "BURDEN" ON THE PROGRAMMER WHO IS DEVELOPING A LARGE PROGRAM AND MAY HAVE A LOT OF OTHER DETAILS TO THINK ABOUT. SECONDLY, THE ABOVE ROUTINE REQUIRES 10 DECIMAL MEMORY STORAGE LOCATIONS. IT IS ALWAYS A GOOD PRACTICE TO TRY AND DEVELOP ROUTINES THAT OPERATE IN A MINIMUM AMOUNT OF MEMORY. LETS TAKE A LOOK AT ANOTHER SUBROUTINE THAT ACCOMPLISHES EXACTLY THE SAME OBJECTIVE, THAT SAVES 20 PERCENT OF MEMORY SPACE, AND THAT WILL NOT INTERFERE WITH THE ORIGINAL CONTENTS OF THE ACCUMULATOR.

```
DECR,   DCL          /DECREMENT CONTENTS OF "L"
        INL          /NOW CHECK TO SEE IF IT HAD BEEN 000
        JFZ NOT0     /IF NOT 000 THEN NOT GOING TO NEW PAGE
        DCH          /IF 000 THEN DECR "H" TO NEXT LOWER PAGE
NOT0,   DCL          /DECREMENT "L" TO COMPLETE SUBROUTINE
        RET          /EXIT SUBROUTINE
```

THE ABOVE SUBROUTINE USED A LITTLE PROGRAMMING CREATIVITY TO COME UP WITH A METHOD OF ACCOMPLISHING THE DESIRED OBJECTIVE. REGISTER "L" WAS DECREMENTED AND THEN INCREMENTED BACK TO ITS ORIGINAL VALUE. THE PROCESS OF INCREMENTING IT BACK TO ITS ORIGINAL VALUE WOULD CAUSE THE CPU FLAGS TO BE SET SO THAT A FLAG TESTING INSTRUCTION COULD BE USED TO SEE IF THE ORIGINAL VALUE WAS 000. IF THAT WAS THE CASE, DECREMENTING IT WOULD CAUSE IT TO GO TO 377, AND THUS REGISTER "H" SHOULD BE DECREMENTED TO THE NEXT LOWER PAGE. THAT IS DONE IF NECESSARY, AND THEN REGISTER "L" IS DECREMENTED TO ITS FINAL VALUE WHETHER OR NOT THE ADDRESS IS GO-

ING TO A NEW PAGE!

WHILE REGISTERS "H" AND "L" ARE THE ONLY REGISTERS THAT CAN BE USED TO POINT TO MEMORY LOCATIONS WHEN USING MEMORY REFERENCE INSTRUCTIONS, IT IS OFTEN NECESSARY TO USE OTHER CPU REGISTERS TO TEMPORARILY HOLD MEMORY ADDRESSES. IT MAY BE DESIRABLE, FOR INSTANCE, TO TRANSFER BLOCKS OF DATA FROM ONE AREA IN MEMORY TO ANOTHER. THIS MUST BE DONE ONE WORD AT A TIME. FIRST A WORD MUST BE EXTRACTED FROM MEMORY LOCATION "M" BY SAY A "LAM" INSTRUCTION WITH REGISTERS "H" AND "L" POINTING TO ADDRESS "M," AND THEN "H" AND "L" MUST BE ALTERED TO AN ADDRESS, LETS CALL IT "N," WHERE THE DATA IS TO BE DEPOSITED. AN "LMA" INSTRUCTION COULD THEN BE USED TO PLACE THE DATA IN THE NEW MEMORY LOCATION. OFTEN A STRING OF DATA WORDS MIGHT BE TRANSFERRED IN SUCH A FASHION. IT WOULD BE RATHER CUMBERSOME IF ONE HAD TO KEEP USING "LHI MMM" AND "LLI MMM" COMMANDS FOLLOWED BY "LHI NNN" AND "LLI NNN" INSTRUCTIONS IN ORDER TO KEEP ALTERING THE MEMORY POINTER REGISTERS BETWEEN THE TWO DIFFERENT AREAS IN MEMORY. HOWEVER, IF "H" AND "L" WERE INITIALLY SET TO POINT TO MEMORY LOCATION "M," AND CPU REGISTERS "D" (SAY FOR THE PAGE ADDRESS) AND "E" (FOR THE ADDRESS ON THE PAGE) WERE SET TO POINT TO MEMORY LOCATION "N," THEN A "SWITCHING" PROGRAM TO EXCHANGE THE CONTENTS OF "H" WITH "D" AND "L" WITH "E" COULD BE DEVELOPED TO CONSIDERABLY EASE THE TASK. SUCH A SUBROUTINE MIGHT BE AS FOLLOWS.

```
SWITCH, LCH         /LOAD CONTENTS OF "H" INTO "C" TEMPORARILY
        LHD         /NOW LOAD "D" INTO "H"
        LDC         /MOVE ORIGINAL "H" FROM "C" INTO "D"
        LCL         /SIMILARLY LOAD "L" INTO "C" TEMPORARILY
        LLE         /PUT "E" INTO "L"
        LEC         /AND STORE ORIGINAL "L" IN "E"
        RET         /EXIT SUBROUTINE
```

NOW, BY SIMPLING CALLING THE SUBROUTINE TO "SWITCH" THE CONTENTS OF THE REGISTERS, THE PROGRAMMER HAS A MEANS OF CHANGING THE MEMORY POINTER REGISTERS BETWEEN TWO DIFFERENT AREAS IN MEMORY. TO ILLUSTRATE HOW QUICKLY A LIBRARY OF SMALL SUBROUTINES STARTS DEVELOPING INTO REAL POTENTIAL, TWO SUBROUTINES ILLUSTRATED ON THE LAST SEVERAL PAGES WILL BE USED IN A SMALL PROGRAM TO ACCOMPLISH THE TASK JUST DISCUSSED - THAT OF MOVING DATA FROM ONE AREA OF MEMORY TO ANOTHER. LETS ASSUME THAT A PROGRAMMER DESIRED TO MOVE THE DATA IN 100 (OCTAL!) WORDS OF MEMORY STARTING AT LOCATION 000 ON PAGE 02 UP TO AN AREA STARTING AT LOCATION 200 ON PAGE 03. THE FOLLOWING PROGRAM WILL DO THE JOB NICELY.

```
SETUP,  LHI 002     /SET UP "H" TO PAGE OF 1ST MEMORY AREA
        LLI 000     /AND "L" TO STARTING LOCATION OF 1ST AREA
        LDI 003     /SET "D" TO PAGE OF 2ND MEMORY AREA
        LEI 200     /AND "E" TO STARTING LOCATION OF 2ND AREA
        LBI 100     /SET UP A COUNTER IN CPU REGISTER "B"
MOVIT,  LAM         /GET CONTENTS OF WORD FROM 1ST MEM AREA
        CAL ADV     /ADVANCE MEMORY POINTER (IN 1ST AREA)
        CAL SWITCH  /CHANGE "H" & "L" TO POINT TO 2ND AREA
        LMA         /DEPOSIT WORD IN 2ND AREA
        CAL ADV     /ADVANCE MEMORY POINTER (IN 2ND AREA)
        CAL SWITCH  /CHANGE BACK TO POINT TO 1ST MEMORY AREA
        DCB         /DECREMENT COUNTER
        JFZ MOVIT   /IF COUNTER NOT = 000, THEN CONTINUE MOVING
        RET         /EXIT RTN (OR "HLT" OR "JMP" ETC.)
```

USING MEMORY LOCATIONS TO STORE POINTERS AND COUNTERS

WHILE CPU REGISTERS MAKE IDEAL STORAGE PLACES FOR POINTERS AND COUNTERS BECAUSE THEY CAN BE DIRECTLY INCREMENTED AND DECREMENTED, THERE ARE SIMPLY NOT ENOUGH OF THEM TO STORE ALL THE POINTERS AND COUNTERS THAT MIGHT BE USED IN A FAIR SIZED PROGRAM. IT THEN BECOMES NECESSARY TO HOLD THE VALUES OF COUNTERS AND POINTERS IN MEMORY LOCATIONS SO THAT THE CPU REGISTERS CAN BE OPENED UP FOR OTHER USES. THIS PRACTICE DOES HAVE A DRAWBACK. SINCE THE CONTENTS OF MEMORY LOCATIONS CANNOT BE DIRECTLY INCREMENTED, THE CONTENTS MUST FIRST BE LOADED INTO A CPU REGISTER, THEN THE INCREMENT OR DECREMENT PERFORMED, THEN THE NEW VALUE PUT BACK INTO ITS MEMORY STORAGE LOCATION. THIS TAKES A LOT OF EXTRA INSTRUCTIONS OVER THAT REQUIRED IF THE COUNTER OR POINTER CAN BE KEPT PERMANENTLY IN A CPU REGISTER - ESPECIALLY SINCE TO EVEN OBTAIN THE COUNTER FROM MEMORY IT WILL ALWAYS BE NECESSARY TO FIRST SET UP THE "H" & "L" REGISTERS TO POINT TO THE MEMORY LOCATION WHERE THE COUNTER OR POINTER IS STORED! HOWEVER, SINCE THAT IS WHAT HAS TO BE DONE IN ALL BUT SMALL PROGRAMS, THE BEST THING TO DO IS TO TRY AND ORGANIZE THE PROCESS USING SUBROUTINES THAT WILL REDUCE THE AMOUNT OF MEMORY USED BY THE OPERATING PROGRAM.

PERHAPS THE FIRST ITEM TO CONSIDER IS WHERE TO STORE THE COUNTERS AND POINTERS FOR A PROGRAM. WELL, IT IS GENERALLY A GOOD IDEA TO SET ASIDE A SECTION OF MEMORY TO BE USED EXCLUSIVELY FOR STORING COUNTERS AND POINTERS FOR THE PROGRAM. PREFERABLY THIS SHOULD BE ON ONE PAGE OF MEMORY (VERSUS CROSSING PAGE BOUNDARIES). WHILE ESSENTIALLY ANY PAGE CAN BE USED, IT MAY BE THAT FOR LARGE PROGRAMS, HAVING THE POINTERS AND COUNTERS ON PAGE 00 WILL SAVE A BIT OF PROGRAMMING ROOM. THIS IS BECAUSE WHENEVER THE PROGRAM NEEDS TO REFER TO A COUNTER, REGISTER "H" (AS WELL AS "L") MUST BE SET UP TO POINT TO THE PAGE WHERE THE COUNTER IS STORED. IT SEEMS THAT THERE IS OFTEN A "ZERO" REGISTER (ONE SET TO 000) AROUND AMONG THE CPU REGISTERS AND THUS A "LHX" ONE WORD INSTRUCTION CAN BE USED TO SET "H" TO THE PAGE INSTEAD OF HAVING TO USE A "LHI XXX" COMMAND AS WILL GENERALLY BE THE CASE IF THE POINTERS AND COUNTERS ARE NOT STORED IN AN AREA ON PAGE 00.

ONCE ONE HAS DECIDED WHERE PARTICULAR COUNTERS ARE TO BE STORED, A SUBROUTINE TO RETRIEVE ANY ONE OF THEM AND INCREMENT OR DECREMENT THE VALUE, THEN RESTORE IT BACK TO MEMORY IS QUITE STRAIGHT-FORWARD.

```
CNTUP,  LCM         /FETCH CNTR INDICATED BY "H" & "L"
        INC         /INCREMENT VALUE OF THE COUNTER IN REG "C"
        LMC         /RESTORE NEW COUNTER VALUE TO MEMORY
        RET         /EXIT SUBROUTINE

CNTDWN, LCM         /FETCH COUNTER
        DCC         /DECREMENT VALUE
        LMC         /RETURN COUNTER TO STORAGE
        RET         /EXIT SUBROUTINE
```

THE TWO SUBROUTINES JUST ILLUSTRATED CAN BE CALLED AS DESIRED TO OBTAIN A COUNTER AND INCREMENT OR DECREMENT THE VALUE ONCE REGISTERS "H" AND "L" HAVE BEEN LOADED WITH THE ADDRESS OF THE COUNTER. NOTE TOO, THAT THE SUBROUTINE WOULD ALSO ALLOW THE RESULT OF THE INCREMENT OR DECREMENT TO BE TESTED BY A CONDITIONAL INSTRUCTION AFTER THE SUBROUTINE IS FINISHED BECAUSE THERE ARE NO INSTRUCTIONS AFTER THE "INC" OR "DCC" THAT AFFECT THE STATUS OF THE CPU FLAGS!

STORING POINTERS IN MEMORY IS GENERALLY A LITTLE MORE COMPLICATED THAN STORING COUNTERS BECAUSE POINTERS GENERALLY REQUIRE TWO STORAGE LOCATIONS. ONE WORD FOR THE PAGE ADDRESS AND ONE FOR THE LOCATION ON THE PAGE. IN ADDITION, SINCE THE "H" & "L" REGISTERS WILL HAVE TO BE USED TO POINT TO WHERE THE POINTERS ARE STORED IN MEMORY, AND SINCE THE POINTERS STORED IN MEMORY CANNOT BE USED AS POINTERS UNTIL THEY ARE PLACED IN THE "H" & "L" REGISTERS, A METHOD OF FIRST OBTAINING THE NEW POINTER INTO UNUSED CPU REGISTERS, THEN SWAPPING IT WITH THE "H" & "L" REGISTERS, MUST BE USED. THE PROCESS IS NOT SO DIFFICULT IF USE IS MADE OF SOME OF THE SUBROUTINES (SUCH AS SWITCH) WHICH HAVE ALREADY BEEN PRESENTED IN THIS CHAPTER.

THE EXAMPLE ILLUSTRATED NEXT SHOWS A GENERAL SUBROUTINE THAT WILL OBTAIN A TWO WORD POINTER STORED IN MEMORY, THEN USE THE POINTER OBTAINED TO PUT THE CONTENTS OF THE ACCUMULATOR INTO A MEMORY LOCATION SPECIFIED BY THE POINTER JUST OBTAINED. NEXT IT WILL INCREMENT THE POINTER AND THEN RESTORE IT BACK TO ITS STORAGE PLACE IN MEMORY. THE ROUTINE ASSUMES THAT THE "H" & "L" REGISTERS WILL BE SET TO THE PAGE ADDRESS OF THE LOCATION WHERE THE POINTER IS STORED BY THE CALLING PROGRAM, AND THAT THE POINTER IS STORED IN TWO CONSECUTIVE WORDS - FIRST THE PAGE AND THEN THE LOCATION ON THE PAGE.

```
POINT1, LDM          /FETCH POINTER PAGE ADDR INTO REG "D"
        INL          /ADVANCE TO PICK UP CONTENTS OF NEXT WORD
        LEM          /GET LOCATION ADDR INTO REGISTER "E"
        CAL SWITCH   /PUT NEW POINTER INTO "H" & "L"
        LMA          /PUT ACC INTO MEM INDICATED BY NEW POINTER
        CAL ADV      /INCREMENT THE NEW POINTER
        CAL SWITCH   /RESTORE NEW POINTER STORAGE ADDRESS
        LME          /DEPOSIT POINTER LOCATION ADDR IN MEM
        DCL          /DECREMENT BACK TO PAGE ADDR STORAGE WORD
        LMD          /DEPOSIT POINTER PAGE ADDR IN MEM
        RET          /EXIT SUBROUTINE
```

THE READER SHOULD NOTE A NICE FEATURE OF THE ABOVE SUBROUTINE. WHEN THE SUBROUTINE IS FINISHED THE CONTENTS OF "H" & "L" ARE SET TO POINT TO THE STORAGE AREA OF THE POINTER STORED IN MEMORY. THUS, THE SUBROUTINE COULD NOW BE CALLED AGAIN IF DESIRED WITHOUT HAVING TO SET UP THE "H" AND "L" REGISTERS AGAIN. FURTHERMORE, WHEN THE ROUTINE IS EXITED, CPU REGISTERS "D" & "E" WILL CONTAIN THE LATEST VALUE OF THE POINTER STORED IN MEMORY, WHICH MIGHT BE VALUABLE IN MANY CASES WHERE FURTHER PROCESSING WAS TO BE DONE IN THE SECTION OF MEMORY WHERE THE STORED POINTER WAS OPERATING.

EXAMINE THE SMALL PROGRAM ILLUSTRATED HERE.

```
BUFFIN, LHI 000     /SET PAGE WHERE BUFFER POINTER STORED
        LLI 240     /SET LOCATION ON PAGE OF BUFFER POINTER
INAGN,  CAL INPUT   /GET A CHARACTER FROM INPUT DEVICE
        CAL POINT1  /PUT THE CHARACTER INTO MEM BUFFER AREA
        CPI 215     /SEE IF CHAR WAS ASCII CODE FOR 'CR'
        JFZ INAGN   /IF NOT, GET ANOTHER CHARACTER
        RET         /EXIT RTN WHEN FIND A 'CR' CHARACTER
```

THE ABOVE PROGRAM, AS SHORT AND SIMPLE AS IT LOOKS, IS REALLY QUITE POWERFUL. THE READER SHOULD BE ABLE TO SEE THAT IT IS A PROGRAM THAT WILL STORE A STRING OF CHARACTERS RECEIVED FROM AN INPUT DEVICE INTO A

"BUFFER" AREA IN MEMORY. IT WILL CONTINUE PLACING CHARACTERS INTO THE MEMORY BUFFER AREA UNTIL IT DETECTS A 'CR' (CARRIAGE-RETURN) CHARACTER. THE LOCATION OF THE MEMORY BUFFER AREA IS STORED IN A POINTER THAT IS LOCATED AT LOCATIONS 240 (PAGE) AND 241 (LOCATION ON THE PAGE) ON PAGE 00. OF COURSE, BEFORE THE ABOVE ROUTINE WAS USED, THE PROGRAMMER WOULD WANT TO PUT THE PROPER ADDRESS FOR THE BUFFER AREA INTO THOSE LOCATIONS. THE ABOVE ROUTINE IS REALLY A GENERAL PURPOSE ROUTINE TO ACCEPT "TEXT SENTENCES" AND STORE THEM IN A MEMORY BUFFER. TO EXPAND THE ABOVE SUB-ROUTINE INTO A COMPLETE PROGRAM REQUIRES VERY LITTLE ADDITIONAL EFFORT.

```
DATAIN, LHI 000     /SET PAGE WHERE "POINT1" POINTER STORED
        LLI 240     /AND ADDRESS ON THE PAGE FOR "POINT1"
        LMI 003     /SET START OF MEMORY BUFFER AREA (PAGE)
        INL         /ADVANCE TO NEXT WORD
        LMI 000     /SET START OF MEM BUFF AREA (LOC ON PAGE)
        LLI 250     /ADDRESS OF A "LINE COUNTER"
        LMI 012     /SET LINE COUNTER TO 10 DECIMAL
MORIN,  CAL BUFFIN  /GET A LINE OF TEXT
        LHI 000     /SET UP STORAGE ADDR OF LINE COUNTER
        LLI 250     / "  "     "     "   "   "      "
        CAL CNTDWN  /DECREMENT LINE COUNTER VALUE
        JFZ MORIN   /IF NOT 10 (DEC) LINES, GET ANOTHER LINE
        HLT         /END OF PGM (COULD USE RET, JMP ETC.)
```

THE ABOVE PROGRAM FIRST "INITIALIZES" THE STARTING LOCATION OF THE "TEXT BUFFER" TO PAGE 03 LOCATION 000 BY SETTING THOSE VALUES INTO THE "POINT1" MEMORY STORAGE WORDS. IT ALSO INITIALIZES A COUNTER STORED IN MEMORY TO A VALUE DETERMINED BY THE PROGRAMMER. THEN THE SUBROUTINE THAT INPUTS LINES OF TEXT IS CALLED. EACH TIME A LINE OF TEXT IS OB-TAINED, THE "LINE COUNTER" IS DECREMENTED AND A DECISION MADE AS TO WHETHER OR NOT ANOTHER LINE OF TEXT SHOULD BE OBTAINED. WHEN A PRE-DETERMINED NUMBER OF LINES HAVE BEEN OBTAINED, THE PROGRAM STOPS. IN-STEAD OF STOPPING, HOWEVER, THE PROGRAM COULD HAVE BEEN DIRECTED TO PRO-CEED ELSEWHERE BY USING A "JMP" COMMAND, OR, THE ENTIRE PROGRAM COULD HAVE BEEN MADE A SUBROUTINE ITSELF BY USING A "RET" AS THE LAST INSTRUC-TION!

IT IS HOPED THAT THE READER IS RAPIDLY BEGINNING TO UNDERSTAND HOW QUICKLY SMALL, GENERAL PURPOSE SUBROUTINES, START DEVELOPING TREMENDOUS POTENTIAL AS THEY ARE TEAMED WITH OTHER ROUTINES. ALSO, THE READER SHOULD BEGIN TO SEE HOW THE USE OF MEMORY AUGMENTS THE CAPABILITY OF THE CPU REGISTERS - BY USING MEMORY LOCATIONS TO STORE POINTERS AND COUNTERS THE PROGRAMMER OPENS A WHOLE NEW DIMENSION TO THE WORLD OF PROGRAMMING. IT IS HOPED THE BEGINNING PROGRAMMER BECOMES A LITTLE BIT EXCITED AS THESE CONCEPTS ARE GRASPED AND UNDERSTOOD - FOR THESE CONCEPTS ARE JUST THE BEGINNING! AND EXCITEMENT STIMULATES THE IMAGINATION AND GIVES ONE INCENTIVE TO GO FORWARD AND INVESTIGATE AND LEARN MORE!

BEFORE GOING FURTHER, HOWEVER, IT MIGHT BE WISE TO SLOW THINGS DOWN FOR JUST A BIT AND RE-ITERATE THE IMPORTANCE OF KEEPING A PROGRAM ORGAN-IZED AS IT IS DEVELOPED. IN THE LAST SEVERAL PAGES, A NUMBER OF SUB-ROUTINES WERE PRESENTED, AND THEN COMBINED TO FORM LARGER SUBROUTINES, AND FINALLY THE "TEXT BUFFER INPUT" PROGRAM JUST PRESENTED. THE PROGRAM PRESENTED USES MEMORY STORAGE IN A VARIETY OF WAYS. FIRST THE PROGRAM ITSELF MUST BE STORED IN MEMORY. SECONDLY, OPERATIONAL PORTIONS OF THE PROGRAM REQUIRE MEMORY STORAGE AREAS FOR POINTERS AND COUNTERS. AND, LAST BUT NOT LEAST, THE PROGRAM REQUIRES THE USE OF MEMORY FOR "DATA" MANIPULATION IN THE FORM OF THE TEXT BUFFER. FURTHERMORE, THE "TEXT BUFFER INPUT" PROGRAM REALLY CONSIST OF A WHOLE GROUP OF SMALLER SUB-

ROUTINES. SUBROUTINES THAT MAY BE STORED IN DIFFERENT AREAS IN MEMORY. WHAT IS NEEDED, AS HAS BEEN DISCUSSED IN THE PREVIOUS CHAPTER, IS A MEMORY MAP TO HELP THE PROGRAMMER PLAN THE ALLOCATION OF MEMORY. IT MIGHT BE A GOOD IDEA FOR THE READER TO DEVELOP A MEMORY MAP FOR THE ABOVE PROGRAM AS PRACTICE. A GOOD METHOD TO FOLLOW WOULD BE TO SET ASIDE ROOM FOR THE MAIN PART OF THE PROGRAM (PERHAPS LEAVING A GOOD AMOUNT OF SPACE FOR EXPANDING THE PROGRAM IF DESIRED). THEN THE VARIOUS SUBROUTINES CAN BE ASSIGNED TO AREAS, POSSIBLY LEAVING A BIT OR ROOM BETWEEN EACH ONE IN THE EVENT FUTURE MODIFICATIONS ARE DESIRED. ONE CAN USE A SEPARATE MAP FOR EACH PAGE OF MEMORY WHERE ROUTINES ARE STORED. FOR AREAS SHOWING THE LOCATIONS OF COUNTERS AND POINTERS, THE MAPS MAY BE "EXPANDED" TO SHOW INDIVIDUAL ADDRESSES.

PG	LOC	RTN	NOTES
ØØ	24Ø	BUFFER	PG ADDR OF PNTR FOR "BUFFIN"
	241	POINTER	LOC ADDR " " " "
	242		
	243		
	244		
	245		
	246		
	247		
	25Ø	COUNTER	USED AS TEXT "LINE COUNTER"
	251		
	252		
	253		
	254		
	255		
	256		
	257		
	26Ø		
	261		
	262		
	263		
	264		
	265		
	266		
	267		
	27Ø		
	271		
	272		
	273		
	274		
	275		
	276		
▼	277		

EXPANDED MAP SHOWING LOCATIONS OF COUNTERS
AND POINTERS FOR THE TEXT BUFFER INPUT PROGRAM

PG	LOC	RTN	NOTES
02	ØØØ	DATAIN,	INPUT 1Ø Dec. Lines of Text
	1Ø		INTO BUFFER AREA ON PG Ø3
	2Ø		ORIG Version Requires X 3
	3Ø		(OCTAL) LOCS — Leave ROOM FOR
	4Ø		EXPANSION
	5Ø		
	6Ø		
	7Ø		
	1ØØ		
	11Ø		
	12Ø		
	13Ø		
	14Ø		
	15Ø		
	16Ø		
	17Ø		
	2ØØ	BUFFIN,	INPUT 1 LINE Text - 'CR'
	21Ø		ENDS LINE (2Ø LOCS)
	22Ø		
	23Ø	POINT1,	Fetch PNTR LOCS IN MEM
	24Ø		DESIGNATED BY CALLING RTN -
	25Ø		DEP ACC → MEM, ADV PNTR, RESTORE
	26Ø	SWITCH,	Exchange H & L with D & E
	27Ø	ADV,	INCR VALUE IN H & L
	3ØØ	CNTDWN,	DECR. CNTR STORED IN MEM
	31Ø		
	32Ø		
	33Ø		
	34Ø		
	35Ø		
	36Ø		
▼	37Ø		

SAMPLE MAP OF TEXT BUFFER INPUT PROGRAM
WITH MAIN ROUTINE AND SUBROUTINES ASSIGNED ON PAGE Ø2

THE SAMPLE MAPS SHOWN HERE ILLUSTRATE ONE WAY THE PROGRAM COULD BE ASSIGNED TO MEMORY LOCATIONS ON PAGE Ø2. NOTE HOW THE USE OF THE MAPS GIVES COHERENCE TO THE PROGRAM THAT IS NOT EASILY DISCERNED BY A PURELY MENTAL IMAGE! (PAGE Ø3 IS ASSUMED TO BE USED SOLELY AS A "TEXT BUFFER" AREA AND A MEMORY MAP FOR THE AREA IS NOT SHOWN).

ONCE THE MEMORY MAPS HAVE BEEN MADE UP AND THE STARTING ADDRESSES OF ALL THE SUBROUTINES ASSIGNED, IT IS AN EASY MATTER TO CONVERT THE MNEMONICS TO MACHINE CODE. AN ASSEMBLER PROGRAM MAY BE USED IF AVAILABLE. FOR PRACTICE, THE READER MIGHT WANT TO TRY DEVELOPING THE MACHINE CODE BY HAND. FOR COMPARISON PURPOSES THE OBJECT CODE FOR THE PROGRAM WOULD APPEAR AS SHOWN HERE IF THE SUBROUTINES ARE ASSIGNED TO THE ADDRESSES

AS SHOWN IN THE EXAMPLE MEMORY MAP.

```
 ADDR    CODE      MNEMONIC                  COMMENTS
------   ----   --------------      --------------------------

02 000   056  DATAIN, LHI 000     /SET PAGE WHERE "POINT1" POINTER STORED
02 001   000
02 002   066          LLI 240     /AND ADDRESS ON THE PAGE FOR "POINT1"
02 003   240
02 004   076          LMI 003     /SET START OF MEMORY BUFFER AREA (PAGE)
02 005   003
02 006   060          INL         /ADVANCE TO NEXT WORD
02 007   076          LMI 000     /SET START OF MEM BUFF AREA (LOC ON PG)
02 010   000
02 011   066          LLI 250     /ADDRESS OF A "LINE COUNTER"
02 012   250
02 013   076          LMI 012     /SET LINE COUNTER TO 10 DECIMAL
02 014   012
02 015   106  MORIN,   CAL BUFFIN /GET A LINE OF TEXT
02 016   200
02 017   002
02 020   056          LHI 000     /SET UP STORAGE ADDR OF LINE COUNTER
02 021   000
02 022   066          LLI 250     / "  "     "     "   "   "      "
02 023   250
02 024   106          CAL CNTDWN  /DECREMENT LINE COUNTER VALUE
02 025   300
02 026   002
02 027   110          JFZ MORIN   /IF NOT 10 (DEC) LINES, GET ANOTHER LINE
02 030   015
02 031   002
02 032   000          HLT         /END OF PGM (COULD USE RET, JMP ETC.)
    .
    .
    .
02 200   056  BUFFIN, LHI 000     /SET PAGE WHERE BUFFER POINTER STORED
02 201   000
02 202   066          LLI 240     /SET LOCATION ON PAGE OF BUFFER POINTER
02 203   240
02 204   106  INAGN,   CAL INPUT  /GET A CHARACTER FROM INPUT DEVICE
02 205   XXX
02 206   XXX
02 207   106          CAL POINT1  /PUT THE CHARACTER INTO MEM BUFFER AREA
02 210   230
02 211   002
02 212   074          CPI 215     /SEE IF CHAR WAS ASCII CODE FOR 'CR'
02 213   215
02 214   110          JFZ INAGN   /IF NOT, GET ANOTHER CHARACTER
02 215   204
02 216   002
02 217   007          RET         /EXIT RTN WHEN FIND A 'CR' CHARACTER
    .
    .
    .
02 230   337  POINT1, LDM         /FETCH POINTER PAGE ADDR INTO REG "D"
02 231   060          INL         /ADV TO PICK UP CONTENTS OF NEXT WORD
02 232   347          LEM         /GET LOCATION ADDR INTO REGISTER "E"
02 233   106          CAL SWITCH  /PUT NEW POINTER INTO "H" & "L"
02 234   260
02 235   002
```

```
ADDR    CODE      MNEMONIC                COMMENTS
------  ----  --------------  -------------------------

02 236  370           LMA       /PUT ACC INTO MEM INDICATED BY NEW PNTR
02 237  106           CAL ADV   /INCREMENT THE NEW POINTER
02 240  270
02 241  002
02 242  106           CAL SWITCH /RESTORE NEW POINTER STORAGE ADDRESS
02 243  260
02 244  002
02 245  374           LME       /DEPOSIT POINTER LOCATION ADDR IN MEM
02 246  061           DCL       /DECR BACK TO PAGE ADDR STORAGE WORD
02 257  373           LMD       /DEPOSIT POINTER PAGE ADDR IN MEM
02 250  007           RET       /EXIT SUBROUTINE
    .
    .
    .
02 260  325  SWITCH,  LCH       /LOAD CONTENTS OF "H" INTO "C" TEMP
02 261  353           LHD       /NOW LOAD "D" INTO "H"
02 262  332           LDC       /MOVE ORIG "H" FROM "C" INTO "D"
02 263  326           LCL       /SIMILARLY LOAD "L" INTO "C" TEMP
02 264  364           LLE       /PUT "E" INTO "L"
02 265  342           LEC       /AND STORE ORIGINAL "L" IN "E"
02 266  007           RET       /EXIT SUBROUTINE
    .
02 270  060  ADV,     INL       /INCREASE VALUE OF REG "L" BY 1
02 271  013           RFZ       /EXIT SUBRTN IF NOT GOING TO NEW PG
02 272  050           INH       /INCREMENT "H" BY 1 IF ON NEW PAGE
02 273  007           RET       /EXIT SUBROUTINE
    .
    .
    .
02 300  327  CNTDWN,  LCM       /FETCH COUNTER
02 301  021           DCC       /DECREMENT VALUE
02 302  372           LMC       /RETURN COUNTER TO STORAGE
02 303  007           RET       /EXIT SUBROUTINE
```

ORGANIZING AND MANIPULATING TABLES

A VERY POWERFUL FEATURE OF A DIGITAL COMPUTER IS ITS ABILITY TO STORE DATA AND TO PROCESS IT AS THE PROGRAMMER DESIRES - PERHAPS BY ARRANGING IT IN SOME SPECIFIC KIND OF ORDER, OR BY PERFORMING MATHEMATICAL OPERATIONS, SUCH AS OBTAINING AN AVERAGE, OR CONDENSING THE DATA IN SOME MANNER. THE COMPUTER IS ALSO SUITED FOR RAPIDLY EXTRACTING INFORMATION OF INTEREST FROM STORAGE BY PERFORMING SUCH FUNCTIONS AS "MATCHING" SIMILAR TYPES OF DATA, AND AS A "CONVERTING" MACHINE - WHERE DATA IN ONE TYPE OF CODE CAN BE QUICKLY CHANGED TO A DIFFERENT REPRESENTATION. IN SUCH APPLICATIONS, IT IS FREQUENTLY NECESSARY TO DEVELOP PROGRAMS THAT ORGANIZE DATA INTO "TABLES" OR TO PROCESS INFORMATION STORED IN "TABLE-LIKE" FORMAT.

THERE ARE A VARIETY OF WAYS TO ORGANIZE TABLES FOR COMPUTER PROCESSING. THE READER HAS ALREADY, WHETHER IT HAS BEEN REALIZED OR NOT, BEEN INTRODUCED TO SEVERAL TYPES OF "TABLES" IN THIS MANUAL. IN THE FIRST CHAPTER MENTION WAS MADE OF USING A "LOOK-UP" TABLE TO CONVERT BETWEEN ASCII AND BAUDOT CODES USED IN VARIOUS KINDS OF ELECTRIC TYPING MACHINES. AND, IN THIS CHAPTER, THE DISCUSSION AND PROGRAMMING CONSIDERATIONS FOR A "TEXT BUFFER" WERE ACTUALLY CONCERNED WITH A "FREE-FORM" TYPE OF TABLE.

FOR THE PURPOSES OF THE FOLLOWING DISCUSSION, TWO BASIC TYPES OF TABLE ORGANIZATIONS WILL BE DISCUSSED. ONE WILL BE REFERRED TO AS "FIXED-FORMAT" AND THE OTHER AS "FREE-FORMAT." THE FIXED-FORMAT TYPE OF TABLE REFERS TO TABLES THAT ARE FIXED BY PROGRAMMING CONSIDERATIONS INTO STRICT, UNCHANGING PATTERNS OF ORGANIZATION. THE FREE-FORMAT KIND USE DIFFERENT PROGRAMMING TECHNIQUES TO ALLOW THE STORAGE OF DATA IN RANDOM LENGTH SECTIONS OF MEMORY. THERE ARE ADVANTAGES AND DISADVANTAGES TO EACH FORMAT AND THE CHOICE OF WHICH ONE TO USE IS GENERALLY A FUNCTION OF THE TYPE OF TASK THAT IS TO BE PERFORMED. FREE-FORMAT ORGANIZATION IS GENERALLY MORE SUITABLE TO TEXT HANDLING TASKS. FIXED FORMAT ORGANIZATION IS GENERALLY THE CHOICE FOR "CONVERSION" TABLES. THERE ARE ALSO CASES WHERE THE CHOICE IS A RELATIVELY MINOR ONE AND IT BECOMES A MATTER OF THE PROGRAMMER'S PREFERENCE.

TO BEGIN DELVING INTO THE SUBJECT, A TABLE WITH MANY PRACTICAL APPLICATIONS WILL BE DISCUSSED. PROGRAMMING CONDIDERATIONS FOR DEVELOPING IT IN BOTH TYPES OF FORMATS WILL BE PRESENTED. IN MANY SITUATIONS, IT IS DESIRABLE FOR A COMPUTER PROGRAM TO HAVE A "CONTROL" TABLE. THAT IS A TABLE THAT WILL INTERPRET COMMANDS FROM AN INPUT DEVICE, AND DEPENDING ON WHAT IS RECEIVED, PERFORM A SPECIFIC TYPE OF FUNCTION. FOR THE PURPOSES OF THIS ILLUSTRATION IT WILL BE ASSUMED THAT AN OPERATOR WILL TYPE IN COMMANDS FROM A KEYBOARD. THE COMMANDS WILL BE IN THE FORM OF WORDS THAT MAY VARY IN LENGTH FROM 2 TO 6 CHARACTERS. WHENEVER A "WORD" HAS BEEN INPUTTED TO THE COMPUTER, THE COMPUTER WILL CHECK TO SEE IF THE "CONTROL TABLE" CONTAINS A MATCHING WORD, AND IF SO, THE COMPUTER WILL OBTAIN THE ADDRESS OF A ROUTINE THAT IT IS TO PERFORM AND EXECUTE THE FUNCTION. WHEN IT IS THROUGH PERFORMING THE ROUTINE, OR IF A "MATCH" FOR THE COMMAND WAS NOT FOUND, THE PROGRAM WILL RETURN TO THE "COMMAND" MODE AND WAIT FOR A NEW KEYBOARD ENTRY AFTER SENDING A RESPONSE ON AN OUTPUT DEVICE TO NOTIFY THE OPERATOR IT IS READY FOR A NEW ENTRY. FOR THIS EXAMPLE, THE OUTPUT DEVICE WILL BE ASSUMED TO BE AN ELECTRIC TYPEWRITER.

FOR A HYPOTHETICAL EXAMPLE, IT WILL BE PROPOSED THAT THE "CONTROL" WORDS WILL CONSIST OF THE FOLLOWING: "GO." "LIST." "MEDIAN." "AVG." "COUNT." "ERASE." THESE CONTROL WORDS MIGHT BE ASSOCIATED WITH A PROGRAM THAT IS TO BE USED BY A SCIENTIST CONDUCTING SOME TYPE OF EXPERIMENT. SUPPOSE THE CONTROL COMMAND "GO" INDICATED THE COMPUTER WAS TO START A 10 SECOND TIMING LOOP. AT THE START OF THE 10 SECOND TIME PERIOD THE PROGRAM WOULD SEND A "RESET" PULSE TO SOME SORT OF EXTERNAL COUNTING DEVICE THAT WAS COUNTING THE "EVENTS" THAT OCURRED IN SOME KIND OF EXPERIMENT. WHEN THE 10 SECOND PERIOD WAS OVER, THE COMPUTER WOULD IMMEDIATELY OBTAIN THE VALUE REGISTERED BY THE EXTERNAL COUNTER AND STORE THE NUMBER OBTAINED IN A "DATA BUFFER." THE "LIST" COMMAND MIGHT DIRECT THE COMPUTER TO PRINT OUT ALL THE DATA VALUES STORED IN THE "DATA BUFFER" (PERHAPS SO THE SCIENTIST COULD LOOK FOR PATTERNS OR JUST HAVE A COPY OF THE RAW EXPERIMENTAL DATA). THE "MEDIAN" COMMAND COULD DIRECT THE COMPUTER TO DETERMINE THE MEDIAN OR MIDDLE VALUE OUT OF ALL THE VALUES STORED IN THE DATA BUFFER AND PRINT OUT THAT NUMBER. SIMILARLY, THE "AVG" DIRECTIVE COULD SIGNIFY THAT THE PROGRAM WAS TO EXECUTE A ROUTINE TO CALCULATE THE AVERAGE VALUE OF THE DATA. THE "COUNT" COMMAND MIGHT BE USED TO HAVE THE COMPUTER INDICATE HOW MANY 10 SECOND EXPERIMENTS HAD BEEN CONDUCTED. AND, THE "ERASE" COMMAND COULD SIGNIFY THAT THE "DATA BUFFER" WAS TO BE "CLEANED OUT" FOR A NEW SET OF EXPERIMENTS.

THE CONTROL TABLE NEEDS TO BE CONSTRUCTED SO THAT THE PROGRAM CAN "SEARCH" FOR A "WORD" THAT IS THE SAME AS THAT ENTERED ON THE KEYBOARD AND IF A "MATCH" IS FOUND, THEN THE TABLE WOULD CONTAIN INFORMATION (AN ADDRESS) THAT WOULD DIRECT THE COMPUTER TO THE PROPER ROUTINE TO BE EX-

ECUTED. THE CONTROL TABLE COULD BE CONSTRUCTED BY SETTING ASIDE AN AREA IN MEMORY THAT CONTAINED THE PROPER CODE FOR THE LETTERS IN EACH "CONTROL WORD" FOLLOWED BY TWO MEMORY WORDS CONTAINING THE PAGE AND LOW ADDRESS WHERE THE APPROPRIATE ROUTINE RESIDED. IF THE CONTROL TABLE WAS CONSTRUCTED IN "FIXED-FORMAT" IT MIGHT APPEAR AS FOLLOWS.

FIXED-FORMAT CONTROL TABLE

ADDRESS	CONTENTS	REMARKS
02 000	307	/CODE FOR LETTER "G"
02 001	317	/ " " " "O"
02 002	000	/NOT USED FOR THIS COMMAND
02 003	000	/NOT USED FOR THIS COMMAND
02 004	000	/NOT USED FOR THIS COMMAND
02 005	000	/NOT USED FOR THIS COMMAND
02 006	001	/PAGE WHERE "GO" ROUTINE STARTS
02 007	100	/LOC ON PG WHERE "GO" STARTS
02 010	314	/CODE FOR LETTER "L"
02 011	311	/ " " " "I"
02 012	323	/ " " " "S"
02 013	324	/ " " " "T"
02 014	000	/NOT USED FOR THIS COMMAND
02 015	000	/NOT USED FOR THIS COMMAND
02 016	001	/PG WHERE "LIST" ROUTINE STARTS
02 017	140	/LOC ON PG WHERE "LIST" STARTS
02 020	315	/CODE FOR LETTER "M"
02 021	305	/ " " " "E"
02 022	304	/ " " " "D"
02 023	311	/ " " " "I"
02 024	301	/ " " " "A"
02 025	316	/ " " " "N"
02 026	001	/PG WHERE "MEDIAN" RTN STARTS
02 027	200	/LOC ON PAGE FOR "MEDIAN"
02 030	301	/CODE FOR LETTER "A"
02 031	326	/ " " " "V"
02 032	307	/ " " " "G"
02 033	000	/NOT USED FOR THIS COMMAND
02 034	000	/NOT USED FOR THIS COMMAND
02 035	000	/NOT USED FOR THIS COMMAND
02 036	001	/PG WHERE "AVG" ROUTINE STARTS
02 037	240	/LOC ON PAGE WHERE "AVG" STARTS
02 040	303	/CODE FOR LETTER "C"
02 041	317	/ " " " "O"
02 042	325	/ " " " "U"
02 043	316	/ " " " "N"
02 044	324	/ " " " "T"
02 045	000	/NOT USED FOR THIS COMMAND
02 046	001	/PG WHERE "COUNT" RTN STARTS
02 047	300	/LOC ON PG WHERE "COUNT" STARTS
02 050	305	/CODE FOR LETTER "E"
02 051	322	/ " " " "R"
02 052	301	/ " " " "A"
02 053	323	/ " " " "S"
02 054	305	/ " " " "E"
02 055	000	/NOT USED FOR THIS COMMAND
02 056	001	/PG WHERE "ERASE" RTN STARTS
02 057	340	/LOC ON PG WHERE "ERASE" STARTS
02 060	000	/**END OF TABLE MARKER**

IT CAN BE NOTED THAT THE FIXED-FORMAT TABLE OCCUPIES MEMORY FROM LOCATION 000 TO 060 (INCLUDING AN "END OF TABLE MARKER" WHICH WILL BE DISCUSSED LATER). OBSERVATION OF THE TABLE SHOWS THAT THERE IS A LOT OF "WASTED" SPACE WHERE MEMORY LOCATIONS ARE FILLED WITH ZEROS AS THE "COMMAND" WORD DID NOT REQUIRE SIX CHARACTERS. MORE CHARACTERISTICS OF THE ABOVE FORMAT WILL BE PRESENTED SHORTLY. FIRST, TWO SIMILAR "FREE-FORMAT" VERSIONS FOR THE SAME "CONTROL" TABLE WILL BE ILLUSTRATED.

FREE-FORMAT CONTROL TABLE - VERSION #1

ADDRESS	CONTENTS	REMARKS
02 000	307	/CODE FOR LETTER "G"
02 001	317	/ " " " "O"
02 002	000	/*END OF COMMAND WORD MARKER*
02 003	001	/PAGE WHERE "GO" ROUTINE STARTS
02 004	100	/LOC ON PG WHERE "GO" STARTS
02 005	314	/CODE FOR LETTER "L"
02 006	311	/ " " " "I"
02 007	323	/ " " " "S"
02 010	324	/ " " " "T"
02 011	000	/*END OF COMMAND WORD MARKER*
02 012	001	/PG WHERE "LIST" ROUTINE STARTS
02 013	140	/LOC ON PG WHERE "LIST" STARTS
02 014	315	/CODE FOR LETTER "M"
02 015	305	/ " " " "E"
02 016	304	/ " " " "D"
02 017	311	/ " " " "I"
02 020	301	/ " " " "A"
02 021	316	/ " " " "N"
02 022	000	/*END OF COMMAND WORD MARKER*
02 023	001	/PG WHERE "MEDIAN" RTN STARTS
02 024	200	/LOC ON PAGE FOR "MEDIAN"
02 025	301	/CODE FOR LETTER "A"
02 026	326	/ " " " "V"
02 027	307	/ " " " "G"
02 030	000	/*END OF COMMAND WORD MARKER*
02 031	001	/PG WHERE "AVG" ROUTINE STARTS
02 032	240	/LOC ON PAGE WHERE "AVG" STARTS
02 033	303	/CODE FOR LETTER "C"
02 034	317	/ " " " "O"
02 035	325	/ " " " "U"
02 036	316	/ " " " "N"
02 037	324	/ " " " "T"
02 040	000	/*END OF COMMAND WORD MARKER*
02 041	001	/PG WHERE "COUNT" RTN STARTS
02 042	300	/LOC ON PG WHERE "COUNT" STARTS
02 043	305	/CODE FOR LETTER "E"
02 044	322	/ " " " "R"
02 045	301	/ " " " "A"
02 046	323	/ " " " "S"
02 047	305	/ " " " "E"
02 050	000	/*END OF COMMAND WORD MARKER*
02 051	001	/PG WHERE "ERASE" STARTS
02 052	340	/LOC ON PG WHERE "ERASE" STARTS
02 053	000	/**END OF TABLE MARKER**

FREE-FORMAT CONTROL TABLE - VERSION #2

ADDRESS	CONTENTS	REMARKS
02 000	307	/CODE FOR LETTER "G"
02 001	317	/ " " " "O"
02 002	001	/PAGE WHERE "GO" ROUTINE STARTS
02 003	100	/LOC ON PG WHERE "GO" STARTS
02 004	314	/CODE FOR LETTER "L"
02 005	311	/ " " " "I"
02 006	323	/ " " " "S"
02 007	324	/ " " " "T"
02 010	001	/PG WHERE "LIST" ROUTINE STARTS
02 011	140	/LOC ON PG WHERE "LIST" STARTS
02 012	315	/CODE FOR LETTER "M"
02 013	305	/ " " " "E"
02 014	304	/ " " " "D"
02 015	311	/ " " " "I"
02 016	301	/ " " " "A"
02 017	316	/ " " " "N"
02 020	001	/PG WHERE "MEDIAN" RTN STARTS
02 021	200	/LOC ON PAGE FOR "MEDIAN"
02 022	301	/CODE FOR LETTER "A"
02 023	326	/ " " " "V"
02 024	307	/ " " " "G"
02 025	001	/PG WHERE "AVG" ROUTINE STARTS
02 026	240	/LOC ON PAGE WHERE "AVG" STARTS
02 027	303	/CODE FOR LETTER "C"
02 030	317	/ " " " "O"
02 031	325	/ " " " "U"
02 032	316	/ " " " "N"
02 033	324	/ " " " "T"
02 034	001	/PG WHERE "COUNT" RTN STARTS
02 035	300	/LOC ON PG WHERE "COUNT" STARTS
02 036	305	/CODE FOR LETTER "E"
02 037	322	/ " " " "R"
02 040	301	/ " " " "A"
02 041	323	/ " " " "S"
02 042	305	/ " " " "E"
02 043	001	/PG WHERE "ERASE" STARTS
02 044	340	/LOC ON PG WHERE "ERASE" STARTS
02 045	000	/**END OF TABLE MARKER**

THE READER CAN IMMEDIATELY NOTICE THAT BOTH OF THE FREE-FORMAT ORGANIZATIONS TAKE LESS MEMORY STORAGE FOR THE TABLE ITSELF THAN THE FIXED-FORMAT ARRANGEMENT. THIS IS GENERALLY THE CASE WHEN THERE ARE LARGE VARIATIONS IN THE LENGTH OF THE DATA (NUMBER OF MEMORY WORDS TO A "FIELD" SUCH AS THE "CONTROL WORDS" IN THE TABLES) THAT IS HELD IN THE TABLE. FOR FIXED-FORMAT TABLES, EACH "BLOCK" (IN THE EXAMPLE BEING DISCUSSED A BLOCK WOULD BE 8 MEMORY WORDS) MUST BE LONG ENOUGH TO CONTAIN THE LARGEST POSSIBLE FIELDS THAT COULD BE ENCOUNTERED IN THE APPLICATION. (IN THE PRESENT ILLUSTRATION, THE "FIELDS" IN A "BLOCK" WOULD BE THE "CONTROL WORD" FIELD AND THE "ADDRESS" FIELD. THE LARGEST "CONTROL WORD" FIELD REQUIRES 6 MEMORY WORDS. ALL THE "ADDRESS" FIELDS REQUIRE 2 WORDS - SO EACH BLOCK MUST HAVE 8 MEMORY LOCATIONS AVAILABLE). NOTE THAT A FIXED FORMAT TABLE MAY NOT REQUIRE MORE ROOM THAN A FREE-FORMAT TABLE OF THE TYPE SHOWN IN VERSION #1 IF THERE IS NOT A LARGE VARIATION IN THE LENGTH OF DATA WITHIN FIELD(S). FOR INSTANCE, HAD ALL OF THE

CONTROL WORDS BEEN SELECTED TO BE 5 AND 6 LETTERS IN LENGTH, THEN VERSION #1 WOULD HAVE ACTUALLY REQUIRED MORE MEMORY SPACE FOR THE TABLE THAN THE FIXED-FORMAT CONFIGURATION!

HOWEVER, THE AMOUNT OF MEMORY SPACE OCCUPIED BY THE TABLE ITSELF IS NOT THE ONLY PROGRAMMING POINT TO BE CONSIDERED WHEN CHOOSING THE TABLE FORMAT TO BE USED IN A PARTICULAR PROGRAM. ONE MUST ALSO LOOK AT SOME OTHER PARAMETERS THAT WILL ALSO HAVE AN EFFECT ON THE TOTAL SIZE OF THE PROGRAM. ONE SUBTLE PARAMETER, FOR INSTANCE, IS HOW WILL THE INPUTTED CHARACTER STRING FOR A "CONTROL WORD" BE "DELIMITED." SUPPOSE, FOR EXAMPLE, THAT A "CONTROL WORD" CHARACTER STRING IS INPUTTED VIA AN ASCII KEYBOARD SUBROUTINE AND STORED IN A SMALL BUFFER AREA IN MEMORY. ONE CAN ASSUME THAT THE ACTUAL INPUT STRING WAS "DELIMITED" (ENDED) BY A SPECIAL CHARACTER SUCH AS A "CARRIAGE-RETURN." THE "CARRIAGE-RETURN" WOULD INFORM THE INPUT ROUTINE TO CEASE ACCEPTING CHARACTERS AND RETURN TO THE "CALLING" PROGRAM. HOWEVER, SINCE THE CHARACTER STRING THAT IS RECEIVED MUST ALSO BE USED BY SOME OTHER ROUTINE (WHEN SEARCHING THE CONTROL TABLE FOR A MATCH), AND SINCE THE CHARACTER STRING CAN VARY IN LENGTH, THEN SOME MEANS MUST BE PROVIDED FOR TELLING THE TABLE SEARCH ROUTINE JUST HOW MANY CHARACTERS ARE IN THE PARTICULAR STRING OF CHARACTERS STORED IN THE BUFFER!

THIS CAN BE DONE IN SEVERAL DIFFERENT WAYS. ONE WAY WOULD BE TO HAVE THE "CARRIAGE-RETURN" CODE RECEIVED BY THE ASCII INPUT ROUTINE STORED AS THE LAST CHARACTER IN THE CHARACTER STRING BUFFER. THE TABLE SEARCH ROUTINE COULD USE THE "C-R" SYMBOL AS A "DELIMITER" TO SIGNIFY THE END OF THE CHARACTER STRING. THE CHARACTER STRING BUFFER WOULD THEN CONTAIN INFORMATION STORED AS SHOWN HERE:

ADDRESS LOCATION	CONTENTS
WORD #1	CODE FOR CHARACTER #1
WORD #2	CODE FOR CHARACTER #2
.	.
.	.
WORD #N	CODE FOR CHARACTER #N
WORD #N+1	CODE FOR CARRIAGE-RETURN

NOTE, THEN, THAT THE CHARACTER BUFFER WOULD HAVE TO BE A BLOCK OF LOCATIONS IN MEMORY LONG ENOUGH TO HOLD (N + 1) CHARACTERS WHERE "N" IS THE MAXIMUM NUMBER OF CHARACTERS ALLOWED IN A CONTROL WORD.

A SECOND WAY TO DELIMIT THE CHARACTER STRING IN THE BUFFER WOULD BE TO SET UP A COUNTER THAT INCREASED IN VALUE EACH TIME A CHARACTER WAS ACCEPTED INTO THE BUFFER. THE VALUE IN THE COUNTER WOULD THEN BE USED BY THE TABLE SEARCH ROUTINE TO INDICATE HOW LONG THE CHARACTER STRING WAS.

STILL ANOTHER TECHNIQUE WOULD BE TO UTILIZE A BUFFER ADDRESS POINTER THAT WOULD POINT TO THE ACTUAL ADDRESS OF THE LAST CHARACTER IN THE BUFFER.

THE SECOND AND THIRD SCHEMES ALLOW THE CHARACTER BUFFER TO BE JUST "N" CHARACTERS IN LENGTH (INSTEAD OF N + 1). HOWEVER, THE SAVINGS IN BUFFER SPACE IS HARDLY ENOUGH TO BE CONCERNED WITH, PARTICULARLY SINCE SOME OTHER LOCATION(S) WOULD HAVE TO BE SET ASIDE FOR STORING THE VALUE OF THE COUNTER OR BUFFER ADDRESS POINTER.

THE DIFFERENT METHODS ARE MENTIONED, HOWEVER, TO DEMONSTRATE THE IM-

PORTANT FACT THAT THERE IS MORE THAN ONE WAY TO APPROACH THE PROBLEM AND THE PROGRAMMER MUST DEVELOP THE PRACTICE OF EXAMINING ALTERNATIVE WAYS. WHILE THE DIFFERENCES ARE OFTEN SUBTLE, CERTAIN CHOICES MAY BE OF PARTICULAR VALUE IN CERTAIN APPLICATIONS.

AN IDEA THAT SHOULD BE MENTIONED AT THIS POINT CONCERNS THE PRACTICE OF TRYING TO DEVELOP PROGRAMS THAT ARE "GOOF-PROOF" - OR "HUMAN-ENGINEERED." THE IMPORTANCE OF THIS FACTOR SHOULD NOT BE OVER-LOOKED. FOR, THOSE THAT DO WILL OFTEN FIND THEMSELVES SPENDING MANY HOURS "REWORKING" PROGRAMS THAT HAVE SUDDENLY "GONE BESERK" WHILE IN OPERATION. THE ABILITY TO PLAN PROGRAMS THAT TAKE THIS IMPORTANT PARAMETER INTO CONSIDERATION GENERALLY DISTINGUISHES THE NOVICE FROM THE EXPERIENCED PROGRAMMER. WHAT IS MEANT BY "HUMAN-ENGINEERING" CAN BE CLEARLY DEMONSTRATED BY THE FOLLOWING DISCUSSION.

SUPPOSE FOR THE EXAMPLE BEING DEVELOPED HERE THAT THE PROGRAMMER ELECTED TO DEVELOP THE CHARACTER STRING INPUT ROUTINE USING SCHEME #1 PRESENTED ABOVE BY SETTING ASIDE A CHARACTER BUFFER N + 1 WORDS IN LENGTH (WHICH WOULD BE 7 IN THIS CASE AS THE MAXIMUM SIZE OF A CONTROL WORD IN THE EXAMPLE IS 6 CHARACTERS). NOW, A NOVICE, OR UNWARY BEGINNER MIGHT PROCEED TO DEVELOP THE ROUTINE ALONG THE FOLLOWING LINES.

```
                MNEMONIC                        COMMENTS
            ----------------                ----------------

INCTRL,  LHI XXX        /SET PAGE ADDR OF START OF CHAR BUFFER
         LLI YYY        /SET LOC ON PAGE OF START OF CHAR BUFFER
INCHAR,  CAL INPUT      /GET A CHARACTER FROM INPUT SUBROUTINE
         LMA            /STORE IN CHARACTER STRING BUFFER
         CPI 215        /SEE IF CHARACTER WAS A "C-R"
         RTZ            /EXIT SUBROUTINE IF "C-R"
         CAL ADV        /ADVANCE BUFFER POINTER
         JMP INCHAR     /LOOP TO GET NEXT CHARACTER
```

AN EXPERIENCED PROGRAMMER WOULD MORE LIKELY HAVE THE SUBROUTINE APPEAR SOMETHING LIKE:

```
                MNEMONIC                        COMMENTS
            ----------------                ----------------

INCTRL,  LHI XXX        /SET PAGE ADDR OF START OF CHAR BUFFER
         LLI YYY        /SET LOC ON PAGE OF START OF CHAR BUFFER
         LBI 006        /SET "SAFETY" COUNTER
INCHAR,  CAL INPUT      /GET A CHARACTER FROM INPUT SUBROUTINE
         CPI 215        /SEE IF CHARACTER WAS A "C-R"
         JFZ CHECK      /IF NOT "C-R" GO TO SAFETY CHECK ROUTINE
         LMA            /IF "C-R" THEN STORE IN BUFFER
         RET            /AND EXIT SUBROUTINE
CHECK,   INB            /EXERCISE REGISTER B TO SET FLAGS
         DCB            /FOR ITS ORIGINAL CONTENTS
         JTZ INCHAR     /IF "B" WAS 000, IGNORE PRESENT CHARACTER
         DCB            /OTHERWISE, DECREMENT VALUE OF "B"
         LMA            /STORE CHARACTER IN BUFFER
         CAL ADV        /ADVANCE BUFFER POINTER
         JMP INCHAR     /AND LOOP TO GET NEXT CHARACTER
```

WHAT DOES THE SECOND SUBROUTINE DO THAT THE FIRST DID NOT? IT GUAR-

ANTEES THAT IF SOMEBODY TYPES IN A CHARACTER STRING MORE THAN SIX CHARACTERS LONG THAT THE "BUFFER" WILL NOT "EXPAND" BEYOND ITS INTENDED LENGTH AND POSSIBLY RESULT IN CHARACTERS BEING LOADED INTO PORTIONS OF MEMORY THAT POSSIBLY CONTAIN PROGRAM INSTRUCTIONS OR OTHER DATA, THE ALTERING OF WHICH MIGHT EVENTUALLY RESULT IN A PROGRAM "BLOW-UP!"

STILL ANOTHER WAY TO DELIMIT AN INPUT CHARACTER BUFFER, AND A METHOD PARTICULARLY SUITED TO DEALING WITH A FIXED FORMAT TABLE, IS TO "CLEAR OUT" THE BUFFER PRIOR TO THE START OF ENTERING A CHARACTER STRING, BY FOR INSTANCE, INSERTING ALL "ZERO" WORDS INTO THE BUFFER. WHEN USING THIS METHOD IT IS NOT DESIRABLE TO INSERT A "C-R" AT THE END OF THE STRING, BUT RATHER TO SIMPLY ALLOW THE PRESENCE OF A "ZERO" WORD DENOTE THE END OF THE CHARACTER STRING.

ONCE THE INPUT CHARACTER BUFFER HAS RECEIVED A CHARACTER STRING AND A METHOD OF DELIMITING THE STRING BEEN SELECTED, ONE CAN PROCEED TO DEVELOP METHODS TO "SEARCH" THE CONTROL TABLE FOR A "CONTROL WORD" THAT MATCHES THE CHARACTER STRING IN THE BUFFER. THE SEARCH ROUTINE WILL REFLECT THE METHOD USED TO ORGANIZE THE TABLE AS WELL AS THE DELIMITING FORMAT USED IN THE CHARACTER STRING BUFFER. THE VARIOUS RAMIFICATIONS OF WHAT IS MEANT BY THIS CAN PERHAPS BEST BE CLARIFIED BY CONSIDERING A FEW PROGRAMMING EXAMPLES.

EXAMINE THE FOLLOWING PORTION OF A "SEARCH" ROUTINE DESIGNED TO LOOK FOR A MATCH BETWEEN THE CHARACTERS IN A BUFFER (TERMINATED BY A ZERO WORD) AND THE CHARACTERS CONTAINED IN THE "CONTROL WORD" FIELDS OF THE BLOCKS MAKING UP THE TABLE.

```
                MNEMONIC                    COMMENTS
              ------------                ------------

     SEARCH,  LDI 002      /SET POINTERS TO STARTING ADDR OF TABLE
              LEI 000      / "     "     "     "     "   "   "
     INITBF,  LHI XXX      /SET POINTERS TO START OF CHAR BUFFER
              LLI YYY      / "     "     "   "   "   "     "
              LBI 006      /SET CONTROL WORD FIELD SIZE COUNTER
     CMATCH,  LAM          /GET CHAR FM BUFFER (FORM CHAR MATCH LOOP)
              CAL ADV      /SUBROUTINE TO ADVANCE BUFFER POINTER
              CAL SWITCH   /EXCHANGE BUFFER PNTR FOR TABLE POINTER
              CPM          /SEE IF HAVE A MATCH CONDITION
              JFZ NXWORD   /IF NO MATCH, GO TO NEXT BLOCK IN TABLE
              DCC          /IF MATCH, DECR FIELD SIZE COUNTER
  **          JTZ FOUNDW   /ALL CHARS IN FIELD MATCHED IF CNTR = 0
              CAL ADV      /CHAR MATCH BUT NOT FINISHED, ADV PNTR
              CAL SWITCH   /EXCHANGE TABLE PNTR FOR BUFFER POINTER
              JMP CMATCH   /LOOP TO SEE IF NEXT CHARACTER MATCHES
     NXWORD,  DCB          /DECR FIELD SIZE CNTR TO FIND END OF
              JTZ SETNXW   /CURRENT CONTROL WORD FIELD, JMP WHEN END
              CAL ADV      /OTHERWISE ADVANCE TABLE POINTER
              JMP NXWORD   /AND LOOP TO LOOK FOR END OF CW FIELD
     SETNXW,  CAL ADV      /AT END OF CONTROL WORD FIELD NEED TO
              CAL ADV      /ADVANCE PNTR OVER THE "ADDRESS" FIELD
              CAL ADV      /TO THE START OF NEXT CONTROL WORD FIELD
  ***         CAL SWITCH   /AND THEN EXCHANGE TABLE FOR BUFFER PNTR
              JMP INITBF   /AND FORM LOOP TO CHECK NEXT BLOCK IN TBL
```

REMEMBER, THE ABOVE ROUTINE ASSUMES THAT THE INPUT CHARACTER BUFFER IS "CLEARED" BEFORE A NEW INPUT CHARACTER STRING IS ACCEPTED. THUS, THE INPUT BUFFER WOULD CONTAIN "ZEROS" IN THE LOCATIONS FROM "N + 1" TO THE

END OF THE BUFFER (WHERE "N" IS THE LAST CHARACTER OF THE INPUT STRING). IF, FOR EXAMPLE, THE INPUT BUFFER CONTAINED THE FOLLOWING:

BUFFER WORD #	CONTENTS
1	CODE FOR "G"
2	CODE FOR "O"
3	000
4	000
5	000
6	000

THEN THE ROUTINE JUST PRESENTED WOULD FIND A MATCH IN THE FIRST "BLOCK" OF THE FIXED FORMAT TABLE DESCRIBED SEVERAL PAGES EARLIER. WHEN THE MATCH WITH THE CONTROL WORD IN THE TABLE WAS FOUND, THE ROUTINE WOULD JUMP TO THE AS YET UNDEFINED "FOUNDW" ROUTINE TO EXTRACT THE ADDRESS OF THE "GO" ROUTINE FROM THE TABLE. HOWEVER, HAD THE INPUT CHARACTER BUFFER CONTAINED:

BUFFER WORD #	CONTENTS
1	CODE FOR "A"
2	CODE FOR "V"
3	CODE FOR "G"
4	000
5	000
6	000

THEN THE ROUTINE WOULD FAIL TO FIND A MATCH IN THE FIRST "CONTROL WORD" FIELD. WHEN THE MATCH FAILED IT WOULD JUMP TO THE "NXWORD" PORTION OF THE PROGRAM TO ADVANCE THE TABLE POINTER TO THE START OF THE NEXT "CONTROL WORD" FIELD IN THE TABLE, AND THEN JUMP BACK TO THE "INITBF" PORTION TO INITIALIZE THE CHARACTER BUFFER POINTER AND PROCEED TO LOOK FOR A MATCH IN THE NEXT BLOCK OF THE TABLE. THIS LOOP WOULD CONTINUE UNTIL THE MATCHING CONTROL WORD "AVG" WAS FOUND ABOUT HALF-WAY DOWN THE TABLE.

HAD SOME "SMART ALECK" OPERATOR KEYED IN THE FOLLOWING TO THE INPUT CHARACTER BUFFER:

BUFFER WORD #	CONTENTS
1	CODE FOR "S"
2	CODE FOR "I"
3	CODE FOR "L"
4	CODE FOR "L"
5	CODE FOR "Y"
6	000

THEN THE PROGRAM WOULD EVENTUALLY "BOMB!" REASON? (HERE COMES HUMAN ENGINEERING AGAIN!) SIMPLY THAT THE ABOVE ROUTINE HAS NO WAY OF DETERMINING WHERE THE END OF THE TABLE EXISTS IN MEMORY. THE HANDLING OF THAT PROBLEM WILL BE DISCUSSED SHORTLY AFTER SOME MORE EXAMPLES RELATED TO THE CURRENT TOPIC HAVE BEEN PRESENTED. THE READER SHOULD NOTE HERE THAT THE *** MARK NEAR THE END OF THE ROUTINE DENOTES A POINT WHERE AN

"END OF TABLE" TEST MIGHT BE INSERTED IN THE ABOVE ROUTINE.

IT IS DESIRABLE AT THIS POINT TO ILLUSTRATE SEVERAL OTHER "SEARCH" ROUTINES TO DEMONSTRATE HOW THEY ARE AFFECTED BY THE TABLE ORGANIZATION AND THE METHOD USED TO DELIMIT THE INPUT CHARACTER BUFFER. SUPPOSE ONE IS STILL USING THE FIXED-FORMAT TABLE BUT INSTEAD OF CLEARING OUT THE INPUT BUFFER BEFORE ACCEPTING A NEW CHARACTER STRING (SO THAT IT IS DELIMITED BY LOCATIONS CONTAINING ZEROS), ONE USES AN INPUT ROUTINE THAT DELIMITS THE BUFFER BY USING A "C-R" SYMBOL. THE ROUTINE TO LOOK FOR A MATCH BETWEEN THE CONTENTS OF THE BUFFER AND A "CONTROL WORD" IN THE TABLE MIGHT APPEAR AS FOLLOWS.

```
                MNEMONIC                    COMMENTS
            ----------------            --------------

          SEARCH, LDI 002       /SET POINTER TO STARTING ADDR OF TABLE
                  LEI 000       / "     "    "     "      "    "    "
          INITBF, LHI XXX       /SET POINTERS TO START OF CHAR BUFFER
                  LLI YYY       / "     "    "    "    "    "     "
                  LBI 006       /SET CONTROL WORD FIELD SIZE COUNTER
          CMATCH, LAM           /GET CHAR FM BUFFER (FORM CHAR MATCH LOOP)
                  CPI 215       /SEE IF SYMBOL FOR "C-R"
                  JTZ LCHAR     /IF SO, GO TO LAST CHARACTER ROUTINE
                  CAL ADV       /OTHERWISE, ADVANCE BUFFER POINTER
                  CAL SWITCH    /EXCHANGE BUFFER PNTR FOR TABLE POINTER
                  CPM           /SEE IF HAVE MATCH CONDX IN TABLE
                  JFZ NXWORD    /IF NO MATCH, GO TO NEXT BLOCK IN TABLE
                  CAL ADV       /IF MATCH, ADVANCE TABLE POINTER
                  CAL SWITCH    /EXCHANGE TABLE POINTER FOR BUFFER PNTR
                  DCB           /DECREMENT COUNTER VALUE (FOR NXWORD RTN)
                  JMP CMATCH    /LOOP TO SEE IF NEXT CHARACTER MATCHES
          LCHAR,  XRA           /IF "C-R" IN BUFFER, CLEAR ACCUMULATOR
                  CAL SWITCH    /EXCHANGE BUFFER POINTER FOR TABLE PNTR
                  CPM           /AND SEE IF HAVE 000 CODE IN TABLE
           **     JTZ FOUNDW    /IF SO, ALL CHARS IN FIELD MATCHED
                  INB           /IF NOT, SEE IF COUNTER IS AT 000
                  DCB           /INDICATING MAX CONTROL WORD FIELD
           **     JTZ FOUNDW    /ENCOUNTERED SO HAVE CONTROL WORD MATCH
            NXWORD, DCB         /IF NOT, DECR FIELD SIZE COUNTER
                  JTZ SETNXW    /IF CNTR = 0, AT END OF "CONTROL WORD" FLD
                  CAL ADV       /IF NOT, ADVANCE TABLE POINTER
                  JMP NXWORD    /AND LOOP TO LOOK FOR END OF FIELD
            SETNXW, CAL ADV     /AT END OF CONTROL WORD FIELD NEED TO
                  CAL ADV       /ADVANCE PNTR OVER THE "ADDRESS" FIELD
                  CAL ADV       /TO THE START OF NEXT CONTROL WORD FIELD
          ***     CAL SWITCH    /AND THEN EXCHANGE TABLE FOR BUFFER PNTR
                  JMP INITBF    /AND FORM LOOP TO CHECK NEXT BLOCK IN TBL
```

THE ABOVE ROUTINE IS A BIT MORE COMPLICATED THAN THE PREVIOUS ONE BECAUSE ONE MUST STILL KEEP TRACK OF THE NUMBER OF CHARACTERS THAT HAVE BEEN EXAMINED WITHIN A "CONTROL WORD FIELD" IN THE TABLE SECTION (FOR USE BY THE "NXWORD" ROUTINE), AND ALSO MAKE AN ADDITIONAL TEST FOR THE END OF THE CHARACTER STRING IN THE INPUT BUFFER WHICH IS SIGNIFIED BY THE CODE FOR A CARRIAGE-RETURN. IT IS ASSUMED IN THE ABOVE ROUTINE THAT THE ROUTINE THAT ACCEPTS A CHARACTER STRING INTO THE INPUT BUFFER LIMITS THE STRING TO A MAXIMUM OF SIX CHARACTERS. NOTE THAT ONE MUST ALSO MAKE SPECIAL PROVISIONS FOR THE CASE WHEN THE CHARACTER STRING IS SIX CHARACTERS IN LENGTH BY TESTING THE COUNTER IN THE "LCHAR" PORTION OF THE ABOVE ROUTINE.

THE COMBINATION OF USING A "C-R" TERMINATED BUFFER AND A FREE-FORMAT TABLE (SUCH AS THE FREE-FORMAT VERSION #1 ILLUSTRATED EARLIER) IS LESS COMPLICATED TO "SEARCH" BECAUSE ONE CAN DROP THE MAINTENANCE OF THE TABLE CONTROL WORD FIELD COUNTER AND INSTEAD TEST FOR THE END OF BUFFER MARKER (C-R) AND USE THE END OF FIELD MARKER (000) IN THE TABLE WHEN A MATCH FAILS AND IT IS NECESSARY TO ADVANCE TO THE NEXT CONTROL WORD IN THE TABLE. THIS SEARCH ROUTINE IS ILLUSTRATED NEXT.

```
                MNEMONIC                    COMMENTS
              ------------                ------------

          SEARCH, LDI 002      /SET POINTER TO STARTING ADDR OF TABLE
                  LEI 000      / "     "    "     "      "   "    "
          INITBF, LHI XXX      /SET POINTER TO START OF CHAR BUFFER
                  LLI YYY      / "     "    "     "    "   "      "
          CMATCH, LAM          /GET CHAR FM BUFFER (FORM CHAR MATCH LOOP)
                  CPI 215      /SEE IF SYMBOL FOR "C-R"
                  JTZ LCHAR    /IF SO, GO TO LAST CHARACTER ROUTINE
                  CAL ADV      /ADVANCE BUFFER POINTER
                  CAL SWITCH   /EXCHANGE BUFFER PNTR FOR TABLE POINTER
                  CPM          /SEE IF HAVE MATCH CONDITION IN TABLE
                  JFZ NXWORD   /IF NOT, GO TO NEXT BLOCK IN TABLE
                  CAL ADV      /IF YES, ADVANCE TABLE POINTER
                  CAL SWITCH   /EXCHANGE TABLE PNTR FOR BUFFER POINTER
                  JMP CMATCH   /LOOP TO TEST NEXT CHARACTER
          LCHAR,  XRA          /CLEAR ACCUMULATOR IF HAVE "C-R" IN BUFF
                  CAL SWITCH   /EXCHANGE BUFFER POINTER FOR TABLE PNTR
                  CPM          /SEE IF ALSO HAVE END OF FIELD MARKER
        **        JTZ FOUNDW   /IF SO, HAVE FOUND MATCHING CONTROL WORD
            NXWORD, LAM        /IF NOT, SEE IF HAVE END OF FIELD MARKER
                  NDA          /***TRICK TO SET FLAGS AFTER A LOAD OP***
                  JTZ SETNXW   /FOUND MARKER, GO TO NEXT BLOCK
                  CAL ADV      /MARKER NOT FOUND, ADVANCE TABLE POINTER
                  JMP NXWORD   /AND CONTINUE LOOKING FOR MARKER
          SETNXW, CAL ADV      /AFTER MARKER FOUND, ADVANCE TABLE PNTR
                  CAL ADV      /OVER THE "ADDRESS" FIELD TO THE START
                  CAL ADV      /OF THE NEXT CONTROL WORD FIELD
        ***       CAL SWITCH   /EXCHANGE TABLE PNTR FOR BUFFER POINTER
                  JMP INITBF   /AND FORM LOOP TO CHECK NEXT BLOCK IN TBL
```

AT FIRST GLANCE, DEVELOPING A SEARCH ROUTINE FOR THE FIXED-FORMAT TABLE - VERSION #2, WOULD APPEAR RATHER DIFFICULT BECAUSE THERE IS NO APPARENT END OF CONTROL WORD FIELD MARKER! HOWEVER, THAT TABLE WAS ORGANIZED TO TAKE ADVANTAGE OF A PARTICULAR FACT THAT THE DEVELOPER WAS AWARE OF THAT WOULD ENABLE THE FIRST PART OF THE "ADDRESS" FIELD TO BE USED AS AN END OF CONTROL WORD FIELD MARKER. THIS FACT IS THAT ALL OF THE CHARACTER CODES THAT MIGHT BE USED IN THE CONTROL WORD FIELD (WHICH CONSIST OF "ASCII" FORMATTED SYMBOLS) HAVE A "1" BIT IN ONE OR BOTH OF THE TWO MOST SIGNIFICANT BITS WITHIN A MEMORY WORD THAT CONTAINS THE CHARACTER. ADDITIONALLY, IT IS KNOWN THAT THE MAXIMUM PAGE ADDRESS THAT CAN BE UTILIZED IN A TYPICAL 8008 SYSTEM IS 077 (OCTAL) WHICH MEANS THAT A MEMORY WORD CONTAINING A MEMORY PAGE ADDRESS CANNOT HAVE A "1" CONDITION IN EITHER ONE OF THE TWO MOST SIGNIFICANT BITS OF THE MEMORY WORD THAT HOLDS THE PAGE ADDRESS! THUS, BY MAKING A SIMPLE TEST, USING A "MASKING" OPERATION DESCRIBED EARLIER IN THIS SECTION, A ROUTINE CAN BE DEVELOPED THAT CAN SAFELY UTILIZE THE PAGE ADDRESS PART OF THE ADDRESS FIELD TO SERVE AS AN END OF A "CONTROL WORD" FIELD! THUS, TO SEARCH VERSION #2 OF THE FREE-FORMAT TABLE, ONE COULD REPLACE THE ROUTINES "LCHAR" AND "NXWORD" USED ABOVE WITH THE FOLLOWING SUBSTITUTE:

```
MNEMONIC                    COMMENTS
------------                --------------

LCHAR,  CAL SWITCH   /EXCHANGE BUFFER POINTER FOR TABLE PNTR
        LAM          /TEST FOR END OF CONTROL FIELD
        NDI 300      /BY SEEING IF TWO MSB'S ARE BOTH "0"
        JTZ FOUNDW   /IF SO, HAVE FOUND MATCHING CONTROL WORD
NXWORD, LAM          /TEST FOR END OF CONTROL FIELD
        NDI 300      /BY SEEING IF TWO MSB'S ARE BOTH "0"
        JTZ SETNXW   /IF SO, HAVE MARKER, GO TO NEXT BLOCK
        CAL ADV      /OTHERWISE ADVANCE TABLE POINTER
        JMP NXWORD   /AND CONTINUE LOOKING
```

AS MENTIONED EARLIER, SOME MEANS OF DETERMINING WHEN THE ENTIRE TABLE HAS BEEN SEARCHED IN THE EVENT A **NON-EXISTENT** TERM IS PLACED IN THE INPUT BUFFER MUST BE INCORPORATED IN THE SEARCH ROUTINE. AGAIN, THIS TASK CAN BE ACCOMPLISHED IN SEVERAL DIFFERENT WAYS. ONE WAY WOULD BE TO SET A COUNTER AT THE START OF THE SEARCH ROUTINE THAT CONTAINED THE TOTAL NUMBER OF "BLOCKS" IN THE TABLE AND DECREMENT IT EACH TIME A BLOCK WAS CHECKED. THE COUNTER COULD BE TESTED FOR A ZERO CONDITION TO SIGNIFY THAT THE TABLE HAD BEEN SEARCHED. ANOTHER WAY TO ACCOMPLISH THE OBJECTIVE WOULD BE TO TEST THE VALUE OF THE TABLE POINTER TO SEE IF IT HAD REACHED A SPECIFIC VALUE WHICH WOULD DENOTE THE END OF THE TABLE. THESE TWO METHODS HAVE SEVERAL DRAWBACKS. ONE IS THAT THE COUNTER METHOD WOULD REQUIRE STORAGE SPACE. A CPU REGISTER COULD BE USED, BUT MORE THAN LIKELY ONE WOULD HAVE TO RESORT TO MAINTAINING A COUNTER IN A MEMORY LOCATION IN ORDER TO CONSERVE CPU REGISTERS - THIS WOULD REQUIRE A SOMEWHAT MORE LENGTHY ROUTINE TO HANDLE THE UPDATING AND TESTING OF THE COUNTER. TESTING TO SEE IF THE TABLE POINTER ADDRESS HAD REACHED A CERTAIN VALUE COULD BE DONE WITH AN "IMMEDIATE" TYPE COMPARISON THUS AVOIDING THE MAINTENANCE OF A STORAGE LOCATION BUT THE METHOD, ALONG WITH THE COUNTER METHOD, IS MORE COMBERSOME IF THE PROGRAMMER DECIDES TO EXPAND THE SIZE OF THE TABLE AT SOME FUTURE TIME. THIS IS BECAUSE THE PROGRAM WOULD HAVE TO BE MODIFIED AT TWO DIFFERENT POINTS - THE TABLE ITSELF, AND THE PORTION OF THE ROUTINE THAT SIGNIFIES THE END OF THE TABLE, EITHER THE COUNTER VALUE, OR THE ADDRESS POINTER VALUE.

A METHOD THAT IS GENERALLY MORE CONVENIENT IS TO PLACE A "ZERO WORD" AT THE END OF THE TABLE AS WAS SHOWN FOR THE EXAMPLE TABLES. THEN, AT THE START OF EACH NEW BLOCK, THE SEARCH ROUTINE CAN CONDUCT A SIMPLE TEST TO SEE IF A ZERO WORD IS PRESENT INDICATING THE END OF THE TABLE. (NATURALLY, IN SPECIAL CASES WHERE FOR INSTANCE A DATA BLOCK MIGHT CONTAIN A "ZERO WORD" AT THE FIRST LOCATION IN A BLOCK, THE METHOD WOULD NOT BE APPROPRIATE AND ONE COULD RESORT TO ONE OF THE ABOVE TECHNIQUES). THE METHOD OF USING A "ZERO WORD" ALSO MAKES IT EASY TO EXPAND THE SIZE OF THE TABLE WITHOUT HAVING TO MODIFY ANY PART OF THE SEARCH ROUTINE. MORE "BLOCKS" CAN SIMPLY BE ADDED (REPLACING THE FORMER "ZERO WORD") AND A NEW ZERO WORD ADDED AFTER THE ADDITIONAL BLOCKS. THE SEARCH ROUTINE, USING THE ALGORITHM PRESENTED BELOW, WOULD THEN AUTOMATICALLY BE ABLE TO FIND THE NEW "ENDING POINT" OF THE TABLE. THE FOLLOWING INSTRUCTIONS COULD SIMPLY BE INSERTED AT THE POINT INDICATED BY THE THREE ASTERISKS IN THE SEARCH ROUTINES PRESENTED EARLIER.

```
MNEMONIC                    COMMENTS
------------                --------------

        LAM          /FETCH FIRST CHARACTER IN NEW BLOCK
        NDA          /***TRICK TO SET FLAGS AFTER LOAD OP***
        JTZ NOSUCH   /IF ZERO, END OF TBL, NO MATCH FOUND
```

THE ROUTINE "NOSUCH" REFERRED TO BY THE END OF TABLE TEST MIGHT BE A SMALL ROUTINE TO DISPLAY A MESSAGE TO THE OPERATOR INDICATING THAT THERE WAS NO SUCH COMMAND IN THE TABLE. OR, THE JTZ INSTRUCTION MIGHT BE REPLACED BY AN "RTZ" INSTRUCTION THAT WOULD RETURN THE PROGRAM TO THE CALLING ROUTINE WHICH MIGHT SIMPLY DIRECT THE PROGRAM BACK TO THE ROUTINE WHICH FETCHES A NEW STRING OF CHARACTERS INTO THE INPUT BUFFER.

ONE OTHER PORTION OF THE SEARCH ROUTINE THAT HAS NOT BEEN TOUCHED UPON IS WHAT THE PROGRAM WOULD DO ONCE A MATCH WAS FOUND BETWEEN THE CHARACTERS IN THE INPUT BUFFER AND A CONTROL WORD FIELD IN THE TABLE. THIS PORTION OF THE ROUTINE WAS REFERRED TO AS "FOUNDW" IN THE PREVIOUS EXAMPLES. "FOUNDW" WOULD SIMPLY BE A ROUTINE THAT WOULD ADVANCE THE TABLE POINTER TO THE END OF THE CURRENT CONTROL WORD FIELD (WHERE THE MATCH OCCURED) AND THEN EXTRACT THE ADDRESS FROM THE ADDRESS FIELD TO ENABLE THE PROGRAM TO JUMP TO THE LOCATION GIVEN BY THE ADDRESS AND PROCEED TO PERFORM A SPECIFIC FUNCTION. THE ROUTINE "FOUNDW" AS GIVEN IN THE EXAMPLE THAT FOLLOWS CONTAINS AN INTRIGUEING PORTION THAT ILLUSTRATES ONE OF THE POWERFUL ASPECTS ABOUT A COMPUTER. THAT IS, A PROGRAM CAN BE DESIGNED TO ALTER THE EXECUTION OF THE PROGRAM ITSELF! THIS IS DONE IN THE EXECUTION OF THE "FOUNDW" ROUTINE WHEN THE PROGRAM EXTRACTS THE "ADDRESS" FROM THE TABLE AND INSERTS IT IN A PORTION OF THE PROGRAM FOR THE ADDRESS PORTION OF A "JUMP" INSTRUCTION WHICH THE PROGRAM THEN PROCEEDS TO EXECUTE! CARE MUST BE TAKEN WHEN DEVELOPING SUCH A PROGRAM TO ENSURE THAT EXACTLY THE RIGHT LOCATIONS ARE MODIFIED BY THE PROGRAM. THIS WILL BE APPARENT AFTER EXAMINATION OF THE FOLLOWING ROUTINE.

```
                MNEMONIC                        COMMENTS
            ----------------                ----------------

        FOUNDW, INB             /CHECK TO SEE IF THE FIELD CNTR IS 000
                DCB             /INDICATING END OF THE CONTROL WORD FIELD
        FNDEND, JTZ  SETJMP     /IF "0," SET UP THE JUMP ADDRESS
                CAL  ADV        /OTHERWISE ADVANCE TABLE POINTER
                DCB             /DECREMENT FIELD COUNTER
                JMP  FNDEND     /AND KEEP LOOKING FOR END OF FIELD
        SETJMP, CAL  ADV        /ADVANCE PNTR TO 1ST PART (PAGE) OF ADDR
                LDM             /AND EXTRACT PAGE ADDRESS & STORE TEMP
                CAL  ADV        /NOW ADVANCE PNTR TO LOC ON PG ADDRESS
                LEM             /AND STORE IT TEMPORARILY
                LHI  MMM        /NOW SET MEM PNTR (H & L) TO POINT TO THE
                LLI  NNN        /2ND BYTE OF THE JUMP INSTR. COMING UP
                LMF             /PUT THE LOW ORDER ADDR IN BYTE 2
                INL             /ADVANCE THE MEMORY POINTER
                LMD             /AND THE PAGE ADDR IN BYTE 3 OF THE JMP
                JMP  NNNMMM     /NOW JUMP TO THE ADDR JUST LOADED INTO
    NNN         AAA             /THESE TWO  (LOW ADDR)
    MMM         BBB             /BYTES      (PAGE ADDR)
```

THE ABOVE "FOUNDW" ROUTINE WAS FOR THE CASE WHERE THE TABLE WAS IN THE FIXED-FORMAT ORGANIZATION AND A COUNTER USED TO FIND THE END OF THE CONTROL WORD FIELD. HAD THE FREE-FORMAT TABLE BEEN USED, THEN THE BEGINNING PORTION OF "FOUNDW" WOULD BE APPROPRIATELY MODIFIED TO FIND THE END OF THE CONTROL WORD FIELD USING THE TECHNIQUES ILLUSTRATED IN THE "NXWORD" PORTION OF THE PREVIOUSLY ILLUSTRATED ROUTINES FOR THAT TYPE OF TABLE.

SINCE THE DISCUSSION OF HANDLING TABLES HAS EXTENDED OVER QUITE A FEW PAGES OF TEXT AND A VARIETY OF ROUTINES HAVE BEEN PRESENTED SHOWING VARIOUS PARTS OF THE PROCESS, IT MIGHT BE BENEFICIAL TO THE READER TO

PRESENT A NICELY PACKAGED SUMMARY BY PRESENTING TWO TABLE SEARCH ROUTINES. ONE USING THE FIXED-FORMAT TABLE COUPLED WITH AN INPUT CHARACTER STRING BUFFER (THAT IS CLEARED PRIOR TO ACCEPTING A NEW CHARACTER STRING). THE OTHER USING A FREE-FORMAT TABLE (VERSION #2) COUPLED WITH AN INPUT BUFFER THAT IS DELIMITED BY A CARRIAGE-RETURN. (THE ACTUAL ROUTINE THAT ACCEPTS CHARACTERS FROM AN I/O DEVICE WILL SIMPLY BE NOTED AS A SUBROUTINE CALL IN THE FOLLOWING EXAMPLES. THAT ROUTINE WOULD BE A FUNCTION OF THE I/O DEVICE USED AND TYPICAL ROUTINES WILL BE CONSIDERED IN THE CHAPTER ON I/O PROGRAMMING IN THIS MANUAL).

```
                MNEMONIC                    COMMENTS
                ----------                  ----------

                                    /
                                    /MAIN PROGRAM CALLING SEQUENCE
NEXCMD,  CAL  CLEARB    /CLEAR THE INPUT CHAR STRING BUFFER
         CAL  INCTRL    /FETCH THE COMMAND STRING FM INPUT DEVICE
         CAL  SEARCH    /SEARCH TABLE & PERFORM COMMAND INPUTTED
         JMP  NEXCMD    /REPEAT LOOP FOR NEXT COMMAND BY OPERATOR
                                    /
                                    /CLEAR INPUT BUFFER SUBROUTINE
CLEARB,  LHI  003       /SET PAGE PNTR TO START OF BUFFER
         LLI  372       /ASSUMMED TO BE AT LOC 372 ON PAGE 003
         LBI  006       /SET CLEARING COUNTER
         XRA            /CLEAR THE ACCUMULATOR
CLEARN,  LMA            /PUT 000 INTO BUFFER POSITION
         INL            /ADVANCE BUFFER POINTER
         DCB            /DECREMENT COUNTER
         JFZ  CLEARN    /IF NOT THROUGH, PUT 000 IN NEXT LOCATION
         RET            /WHEN THROUGH RETURN TO CALLING ROUTINE
                                    /
                                    /FETCH INPUT COMMAND STRING
INCTRL,  LHI  003       /SET PAGE ADDR OF START OF CHAR BUFFER
         LLI  372       /SET LOC ON PAGE OF START OF CHAR BUFFER
         LBI  006       /SET CNTR FOR MAXIMUM SIZE OF BUFFER
INCHAR,  CAL  INPUT     /CALL SUBROUTINE TO INPUT CHARACTER FM I/O
         CPI  215       /SEE IF CHARACTER WAS A "C-R"
         RTZ            /IF SO, MAKE NO ENTRY
CHECK,   INB            /EXERCISE REGISTER B (CNTR) TO SET FLAGS
         DCB            /ACCORDING TO ORIGINAL CONTENTS
         JTZ  INCHAR    /IGNORE NEW CHARACTER IF CNTR WAS 000
         DCB            /OTHERWISE DECREMENT VALUE OF CNTR
         LMA            /AND STORE CHARACTER IN BUFFER
         CAL  ADV       /ADVANCE BUFFER POINTER
         JMP  INCHAR    /AND LOOP TO FETCH NEXT CHARACTER FROM I/O
                                    /
                                    /TABLE SEARCH ROUTINE - COMPARES CHARACTER
                                    /STRING IN INPUT BUFFER AGAINST ENTRIES IN
                                    /THE CONTROL WORD FIELDS OF FIXED-FORMAT
                                    /TABLE (SIX LOCATIONS IN THE FIELD)
SEARCH,  LDI  002       /SET POINTERS TO STARTING ADDR OF TABLE
         LEI  000       / "     "     "     "     "   "    "
INITBF,  LHI  003       /SET POINTERS TO START OF CHAR BUFFER
         LLI  372       / "     "     "     "   "   "    "
         LBI  006       /SET CONTROL WORD FIELD SIZE COUNTER
CMATCH,  LAM            /GET CHAR FM BUFFER (FORM CHAR MATCH LOOP)
         CAL  ADV       /SUBROUTINE TO ADVANCE BUFFER POINTER
         CAL  SWITCH    /EXCHANGE BUFFER PNTR FOR TABLE POINTER
         CPM            /SEE IF HAVE A CHARACTER MATCH CONDX
         JFZ  NXWORD    /IF NO MATCH, GO TO NEXT BLOCK IN TABLE
```

```
                MNEMONIC                    COMMENTS
            ----------------            --------------

                    DCB                 /IF MATCH, DECR FIELD SIZE COUNTER
                    JTZ   FOUNDW        /IF CNTR = 0, ALL CHARS IN FIELD MATCHED
                    CAL   ADV           /CHAR MATCH BUT NOT FINISHED, ADV PNTR
                    CAL   SWITCH        /EXCHANGE TABLE PNTR FOR BUFFER POINTER
                    JMP   CMATCH        /LOOP TO SEE IF NEXT CHARACTER MATCHES
            NXWORD, DCB                 /DECR FIELD SIZE CNTR TO FIND END OF
                    JTZ   SETNXW        /CURRENT CONTROL WORD FIELD, JMP WHEN END
                    CAL   ADV           /OTHERWISE ADVANCE TABLE POINTER
                    JMP   NXWORD        /AND LOOP TO LOOK FOR END OF CW FIELD
            SETNXW, CAL   ADV           /AT END OF CONTROL WORD FIELD NEED TO
                    CAL   ADV           /ADVANCE PNTR OVER THE "ADDRESS" FIELD
                    CAL   ADV           /TO THE START OF NEXT CONTROL WORD FIELD
                    LAM                 /AND THEN FETCH 1ST CHAR IN NEW BLOCK
                    NDA                 /SET THE FLAGS AFTER THE LOAD OPERATION
                    RTZ                 /RETURN IF END OF TABLE (NO MATCH FOUND)
                    CAL   SWITCH        /OTHERWISE EXCHANGE TABLE PNTR FOR BUFF
                    JMP   INITBF        /AND FORM LOOP TO CHECK NEXT BLOCK IN TBL
            FOUNDW, CAL   ADV           /ADVANCE PNTR TO 1ST PART (PAGE) OF ADDR
                    LDM                 /AND EXTRACT PAGE ADDRESS TO STORE TEMP
                    CAL   ADV           /ADVANCE PNTR TO LOC ON PG ADDRESS
                    LEM                 /AND STORE IT TEMPORARILY
                    LHI   MMM           /NOW SET MEM PNTR (H & L) TO POINT TO THE
                    LLI   NNN           /2ND BYTE OF THE JUMP INSTR. COMING UP
                    LME                 /PUT THE LOW ORDER ADDR IN BYTE 2
                    INL                 /ADVANCE THE MEMORY POINTER
                    LMD                 /AND THE PAGE ADDR IN BYTE 3 OF THE JMP
                    JMP   NNNMMM        /NOW JUMP TO THE ADDR JUST LOADED INTO
            NNN     AAA                 /THESE TWO (LOW ADDR)
            MMM     BBB                 /BYTES     (PAGE ADDR)
                                        /
                                        /AT THE CONCLUSION OF THE ROUTINE THAT
                                        /THE "SEARCH" ROUTINE JUMPS TO WHEN A
                                        /MATCH IS FOUND, A "RET" INSTRUCTION
                                        /SHOULD BE EXECUTED TO RETURN THE PROGRAM
                                        /TO THE MAIN CALLING ROUTINE
                                        /
```

THE SUBROUTINES "SWITCH" AND "ADV" HAVE BEEN DETAILED EARLIER IN THIS CHAPTER AND ARE NOT REPEATED IN THE ABOVE EXAMPLE.

THE NEXT EXAMPLE IS FOR THE CASE WHERE THE INPUT BUFFER IS DELIMITED BY A CARRIAGE-RETURN AND A FREE-FORMAT TABLE (OF THE TYPE ILLUSTRATED AS VERSION #2) IS USED.

```
                MNEMONIC                    COMMENTS
            ----------------            --------------

                                        /
                                        /MAIN PROGRAM CALLING SEQUENCE
            NEXCMD, CAL   INCTRL        /FETCH THE COMMAND STRING FM INPUT DEVICE
                    CAL   SEARCH        /SEARCH TABLE & PERFORM COMMAND INPUTTED
                    JMP   NEXCMD        /REPEAT LOOP FOR NEXT COMMAND BY OPERATOR
                                        /
                                        /FETCH INPUT COMMAND STRING
            INCTRL, LHI   003           /SET PAGE ADDR OF START OF CHAR BUFFER
                    LLI   371           /SET LOC ON PG OF START OF BUFF (N+1)
```

```
                MNEMONIC                COMMENTS
                ----------              ----------

                LBI 006         /SET CNTR FOR MAX # USABLE CHARACTERS
INCHAR,         CAL INPUT       /CALL SUBROUTINE TO INPUT CHARACTER FM I/O
                CPI 215         /SEE IF CHAR WAS A "C-R"
                JFZ CHECK       /IF NOT, CHECK FOR BUFFER OVERFLOW
                LMA             /IF YES, STORE "C-R" AS LAST CHAR IN BUFF
                RET             /AND RETURN TO CALLING ROUTINE
CHECK,          INB             /EXERCISE REGISTER B (CNTR) TO SET FLAGS
                DCB             /ACCORDING TO ORIGINAL CONTENTS
                JTZ INCHAR      /IGNORE NEW CHARACTER IF CNTR WAS 000
                DCB             /OTHERWISE DECREMENT VALUE OF CNTR
                LMA             /AND STORE CHARACTER IN BUFFER
                CAL ADV         /ADVANCE BUFFER POINTER
                JMP INCHAR      /AND LOOP TO FETCH NEXT CHARACTER FROM I/O
                                /
                                /TABLE SEARCH ROUTINE
SEARCH,         LDI 002         /SET POINTERS TO STARTING ADDR OF TABLE
                LEI 000         / "     "     "     "     "   "    "
INITBF,         LHI 003         /SET POINTERS TO START OF CHAR BUFFER
                LLI 371         / "     "     "     "   "   "     "
CMATCH,         LAM             /GET CHAR FM BUFFER (FORM CHAR MATCH LOOP)
                CPI 215         /SEE IF SYMBOL FOR "C-R"
                JTZ LCHAR       /IF SO, GO TO LAST CHARACTER ROUTINE
                CAL ADV         /OTHERWISE ADVANCE BUFFER POINTER
                CAL SWITCH      /EXCHANGE BUFFER POINTER FOR TABLE PNTR
                CPM             /SEE IF HAVE MATCH CONDITION IN TABLE
                JFZ NXWORD      /IF NOT, GO TO NEXT BLOCK IN TABLE
                CAL ADV         /IF YES, ADVANCE TABLE POINTER
                CAL SWITCH      /EXCHANGE TABLE PNTR FOR BUFFER POINTER
                JMP CMATCH      /LOOP TO TEST NEXT CHARACTER
LCHAR,          CAL SWITCH      /EXCHANGE BUFFER POINTER FOR TABLE PNTR
                LAM             /TEST FOR END OF CONTROL FIELD
                NDI 300         /BY SEEING IF TWO MSB'S ARE BOTH "0"
                JTZ FOUNDW      /IF SO, HAVE FOUND MATCHING CONTROL WORD
NXWORD,         LAM             /TEST FOR END OF CONTROL FIELD
                NDI 300         /BY SEEING IF TWO MSB'S ARE BOTH "0"
                JTZ SETNXW      /IF SO, HAVE MARKER, GO TO NEXT BLOCK
                CAL ADV         /OTHERWISE, ADVANCE TABLE POINTER
                JMP NXWORD      /AND CONTINUE LOOKING
SETNXW,         CAL ADV         /AT END OF CONTROL WORD FIELD NEED TO
                CAL ADV         /ADVANCE PNTR OVER THE "ADDRESS" FIELD
                LAM             /AND THEN FETCH 1ST CHAR IN NEW BLOCK
                NDA             /SET THE FLAGS AFTER THE LOAD OPERATION
                RTZ             /RETURN IF END OF TABLE (NO MATCH FOUND)
                CAL SWITCH      /OTHERWISE EXCHANGE TABLE PNTR FOR BUFF
                JMP INITBF      /AND FORM LOOP TO CHECK NEXT BLOCK IN TBL
FOUNDW,         LDM             /EXTRACT PAGE ADDRESS AND STORE TEMP
                CAL ADV         /ADVANCE TABLE POINTER
                LEM             /STORE LOC ON PAGE TEMPORARILY
                LHI MMM         /NOW SET MEM PNTR (H & L) TO POINT TO THE
                LLI NNN         /2ND BYTE OF THE JUMP INSTR. COMING UP
                LMF             /PUT THE LOW ORDER ADDR IN BYTE 2
                INL             /ADVANCE THE MEMORY POINTER
                LMD             /AND THE PAGE ADDR IN BYTE 3 OF THE JMP
                JMP NNNMMM      /NOW JUMP TO THE ADDR JUST LOADED INTO
NNN             AAA             /THESE TWO (LOW ADDR)
MMM             BBB             /BYTES     (PAGE ADDR)
                                /
                                /AFTER PROCESSING CMND, RETURN TO MAIN RTN
```

ANOTHER PARTICULARLY POWERFUL CAPABILITY OF A MINI-COMPUTER IS ITS ABILITY TO RAPIDLY MANIPULATE AND ORGANIZE INFORMATION. A TYPICAL OPERATION IS TO SORT DATA INTO SOME DESIRED FORM SUCH AS TO ARRANGE A LIST OF NAMES INTO ALPHABETICAL ORDER, OR POSSIBLY TO ARRANGE A LIST OF ADDRESSES BY ZIP CODE ZONE NUMBERS.

THE KEY INGREDIENT IN DEVELOPING A PROGRAM TO PERFORM SORTING OPERATIONS IS TO PLAN THE ORGANIZATION OF THE STORAGE OF THE DATA IN MEMORY SO THAT THE OPERATING PORTION OF THE PROGRAM IS RELATIVELY SIMPLE. A SIMPLE TECHNIQUE INVOLVES JUSTIFYING THE DATA INTO FIELDS SO THAT SIMPLE COMPARING ALGORITHMS CAN BE UTILIZED.

AN AN EXAMPLE OF A SORTING PROGRAM, ASSUME ONE HAD A LIST OF NAMES THAT ONE WISHED TO HAVE THE COMPUTER PLACE IN ALPHABETICAL ORDER. A HYPOTHETICAL LIST MIGHT CONSIST OF THE FOLLOWING NAMES:

JONES, R. M.
SMITH, C.
WILLIAMS, P. K.
DAVIS, Z. T.
THOMPSON, A. R.
THOMAS, F.
ALLISON, A. B.
SMITH, T. P.

IT CAN BE SUPPOSED THAT THE NAMES WILL BE INPUTTED AND STORED IN THE COMPUTER IN THE ORDER SHOWN ABOVE. THE FIRST OBJECTIVE OF THE PROGRAM WOULD BE TO HAVE THE INCOMING NAMES BE STORED IN A MANNER THAT WOULD BE EASY FOR THE SORT ROUTINE TO OPERATE ON. A GOOD TECHNIQUE TO USE WOULD BE TO SET UP "FIELDS" FOR THE INFORMATION BEING STORED. IN THIS CASE ONE WOULD WANT TO SET UP THREE FIELDS. ONE FOR THE LAST NAME, ONE FOR THE FIRST INITIAL, AND ONE FOR THE MIDDLE INITIAL. THE SIZE OF EACH FIELD WOULD NEED TO BE DETERMINED. FOR THE EXAMPLE LIST SHOWN ABOVE THE LONGEST LAST NAME ENCOUNTERED HAS EIGHT LETTERS SO THE FIELD FOR THE LAST NAMES MUST HAVE SPACE FOR AT LEAST EIGHT CHARACTERS, SINCE ONE COMPUTER "WORD" IN MEMORY WILL STORE THE CODE FOR ONE LETTER IN THE NAME. HOWEVER, IN ORDER TO MAKE THE PROGRAM BE MORE GENERAL PURPOSE, ONE COULD SELECT A LONGER FIELD LENGTH TO ALLOW LONGER NAMES TO BE STORED. FOR ILLUSTRATIVE PURPOSES, A LAST NAME FIELD OF 14 (DECIMAL) UNITS WILL BE PLANNED. (NOTE THAT THIS IS A PURELY ARBITRARY SELECTION.) THE FIELD LENGTH FOR EACH INITIAL WOULD ONLY HAVE TO BE 1 MEMORY WORD. THUS THE TOTAL LENGTH OF THE THREE FIELDS MAKING UP A "BLOCK" WOULD BE 16 (DECIMAL) OR 20 OCTAL MEMORY WORDS. NOTE THAT IN SELECTING THE FIELD LENGTHS FOR THIS EXAMPLE, SPACE WAS NOT INCLUDED FOR THE COMMA (,) SIGN AFTER THE LAST NAME, OR THE PERIODS (.) AFTER EACH INITIAL. THIS IS BECAUSE SINCE THESE SIGNS ARE REPITITIOUS ONE CAN SAVE VALUABLE MEMORY SPACE BY DELETING THESE MARKS DURING THE INPUT OPERATION, AND THEN SIMPLY ADD THEM BACK IN AT THE APPROPRIATE POINT WHEN THE DATA IS DISPLAYED BY THE OUTPUT DEVICE.

THE INPUT ROUTINE WOULD NEED TO ALWAYS START INSERTING CHARACTERS AT THE BEGINNING OF A FIELD AND THEN INSERT SPACES OR SOME SPECIAL CODE (SUCH AS A 000 WORD) IN ALL OF THE UNUSED MEMORY WORDS IN A FIELD SO THAT THE NAMES COULD BE CONSIDERED AS BEING "LEFT JUSTIFIED" IN EACH FIELD. THE REASON FOR THIS WILL BE MADE CLEAR SHORTLY.

THE FOLLOWING ROUTINE MIGHT BE USED TO ACCEPT INFORMATION FROM A KEYBOARD DEVICE AND STORE THE NAMES IN MEMORY IN THE DESIRED FORMAT.

```
                MNEMONIC                    COMMENTS
                ----------                  ----------
   ACCEPT, LHI 004          /INITIALIZE NAMES STORAGE AREA PNTR
           LLI 000          /TO START OF STORAGE AREA
   NOTFND, LAM              /NOW FETCH 1ST LOCATION IN A BLOCK
           NDA              /SET FLAGS AFTER LOAD OPERATION
           JTZ FNDEND       /AND TEST FOR END OF STORAGE AREA
           LAI 020          /IF NOT END, THEN ADVANCE POINTER
           ADL              /TO NEXT BLOCK BY ADDING 20 OCTAL
           LLA              /TO MEM PNTR ADDRESS & RESTORE PNTR
   CKPAGE, CTZ INCRH        /ADVANCE PAGE ADDR OF PNTR IF REQ'D
           LAI 010          /NOW TEST TO SEE IF STILL
           CPH              /IN STORAGE AREA (PAGES 04 - 07 OCTAL)
 *         JTZ TOMUCH       /OPTIONAL DISPLAY MSG IF STORAGE FILLED
           JMP NOTFND       /KEEP LOOKING FOR END OF STORAGE AREA
   FNDEND, LBI 016          /SETUP LAST NAMES FIELD COUNTER
           CAL INPUT        /AND FETCH A CHARACTER FROM INPUT RTN
           CPI 252          /CHECK FOR * CODE (FINISHED INDICATOR)
           JFZ NOTDON       /PROCEED IF NOT * CODE
           XRA              /IF * CODE, THEN PLACE A 000 WORD AT
           LMA              /START OF BLOCK AS AN ENDING MARKER
           RET              /AND EXIT ROUTINE
   NOTDON, CPI 215          /TEST FOR CARRIAGE-RETURN CODE
           JTZ FNDEND       /AND IGNORE IF 1ST CHAR IN FIELD
           CPI 256          /TEST FOR PERIOD (.) CODE
           JTZ FNDEND       /AND IGNORE IF 1ST CHAR IN FIELD
           CPI 254          /TEST FOR COMMA (,) CODE
           JTZ FNDEND       /AND IGNORE IF 1ST CHAR IN FIELD
           LMA              /IF NONE OF ABOVE, PUT CHAR IN FIELD
           DCB              /DECREMENT THE FIELD SIZE COUNTER
           INL              /ADVANCE THE STORAGE POINTER
** NEXTIN, CAL INPUT       /AND FETCH THE NEXT CHAR IN LAST NAME
           CPI 215          /TEST FOR CARRIAGE-RETURN
           JTZ HAVECR       /FINISHED BLOCK IF HAVE C-R HERE
           CPI 254          /TEST FOR COMMA
           JTZ HAVECM       /FINISHED LAST NAME FIELD IF HAVE COMMA
           LMA              /OTHERWISE PLACE CHAR IN LAST NAME FIELD
           INL              /ADVANCE THE STORAGE POINTER
           DCB              /DECREMENT LAST NAMES FIELD SIZE CNTR
           JTZ FULFLD       /AND SEE IF FIELD IS FILLED
           JMP NEXTIN       /IF NOT, GET NEXT CHARACTER IN LAST NAME
   HAVECR, XRA              /IF HAVE C-R, PUT A 000 IN MEM WORDS
           LMA              /FOR REST OF CURRENT BLOCK
           LAL              /FETCH MEMORY POINTER TO ACCUMULATOR
           NDI 017          /MASK OFF 4 MOST SIGNIFICANT BITS
           CPI 017          /TEST FOR END OF BLOCK
           JTZ NEXBLK       /PREPARE FOR NEXT BLOCK IF DONE
           INL              /OTHERWISE ADVANCE POINTER
           JMP HAVECR       /AND CONTINUE PUTTING 000 WORDS IN BLOCK
   HAVECM, XRA              /IF HAVE COMMA, PUT 000 WORDS IN REST
           LMA              /OF "LAST NAME" FIELD
           INL              /ADVANCE FIELD POINTER
           DCB              /DECREMENT "LAST NAMES" FIELD CNTR
           JTZ FULFLD       /GO PROCESS INITIALS WHEN DONE
           JMP HAVECM       /ELSE CONTINUE TO CLEAR REST OF FIELD
   NEXBLK, INL              /ADVANCE MEM PNTR TO START OF NEXT BLOCK
           JMP CKPAGE       /AND PREPARE FOR NEXT NAME ENTRY
** FULFLD, CAL INPUT       /GET CHARACTER FOR 1ST INITIAL OF NAME
           CPI 254          /TEST FOR COMMA
           JTZ FULFLD       /IGNORE COMMA AT THIS POINT
           CPI 215          /TEST FOR C-R
```

```
                JFZ  SAVIN1   /IF NOT C-R, STORE CHARACTER
                XRA          /BUT, IF C-R, PUT IN 000 WORD
                LMA          /FOR BOTH INITIAL FIELDS
                INL          /BY ABOVE INSTRUCTION, THEN ADVANCING PNTR
                JMP  SAVIN2  /AND THEN FOLLOWING THIS JUMP COMMAND
     SAVIN1,    LMA          /STORE 1ST INITIAL IN 1ST INITIAL FIELD
                INL          /THEN ADVANCE STORAGE POINTER
** INITF2,      CAL  INPUT   /LOOK FOR 2ND INITIAL
                CPI  256     /CHECK FOR PERIOD
                JTZ  INITF2  /IGNORE A PERIOD
                CPI  215     /TEST FOR C-R
                JFZ  SAVIN2  /IF NOT C-R THEN STORE 2ND INITIAL
                XRA          /BUT IF WAS C-R, PLACE 000 WORD IN MEM
     SAVIN2,    LMA          /STORE THE CHARACTER OR 000 SUBSTITUTE
                INL          /ADVANCE POINTER TO NEW BLOCK
                JMP  CKPAGE  /AND CONTINUE LOADING IN NAMES
     INCRH,     INH          /SUBROUTINE TO INCREMENT REGISTER "H"
                RET          /AND RETURN TO CALLING ROUTINE
```

THE ABOVE ROUTINE HAS A NUMBER OF SPECIAL FACTORS INCLUDED IN IT TO ILLUSTRATE CONSIDERATIONS THAT PROGRAMMERS MUST LEARN TO TAKE INTO ACCOUNT WHEN DEVELOPING SUCH PROGRAMS. SOME OF THESE FACTORS ARE POINTED OUT IN THE FOLLOWING DISCUSSION OF THE ABOVE ROUTINE.

THE FIRST FUNCTION THE ABOVE ROUTINE PERFORMS IS TO LOOK FOR THE "END" OF THE NAME STORAGE AREA. THIS IS DONE BY TESTING THE FIRST CHARACTER IN EACH "BLOCK" TO SEE IF IT CONTAINS A 000 WORD. AS SHOWN LATER IN THE ROUTINE, A 000 WORD WILL BE ENTERED AT THAT LOCATION WHENEVER THE OPERATOR HAS FINISHED ENTERING A SERIES OF NAMES THAT WILL BE SORTED. IT SHOULD BE NOTED THAT WHENEVER IT IS DESIRED TO "INITIALIZE" THE NAME STORAGE AREA SO THAT IT APPEARS TO THE PROGRAM THAT THE STORAGE AREA IS EMPTY, A SUBROUTINE THAT WILL PLACE A 000 WORD AT PAGE 04 LOCATION 000 SHOULD BE EXECUTED. (THAT SIMPLE SUBROUTINE IS NOT SHOWN ABOVE). THE ABOVE ROUTINE ALSO MAKES A TEST, EACH TIME THE MEMORY POINTER IS ADVANCED TO A NEW BLOCK, TO DETERMINE WHETHER THE POINTER IS STILL IN THE ALLOTED NAMES STORAGE AREA. FOR THIS EXAMPLE THE STORAGE AREA WAS PLANNED TO RESIDE IN LOCATIONS FROM PAGE 04 LOCATION 000 TO PAGE 07 LOCATION 377. SHOULD THE ROUTINE GO BEYOND THE DESIGNATED STORAGE AREA BEFORE AN END OF "TABLE" MARKER IS FOUND, THE ROUTINE WOULD JUMP TO A ROUTINE TERMED "TOMUCH" WHICH MIGHT PRINT OUT A MESSAGE TO THE OPERATOR INDICATING THAT THE STORAGE AREA WAS ALREADY FILLED WITH NAMES. (THAT ROUTINE IS NOT INCLUDED IN THE EXAMPLE ABOVE). THE REFERENCE TO THE ROUTINE "TOMUCH" IS NOTED BY AN ASTERISK IN THE ABOVE PROGRAM SOURCE LISTING.

WHEN THE ROUTINE HAS FOUND THE END OF THE NAMES STORAGE AREA, INDICATING WHERE ADDITIONAL INCOMING NAMES CAN BE STORED (PROVIDED THE STORAGE AREA HAS NOT BEEN EXHAUSTED) THE ROUTINE THEN PROCEEDS TO ACCEPT DATA FROM AN INPUT SUBROUTINE. THE FIRST CHARACTER ACCEPTED AT THE START OF A NEW NAME (BLOCK) IS TESTED TO SEE IF IT IS A SPECIAL CODE (AN ASTERISK IN THIS CASE) THAT THE OPERATOR WOULD USE TO SIGNIFY TO THE PROGRAM THAT ALL THE DESIRED NAMES HAD BEEN ENTERED. IF THIS CODE WAS RECEIVED THEN A 000 CODE WOULD BE PLACED IN THE FIRST MEMORY WORD FOR THE "BLOCK" FOR THE END OF "TABLE" MARKER AS MENTIONED ABOVE. THE ROUTINE WOULD THEN EXIT THE ABOVE ROUTINE.

IF THE FIRST CHARACTER IN A NEW BLOCK IS NOT THE SPECIAL "END" CODE, A CHECK IS MADE TO SEE IF IT IS A CARRIAGE-RETURN, COMMA, OR PERIOD SIGN. ANYONE OF THOSE CODES WOULD BE IGNORED AS THE FIRST CHARACTER IN A BLOCK FOR THE FOLLOWING REASONS. THE RECEIPT OF A CARRIAGE-RETURN OR COMMA WOULD OBVIOUSLY BE INVALID AT THIS POINT BECAUSE NO LETTERS FOR A NAME HAVE BEEN ENTERED AND THE ACCEPTANCE OF EITHER OF THOSE OPERATORS

WOULD CAUSE THE LAST NAME FIELD TO BE COMPLETELY FILLED WITH 000 WORDS - INCLUDING THE FIRST LOCATION. THIS ACTION WOULD RESULT IN AN EFFECTIVE END OF STORAGE AREA MARKER BEING PLACED AT THE LOCATION OF THE CURRENT BLOCK. THE RECEIPT OF A PERIOD SIGN WOULD MOST LIKELY BE THE PERIOD SIGN FROM THE LAST INITIAL FIELD ENTERED (WHICH IS TO BE IGNORED) AND CERTAINLY WOULD NOT BE A VALID LETTER FOR THE BEGINNING OF A LAST NAME. THE INCORPORATION OF THESE CHECKS ACT AS SAFEGUARDS FOR HUMAN OPERATOR ERRORS AND ARE ANOTHER EXAMPLE OF "HUMAN ENGINEERING" FACTORS IN THE DEVELOPMENT OF A PROGRAM.

IF THE FIRST CHARACTER IS NOT ONE OF THE ABOVE IT IS STORED IN THE FIRST LOCATION IN THE "LAST NAME FIELD." AFTER THE FIRST CHARACTER HAS BEEN STORED, EACH CHARACTER RECEIVED FROM THE INPUT ROUTINE IS TESTED TO SEE IF IT IS A CARRIAGE-RETURN OR COMMA. IF IT IS A COMMA, SIGNIFYING THE END OF THE "LAST NAME FIELD," ANY UNFILLED LOCATIONS IN THE FIELD ARE FILLED WITH ZEROS AND THE PROGRAM PROCEEDS TO THE "INITIAL" FIELDS. HOWEVER, IF A CARRIAGE-RETURN IS NOTED, THE PROGRAM FILLS THE ENTIRE REMAINDER OF THE CURRENT BLOCK, INCLUDING THE "INITIAL" FIELDS WITH ZERO WORDS AS A CARRIAGE-RETURN SIGNIFIES THE COMPLETION OF A NAME ENTRY. AN ADDITIONAL SAFEGUARD IS BUILT INTO THE ROUTINE IN THIS SECTION TO PREVENT TOO MANY CHARACTERS FROM BEING ENTERED INTO THE LAST NAME FIELD. WHEN THE FIELD HAS BEEN FILLED, THE POINTER IS NOT ADVANCED UNTIL A CARRIAGE-RETURN OR COMMA IS RECEIVED.

ONCE THE LAST NAME FIELD HAS BEEN PROCESSED, THE ROUTINE WILL ACCEPT ANY MORE CHARACTERS AS INITIALS, BUT IGNORES THE PERIOD SIGNS AFTER THE INITIALS. WHEN AN ENTIRE NAME HAS BEEN PROCESSED THE PROGRAM THEN LOOPS TO ACCEPT ANOTHER NAME BLOCK AFTER CHECKING TO MAKE SURE THE STORAGE AREA IS NOT FILLED AND REPEATS THE PROCESS DESCRIBED.

THE ABOVE ROUTINE COULD BE MODIFIED TO INCLUDE AN OPERATOR CONVENIENCE - THE ABILITY TO ERASE A CURRENT ENTRY IF THE OPERATOR MADE A MISTAKE WHILE TYPING IN A NAME. THIS COULD BE DONE BY EXECUTING A ROUTINE IMMEDIATELY AFTER THE POINTS DESIGNATED IN THE PROGRAM BY A DOUBLE ASTERISK (**). THE ROUTINE COULD BE USED TO CHECK FOR A SPECIAL "ERASE" CODE." IF THIS CODE WAS DETECTED, THE PROGRAM COULD RESET THE POINTERS TO THE START OF THE CURRENT NAME BLOCK AND ALLOW RE-ENTRY OF THE NAME. SUCH A ROUTINE MIGHT BE AS SHOWN HERE:

```
                MNEMONIC                    COMMENTS
            ----------------            ----------------

ERRORT,     CPI  377        /CHECK FOR A "RUBOUT" CODE
            JFZ  AWAY       /EXIT ROUTINE IF NOT A "RUBOUT"
            LAL             /IF HAVE A "RUBOUT" THEN FETCH POINTER
            NDI  360        /REMOVE 4 LEAST SIGNIFICANT BITS
            LLA             /AND RESTORE POINTER TO START OF BLOCK
            JMP  FNDEND     /JUMP TO RE-ENTER NAME
AWAY,       ***             /*** NEXT INSTRUCTION IN CURRENT SEQUENCE
```

WHILE THE PREVIOUS ROUTINE SEEMS A BIT LONG AT FIRST GLANCE, ONE MUST REMEMBER THAT IT IS DOING QUITE A FEW FUNCTIONS AND IS QUITE GENERAL PURPOSE IN OVER-ALL DESIGN. THE PROGRAM ALLOWS ONE TO BUILD UP A LIST OF NAMES IN A DESIGNATED AREA OF MEMORY, PLACING THE DATA IN FORMATTED FIELDS, CHECKS FOR SELECTED OPERATOR ERRORS, AND BOUNDS OR LIMITS THE STORAGE AREA. THE PROGRAM, USING THE BASIC CONCEPTS PRESENTED, CAN BE MODIFIED TO SERVE AS A BASIC STRUCTURE FOR INPUTTING A VARIETY OF TYPES OF DATA INTO JUSTIFIED FIELDS OF DATA. TO PROVIDE A CLEAR MENTAL PICTURE OF HOW THE LIST OF NAMES GIVEN SEVERAL PAGES EARLIER WOULD APPEAR WHEN INPUTTED TO MEMORY USING THE PROGRAM ILLUSTRATED, A DIAGRAM

SHOWING MEMORY LOCATIONS AND THEIR CONTENTS IS PROVIDED BELOW SHOWING HOW THE DATA WOULD LOOK WHEN ORGANIZED BY THE ABOVE PROGRAM. THE DIAGRAM SHOWS ADDRESSES (ON PAGE 04) WITH THE CONTENTS OF THE MEMORY LOCATION SHOWN BENEATH IT, FOLLOWED BY THE ALPHABETICAL REPRESENTATION FOR THE CODE WHERE APPLICABLE.

```
ADDR:  000 001 002 003 004 005 006 007 010 011 012 013 014 015 016 017
CONT:  312 317 316 305 323 000 000 000 000 000 000 000 000 000 322 315
LETR:   J   O   N   F   S   -   -   -   -   -   -   -   -   -   R   M

ADDR:  020 021 022 023 024 025 026 027 030 031 032 033 034 035 036 037
CONT:  323 315 311 324 310 000 000 000 000 000 000 000 000 000 303 000
LETR:   S   M   I   T   H   -   -   -   -   -   -   -   -   -   C   -

ADDR:  040 041 042 043 044 045 046 047 050 051 052 053 054 055 056 057
CONT:  327 311 314 314 311 301 315 323 000 000 000 000 000 000 320 313
LETR:   W   I   L   L   I   A   M   S   -   -   -   -   -   -   P   K

ADDR:  060 061 062 063 064 065 066 067 070 071 072 073 074 075 076 077
CONT:  304 301 326 311 323 000 000 000 000 000 000 000 000 000 332 324
LETR:   D   A   V   I   S   -   -   -   -   -   -   -   -   -   Z   T

ADDR:  100 101 102 103 104 105 106 107 110 111 112 113 114 115 116 117
CONT:  324 310 317 315 320 323 317 316 000 000 000 000 000 000 301 322
LETR:   T   H   O   M   P   S   O   N   -   -   -   -   -   -   A   R

ADDR:  120 121 122 123 124 125 126 127 130 131 132 133 134 135 136 137
CONT:  324 310 317 315 301 323 000 000 000 000 000 000 000 000 306 000
LETR:   T   H   O   M   A   S   -   -   -   -   -   -   -   -   F   -

ADDR:  140 141 142 143 144 145 146 147 150 151 152 153 154 155 156 157
CONT:  301 314 314 311 323 317 316 000 000 000 000 000 000 000 301 302
LETR:   A   L   L   I   S   O   N   -   -   -   -   -   -   -   A   B

ADDR:  160 161 162 163 164 165 166 167 170 171 172 173 174 175 176 177
CONT:  323 315 311 324 310 000 000 000 000 000 000 000 000 000 324 320
LETR:   S   M   I   T   H   -   -   -   -   -   -   -   -   -   T   P

ADDR:  200 201 202 203 204 205 206 207 210 211 212 213 214 215 216 217
CONT:  000 *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
LETR:   -  .......DON'T CARE ABOUT MEMORY CONTENTS BEYOND HERE.......
```

ONCE THE DATA HAS BEEN ORGANIZED IN A SUITABLE MANNER IN MEMORY, ONE CAN PROCEED TO DEVELOP A RELATIVELY SIMPLE SORT ROUTINE TO ARRANGE THE NAMES IN ALPHABETICAL ORDER. THE TECHNIQUE TO BE ILLUSTRATED CONSIST OF COMPARING THE LETTERS, STARTING WITH THE LEFT-MOST POSITION IN A BLOCK (AS SEEN IN THE MEMORY DIAGRAM ABOVE) AGAINST THE LETTER IN THE SAME POSITION IN THE NEXT BLOCK IN MEMORY. BY "LETTER" WHAT IS ACTUALLY MEANT IS THE ASCII CODE (IN THIS EXAMPLE) FOR A LETTER. IT SO HAPPENS THAT THE ASCII CODE IS ARRANGED SUCH THAT THE ALPHABET GOES IN AN ASCENDING NUMERICAL ORDER. THE LETTER A IS REPRESENTED AS 301, THE LETTER B AS 302, C AS 303, AND SO FORTH ON UP TO THE LETTER Z WHICH HAS AN OCTAL REPRESENTATION OF 332. HOW CONVENIENT! THIS MEANS THAT IF THE VALUE IN A MEMORY WORD (REPRESENTING A LETTER IN ASCII FORMAT) IS COMPARED AGAINST ANOTHER MEMORY WORD CONTAINING AN ASCII CODED LETTER, THAT THE LOWER VALUE LOCATION CONTAINS A LOWER ORDER LETTER IN THE ALPHABET.

WITH THIS INFORMATION ONE CAN QUICKLY DISCERN THAT ONE CAN QUITE EASILY DEVELOP AN ALGORITHM TO ARRANGE NAMES ALPHABETICALLY. IF THE

VALUE OF MEMORY LOCATION IN THE FIRST POSITION OF SAY THE FIRST BLOCK (THE NTH BLOCK) IS COMPARED AGAINST THE VALUE OF THE FIRST POSITION IN THE NEXT BLOCK (N+1 BLOCK) AND FOUND TO BE GREATER IN VALUE, THAN THE FIRST (NTH) BLOCK HAS A NAME THAT IS HIGHER ALPHABETICALLY THAN THE NAME IN THE SECOND (N+1) BLOCK AND THUS ONE CAN IMMEDIATELY PROCEED TO EXCHANGE THE CONTENTS OF THE TWO BLOCKS TO ARRANGE THE NAMES IN ASCENDING ALPHABETICAL ORDER. IF, HOWEVER, THE CODE IN THE FIRST BLOCK IS LESS IN VALUE THAN THE SECOND BLOCK, THEN THE PRESENT ORDER IS CORRECT AND THE PROGRAM CAN PROCEED TO CHECK THE SECOND BLOCK AGAINST THE THIRD ONE. IF THE LETTERS IN THE FIRST POSITION CHECKED ARE EQUAL IN VALUE, THEN ONE CANNOT YET MAKE A DECISION ABOUT THE ALPHABETICAL ORDER, BUT RATHER MUST GO ON TO COMPARE THE VALUES OF THE SECOND LETTER WITHIN THE TWO BLOCKS!

TO FURTHER COMPLETE THE ALGORITHM ONE MUST ALSO CONSIDER THE POSSIBILITY THAT WHEN ONE EXCHANGES THE CONTENTS OF BLOCKS "N" AND "N+1" THAT THE NEW CONTENTS OF "N" WILL NOW BE OF LESSER ORDER THAN THAT CONTAINED IN BLOCK "N-1." THUS, WHENEVER ONE PERFORMS AN EXCHANGE OF TWO BLOCKS ONE MUST HAVE THE PROGRAM GO BACK AND DO A COMPARISON BETWEEN THE "N" AND "N-1" BLOCKS. ONE CAN ENVISION THE ALGORITHM AS PROCEEDING IN A "SEE-SAW" MANNER - COMPARING THE "NTH" BLOCK AGAINST THE "N+1" BLOCK UNTIL AN EXCHANGE IS NECESSARY, THEN SWITCHING TO COMPARE BETWEEN THE "NTH" AND "N-1" BLOCK UNTIL AN EXCHANGE IS NOT NECESSARY. AT THAT POINT THE PROCESS REVERTS BACK TO COMPARING THE "NTH" AND "N+1" BLOCKS UNTIL ANOTHER EXCHANGE IS REQUIRED. LOOKED AT ANOTHER WAY, THE DATA BLOCKS COULD BE VIEWED AS "RIPPLING" UPWARDS OR DOWNWARDS IN MEMORY AS THE PROCESS PROCEEDS. HIGHER ORDERED NAMES GETTING SHOVED TO HIGHER ADDRESSED BLOCKS, LOWER ORDERED NAMES BEING PUSHED TO LOWER ADDRESSED BLOCKS.

THIS TYPE OF ALGORITHM IS NOT THE ONLY WAY ONE COULD PROCEED TO SORT THE DATA. THERE ARE OTHER TYPES OF ALGORITHMS THAT CAN PERFORM THE SAME JOB, SOME OF WHICH ARE FASTER WHEN LARGE DATA BASES ARE INVOLVED (BUT MORE COMPLICATED PROGRAMMING-WISE). SUCH ALGORITHMS GENERALLY HAVE CONSIDERABLE VALUE ON LARGE MACHINES. HOWEVER, THE ABOVE ALGORITHM IS QUITE SUITABLE FOR TYPICAL SORTING JOBS THAT A 8008 UNIT MIGHT BE CALLED UPON TO PERFORM. FOR THOSE WHO MIGHT WANT TO INVESTIGATE OTHER ALGORITHMS THEY MIGHT CONSIDER THE CONCEPT OF HAVING A PROGRAM THAT IMMEDIATELY CLASSIFIES A NAME INTO, SAY, THE FIRST, SECOND, OR THIRD SECTION OF THE ALPHABET.

A PROGRAM FOR THE "RIPPLE" SORTING ALGORITHM DISCUSSED ABOVE IS PRESENTED BELOW.

```
                MNEMONIC                    COMMENTS
                ----------                  ----------

SORT,    LHI 004        /INITIALIZE POINTER TO START
         LLI 000        /OF NAMES BLOCK STORAGE AREA
INITBK,  LBI 020        /SET BLOCK LENGTH COUNTER
         LCM            /GET 1ST CHAR FM BLOCK "N" INTO "C" REGIS
         LAL            /FETCH "N" BLOCK POINTER
         ADI 020        /ADVANCE POINTER TO BLOCK "N+1"
         LLA            /RESTORE POINTER
         CPI 020        /CHECK TO SEE IF GOING TO NEW PAGE
         CTS INCRH      /ADVANCE PAGE PNTR IF REQUIRED
         LAM            /GET 1ST CHAR FM BLOCK "N+1" INTO ACC
         NDA            /SET FLAGS AFTER LOADING OPERATION
         RTZ            /END OF STORAGE - SORT OPS COMPLETED
         CPC            /COMPARE "N+1" LETTER TO "N" LETTER
```

MNEMONIC		COMMENTS
	JTS XCHANG	/"N" > "N+1" SO EXCHANGE BLOCK CONTENTS
	JTZ CKNEXT	/"N" = "N+1" SO CHECK NEXT LETTER IN BLOCK
	JMP INITBK	/"N" < "N+1" SO ORDER O.K., DO NEXT BLOCK
CKNEXT,	DCB	/DECREMENT BLOCK LENGTH COUNTER
	JFZ NOTFIN	/CONTINUE IF NOT FINISHED BLOCK
BACKER,	SUI 017	/PNTR FOR LAST OF "N+1" BECOMES 1ST OF "N"
	JMP INITBK	/BACK TO COMPARE NEXT BLOCK
NOTFIN,	LAL	/FETCH "N+1" BLOCK POINTER
	NDA	/CLEAR THE CARRY FLAG WITH THIS "NO-OP"
	SUI 017	/DECREASE POINTER TO "N" BLOCK
	LLA	/RESTORE POINTER
	CTC DECRH	/IF UNDERFLOW THEN DECREMENT PAGE POINTER
	LCM	/FETCH CHARACTER FROM "N" BLOCK TO REG "C"
	LAL	/FETCH "N" BLOCK POINTER
	ADI 020	/INCREASE POINTER TO "N+1" BLOCK
	LLA	/RESTORE POINTER
	CPI 020	/CHECK TO SEE IF GOING TO NEW PAGE
	CTS INCRH	/ADVANCE PAGE PNTR IF REQUIRED
	LAM	/GET CHARACTER FROM "N+1" BLOCK
	CPC	/COMPARE "N+1" LETTER TO "N" LETTER
	JTS XCHANG	/"N" > "N+1" SO EXCHANGE BLOCK CONTENTS
	JTZ CKNEXT	/"N" = "N+1" SO CHECK NEXT LETTER IN BLOCK
FINEND,	DCB	/"N" < "N+1" SO ORDER O.K., DO NEXT BLOCK
	JTZ BACKER	/AT END OF BLOCK "N+1" RESET PNTR FOR "N"
	INL	/ADVANCE POINTER
	JMP FINEND	/AND LOOP TO LOOK FOR END OF BLOCK
XCHANG,	LAL	/FETCH "N+1" POINTER
	NDI 360	/MASK OFF LSB'S TO RESTORE POINTER
	LLA	/TO START OF "N+1" BLOCK
	LBI 020	/SET BLOCK LENGTH COUNTER
NOTYET,	LCM	/FETCH "N+1" INTO REGISTER "C"
	LAL	/FETCH "N+1" POINTER TO ACCUMULATOR
	NDA	/CLEAR THE CARRY FLAG
	SUI 020	/DECREASE POINTER TO "N" BLOCK
	LLA	/RESTORE POINTER
	CTC DECRH	/DECREMENT PAGE POINTER IF REQUIRED
	LDM	/FETCH "N" INTO REGISTER "D"
	LMC	/PLACE FORMER "N+1" INTO "N"
	LAL	/FETCH "N" POINTER TO ACCUMULATOR
	ADI 020	/INCREASE POINTER TO "N+1" BLOCK
	LLA	/RESTORE POINTER
	CPI 020	/CHECK TO SEE IF GOING TO NEW PAGE
	CTS INCRH	/INCREMENT PAGE POINTER IF REQUIRED
	LMD	/PLACE FORMER "N" INTO "N+1"
	INL	/ADVANCE "N+1" POINTER
	DCB	/DECREMENT BLOCK LENGTH COUNTER
	JFZ NOTYET	/CONTINUE IF NOT FINISHED EXCHANGING
	LAL	/IF FINISHED EXCHANGING FETCH "N+1" PNTR
	NDA	/CLEAR CARRY FLAG
	SUI 040	/BACK POINTER FROM "N+1" TO "N-1" BLOCK
	LLA	/RESTORE POINTER
	CTC DECRH	/DECREMENT PAGE POINTER IF REQUIRED
	LAH	/FETCH CURRENT PAGE
	CPI 003	/MAKE SURE STILL IN STORAGE AREA
	JFZ INITBK	/YES - DO AN EFFECTIVE "N-1" TO "N" TEST
	JMP SORT	/WENT BACK TOO FAR - GO TO STARTING BLOCK!

THE "INCRH" REFERRED TO BY THE SORT ROUTINE WAS PRESENTED EARLIER AS PART OF THE ROUTINE THAT ACCEPTED NAMES INTO THE STORAGE AREA. THE "DECRH" ROUTINE NOT SHOWN SHOULD BE A SNAP FOR ANYONE WHO HAS REACHED THIS POINT IN THE MANUAL. (IF IT IS NOT, FOR HEAVENS SAKE GO BACK!)

IF ONE MENTALLY PROCEEDS THROUGH THE SORT ROUTINE WHILE REFERRING TO THE DIAGRAM GIVEN SEVERAL PAGES EARLIER SHOWING THE NAMES AS ORIGINALLY STORED IN MEMORY, ONE SHOULD BE ABLE TO CLEARLY DISCERN THE OPERATION OF THE SORT PROGRAM. FOR EXAMPLE, FOR THE FIRST THREE NAMES THE PROGRAM ENCOUNTERS IN THE ORIGINAL EXAMPLE SETUP, THE PROGRAM WILL ONLY HAVE TO TEST THE FIRST LETTER IN EACH BLOCK. WHEN THE NAME IN THE 4TH BLOCK IS EXAMINED, AN EXCHANGE WILL HAVE TO MADE WITH THE NAME IN THE THIRD BLOCK, THEN THE PROGRAM WILL FIND WHEN CHECKING THE "N-1" BLOCK (WHICH WAS THE ORIGINAL SECOND BLOCK) THAT THE NAME "DAVIS, Z. T." HAS TO BE EXCHANGED AGAIN, AND THIS WILL HAPPEN ONE MORE TIME UNTIL THE NAME "DAVIS, Z.T." ARRIVES AT THE FIRST BLOCK IN THE STORAGE AREA. AT THIS POINT THE PROGRAM GOES BACK TO CHECKING AGAINST THE "N+1" BLOCK. THE NAMES WOULD NOW APPEAR IN MEMORY IN THE FOLLOWING ORDER.

```
BLOCK #1:  DAVIS, Z. T.
BLOCK #2:  JONES, R. M.
BLOCK #3:  SMITH, C.
BLOCK #4:  WILLIAMS, P. K.
BLOCK #5:  THOMPSON, A. R.
BLOCK #6:  THOMAS, F.
BLOCK #7:  ALLISON, A. B.
BLOCK #8:  SMITH, T. P.
```

NOW THE PROGRAM WOULD GET DOWN TO BLOCK FIVE BEFORE IT FOUND IT NECESSARY TO EXCHANGE BLOCK FIVE WITH BLOCK FOUR. THE NEXT "N-1" TEST WOULD FAIL, HOWEVER, AND THE PROGRAM WOULD PROCEED BACK UP TO BLOCK SIX WHERE IT WOULD FIND THE NAME "THOMAS, F." AND HAVE TO EXCHANGE IT WITH "WILLIAMS, P. K." AND THEN EXCHANGE IT AGAIN WITH "THOMPSON, A. R." AT THIS POINT THE NAMES STORAGE AREA WOULD APPEAR AS:

```
BLOCK #1:  DAVIS, Z. T.
BLOCK #2:  JONES, R. M.
BLOCK #3:  SMITH, C.
BLOCK #4:  THOMAS, F.
BLOCK #5:  THOMPSON, A. R.
BLOCK #6:  WILLIAMS, P. K.
BLOCK #7:  ALLISON, A. B.
BLOCK #8:  SMITH, T. P.
```

AT THIS POINT THE PROGRAM WOULD GET UP TO BLOCK NUMBER SEVEN WHERE IT WOULD FIND "ALLISON, A. B." AND IT WOULD THEN HAVE TO EXCHANGE NAMES ALL THE WAY BACK DOWN THE LINE TO GET IT INTO BLOCK NUMBER ONE. FINALLY, THE PROGRAM WOULD FIND THAT "SMITH, T. P." HAD TO BE MOVED BACK ENDING UP IN BLOCK NUMBER FIVE. ALL OF THE ABOVE WOULD HAVE HAPPENED IN A MERE FRACTION OF A SECOND AS THE 8008 CPU EXECUTED THE INSTRUCTIONS AT MICRO-SECOND SPEEDS - RESULTING IN THE NAMES ORGANIZED IN THE FOLLOWING DESIRED MANNER.

```
BLOCK #1:  ALLISON, A. B.
BLOCK #2:  DAVIS, Z. T.
BLOCK #3:  JONES, R. M.
BLOCK #4:  SMITH, C.
BLOCK #5:  SMITH, T. P.
BLOCK #6:  THOMAS, F.
BLOCK #7:  THOMPSON, A. R.
BLOCK #8:  WILLIAMS, P. K.
```

SIMILAR TYPES OF SORTING OR ARRANGING OPERATIONS CAN ALSO BE DONE WITH NUMBERS IN EITHER ASCII, BCD, OR BINARY FORM OR WITH OTHER TYPES OF DATA.

ONE COULD COMBINE A "CONTROL TABLE" USING ONE OF THE TYPES DISCUSSED EARLIER IN THIS CHAPTER WITH THE NECESSARY INPUT, FORMATTING, AND SORT SUBROUTINE ADDRESSES STORED IN THE TABLE, AND THUS MAKE UP A POWERFUL YET EASY TO USE PROGRAM PACKAGE SUITED TO THE USER'S SPECIFIC REQUIREMENTS.

BY UTILIZING THE CONCEPTS (AS WELL AS POSSIBLY SOME OF THE SPECIFIC ROUTINES) PRESENTED IN THIS SECTION, THE READER SHOULD BE ABLE TO SEE THE WAY TOWARDS DEVELOPING SOPHISTICATED PROGRAMS CAPABLE OF PERFORMING FUNCTIONS TAILORED TO THE INDIVIDUAL'S OWN REQUIREMENTS.

MORE INFORMATION ON HANDLING I/O ROUTINES WILL BE PRESENTED IN A LATER CHAPTER. FOR THOSE INTERESTED IN UTILIZING THE MATHEMATICAL CAPABILITIES OF THE DIGITAL COMPUTER (PERHAPS COMBINING SUCH OPERATIONS WITH SOME OF THOSE JUST DISCUSSED) SIMPLY PROCEED ON TO STUDY THE NEXT CHAPTER WHICH IS DEVOTED TO JUST THAT SUBJECT!

MATHEMATICAL OPERATIONS

THE ABILITY OF A DIGITAL COMPUTER TO BE ABLE TO HANDLE MATHEMATICAL OPERATIONS COUPLED WITH IT'S ABILITY TO MANIPULATE TEXT GIVES THE MACHINE A UNIQUE COMBINATION OF FUNCTIONALITY THAT ACCOUNTS FOR IT'S GROWING POPULARITY. PROGRAMMING A COMPUTER USING MACHINE LANGUAGE TO PERFORM MATHEMATICAL FUNCTIONS IS PERHAPS A BIT MORE COMPLICATED THAN HAVING IT PERFORM ROUTINE TEXT MANIPULATIONS, BUT IT IS NOT AS DIFFICULT AS SOME PEOPLE TEND TO THINK BEFORE BEING INTRODUCED TO THE SUBJECT. LIKE MOST OTHER PROGRAMMING TASKS, THE KEY TO SUCCESS IS ORGANIZATION OF THE PROGRAM INTO SMALL ROUTINES THAT CAN BE BUILT UPON TO FORM MORE POWERFUL COMBINATIONS.

THE INSTRUCTION SET OF THE 8008 CPU CONTAINS A NUMBER OF PRIMARY MATHEMATICAL INSTRUCTIONS THAT ARE THE BASIS FOR DEVELOPING MATHEMATICAL PROGRAMS. THE GROUPS USED MOST OFTEN INCLUDE THE ADDITION, SUBTRACTION AND "ROTATE" INSTRUCTIONS. (DO YOU RECALL THAT ROTATING A BINARY NUMBER TO THE LEFT EFFECTIVELY DOUBLES, OR MULTIPLIES THE ORIGINAL VALUE BY TWO, AND ROTATING IT TO THE RIGHT ESSENTIALLY DIVIDES THE ORIGINAL VALUE IN HALF?)

DEALING WITH NUMBERS OF SMALL MAGNITUDE USING A 8008 CPU IS SIMPLICITY ITSELF. FOR INSTANCE, IF ONE WANTED TO ADD, SAY THE NUMBERS 2 AND 7, ONE COULD LOAD ONE NUMBER INTO REGISTER "B" IN THE CPU AND LOAD THE OTHER INTO THE ACCUMULATOR. THE SIMPLE DIRECTIVE:

```
ADB
```

WOULD RESULT IN THE VALUE 011 (OCTAL!) BEING LEFT IN THE ACCUMULATOR. SUBTRACTION IS JUST AS EASY. IF ONE PLACED 7 IN THE ACCUMULATOR AND 2 IN REGISTER "B" AND EXECUTED A:

```
SUB
```

THE VALUE 5 WOULD BE LEFT IN THE ACCUMULATOR.

MULTIPLICATION, WITH SMALL NUMBERS, CAN BE READILY ACCOMPLISHED USING A SIMPLE ALGORITHM OF ADDING THE MULTIPLICAND TO ITSELF THE NUMBER OF TIMES DICTATED BY THE MULTIPLIER. SUPPOSE ONE DESIRED TO HAVE THE COMPUTER MULTIPLY 2 TIMES 3. PLACING THE VALUE 2 IN REGISTER "B" AND 3 IN REGISTER "C" AND EXECUTING THE FOLLOWING INSTRUCTION SEQUENCE:

```
START, XRA
MULTIP, ADB
        DCC
        JFZ MULTIP
 STOP, HLT
```

WOULD RESULT IN THE VALUE 6 ENDING UP IN THE ACCUMULATOR. AS SHALL BE DISCUSSED FURTHER ON, THE ABOVE ALGORITHM IS NOT VERY EFFICIENT WHEN THE NUMBERS BECOME LARGE. MORE EFFICIENT MULTIPLICATION ALGORITHMS ARE BASED ON ROTATE OPERATIONS WHICH EFFECTIVELY MULTIPLY A NUMBER BY A POWER OF TWO. FOR INSTANCE, MULTIPLYING A NUMBER BY 32 (DECIMAL) WOULD REQUIRE 32 (DECIMAL) LOOPS THROUGH THE ABOVE ROUTINE, BUT ONLY 5 ROTATE LEFT OPERATIONS! HOWEVER, THE ABOVE ROUTINE ILLUSTRATES HOW A NUMBER CAN BE MULTIPLIED EVEN THOUGH THE COMPUTER DOES NOT HAVE A SPECIFIC "MULTIPLY" INSTRUCTION.

ONE CAN ALSO DIVIDE SMALL VALUED NUMBERS THAT HAVE INTEGER RESULTS USING A SIMILARLY SIMPLE ALGORITHM THAT SUBTRACTS INSTEAD OF ADDS. FOR

INSTANCE, A REVERSE OF THE PREVIOUS EXAMPLE WOULD BE TO DIVIDE THE NUMBER 6 BY THE VALUE 2. THE SUBTRACTION ALGORITHM WOULD APPEAR AS:

```
START, LCI 000
DIVIDE, NDA
        JTZ STOP
        SUB
        INC
        JMP DIVIDE
 STOP, HLT
```

IN THE ABOVE ALGORITHM, THE ROUTINE STARTS WITH THE NUMBER 6 IN THE ACCUMULATOR. THE DIVISOR IS IN REGISTER "B." REGISTER "C" IS USED AS A COUNTER TO COUNT HOW MANY TIMES THE VALUE IN "B" CAN BE SUBTRACTED UNTIL THE CONTENTS OF THE ACCUMULATOR IS EQUAL TO ZERO. AS POINTED OUT PREVIOUSLY, THE ALGORITHM ONLY WORKS IF THE RESULT IS AN INTEGER VALUE. DIVISION IS PERHAPS THE MOST DIFFICULT BASIC MATHEMATICAL FUNCTION TO PERFORM ON A DIGITAL COMPUTER BECAUSE OF MATHEMATICAL PECULIARITIES (INVOLVING THE MANIPULATION OF FRACTIONAL VALUES). HOWEVER, AS WILL BE ILLUSTRATED LATER, THERE ARE WAYS AROUND THE ABOVE LIMITATION. THE ABOVE ILLUSTRATION IS MERELY TO GIVE THE NOVICE ENCOURAGEMENT BY ILUSTRATING THAT SUCH OPERATIONS ARE POSSIBLE EVEN THOUGH A SPECIFIC "DIVIDE" COMMAND IS NOT A PART OF THE TYPICAL DIGITAL COMPUTER'S INSTRUCTION SET!

THE DISCUSSION SO FAR HAS BEEN LIMITED TO NUMBERS OF RELATIVELY SMALL MAGNITUDE. SPECIFICALLY, NUMBERS SMALL ENOUGH TO BE CONTAINED IN A SINGLE EIGHT BIT BINARY REGISTER OR MEMORY LOCATION IN A 8008 UNIT. MANY USER'S WHO WANT TO USE THE DIGITAL COMPUTER TO PERFORM MATHEMATICAL OPERATIONS SEEM TO GET "STUMPED" WHEN FIRST COMING ACROSS A REQUIREMENT TO MANIPULATE NUMBERS THAT ARE TOO LARGE IN MAGNITUDE TO FIT IN ONE MEMORY WORD OR CPU REGISTER. WITH A 8008 BASED MACHINE, AND INDEED MOST MINI-COMPUTERS, SUCH A REQUIREMENT TYPICALLY ARRIVES SHORTLY AFTER ONE HAS STARTED OPERATING THEIR MACHINE! THE REASON IS SIMPLY THAT THE LARGEST VALUED NUMBER THAT CAN BE PLACED IN AN "N-BIT" REGISTER IS THE VALUE (2↑N)-1. SINCE THE 8008 CPU USES BUT 8 (DECIMAL) BITS IN A WORD, THE LARGEST NUMBER THAT CAN BE REPRESENTED IN A SINGLE WORD IF ALL THE BITS ARE USED IS A MERE 255 (DECIMAL). IF ONE DESIRES TO MAINTAIN THE "SIGN" (WHETHER IT IS "PLUS" OR "MINUS") AND USES ONE BIT IN A WORD FOR THAT PURPOSE, THEN THE LARGEST NUMBER THAT CAN BE REPRESENTED IN A SINGLE WORD IS A PALTRY 127 (DECIMAL) - HARDLY ENOUGH TO BOTHER USING A COMPUTER TO MANIPULATE SUCH LIMITED MAGNITUDES!

BUT, THE SECRET TO RAPIDLY INCREASING THE MAGNITUDES OF THE NUMBERS THAT CAN BE HANDLED BY A DIGITAL COMPUTER IS HELD IN THAT FORMULA JUST PRESENTED - (2↑N)-1. FOR THAT FORMULA SAYS THAT THE SIZE OF THE NUMBER THAT CAN BE STORED IN A BINARY REGISTER ESSENTIALLY DOUBLES FOR EVERY BIT ADDED TO THE REGISTER. THUS, IF ONE WERE TO STORE A NUMBER USING THE AVAILABLE BITS IN TWO REGISTERS OR MEMORY WORDS IN A 8008 SYSTEM, ONE WOULD BE ABLE TO REPRESENT NUMBERS AS LARGE AS (2↑16)-1 OR 65,535 (DECIMAL). IF ONE OF THOSE 16 BITS WERE RESERVED FOR A "SIGN" INDICATOR THE MAGNITUDE WOULD BE LIMITED TO (2↑15)-1 OR 32,767. THAT IS CERTAINLY A LOT MORE THAN THE VALUE OF 127 THAT CAN BE HELD IN JUST ONE WORD! BUT, WHY STOP AT HOLDING A NUMBER IN TWO WORDS? THERE IS NO NEED TO, ONE CAN KEEP ADDING WORDS TO BUILD UP AS MANY BITS AS DESIRED. THREE WORDS OF 8 BITS, LEAVING ONE BIT OUT FOR A SIGN INDICATOR WOULD ALLOW NUMBERS UP TO (2↑23)-1 OR 8,388,607 (DECIMAL). FOUR WORDS, WOULD ALLOW REPRESENTING A SIGNED NUMBER UP TO (2↑31)-1 WHICH IS APPROXIMATELY 1,107,483,647! ONE COULD ADD STILL MORE WORDS IF REQUIRED. GENERALLY, HOWEVER, ONE SELECTS THE NUMBER OF "SIGNIFICANT DIGITS" THAT WILL BE IMPORTANT IN THE CALCULATIONS TO BE PERFORMED AND USES ENOUGH WORDS TO

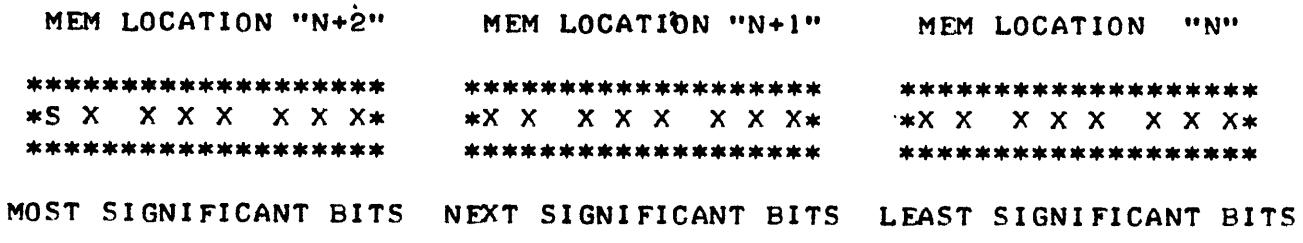
ENSURE THAT THE "PRECISION," OR NUMBER OF SIGNIFICANT DIGITS REQUIRED FOR THE OPERATIONS CAN BE REPRESENTED IN THE TOTAL NUMBER OF BITS AVAILABLE WITHIN THE "GROUPED" WORDS. THE USE OF MORE THAN ONE COMPUTER WORD OR REGISTER TO STORE AND MANIPULATE NUMBERS AS THOUGH THEY WERE IN ONE LARGE CONTINUOUS REGISTER IS COMMONLY REFERRED TO AS "MULTIPLE-PRECISION" ARITHMETIC. ONE OFTEN HEARS COMPUTER TECHNOLOGISTS SPEAKING OF "DOUBLE-PRECISION" OR "TRIPLE-PRECISION" ARITHMETIC. THIS SIMPLY MEANS THAT THE MACHINE IS USING TECHNIQUES (GENERALLY PROGRAMMING TECHNIQUES) THAT ENABLE IT TO HANDLE NUMBERS STORED IN TWO OR THREE REGISTERS AS THOUGH THEY WERE ONE NUMBER IN A VERY LARGE REGISTER.

THE 8008 CPU IS CAPABLE OF MULTIPLE-PRECISION ARITHMETIC. IN FACT IT DOES IT QUITE NICELY BECAUSE THE DESIGNERS OF THE INTEL 8008 CPU CHIP TOOK PARTICULAR CARE TO INCLUDE SOME SPECIAL INSTRUCTIONS FOR JUST SUCH OPERATIONS. (SUCH AS THE ADD AND SUBTRACT WITH CARRY INSTRUCTIONS.) MULTIPLE-PRECISION ARITHMETIC IS NOT DIFFICULT - IT TAKES A LITTLE EXTRA CONSIDERATION IN THE AREA OF ORGANIZING THE PROGRAM TO HANDLE AND STORE NUMBERS THAT ARE CONTAINED IN MULTIPLE WORDS IN MEMORY, BUT WITH THE USE OF EFFECTIVE "SUBROUTINING" OR SO CALLED "CHAINING" OPERATIONS THE TASK MAY BE HANDLED WITH RELATIVE EASE.

IN ORDER TO EFFECTIVELY DEAL WITH MULTIPLE-PRECISION ARITHMETIC ONE MUST ESTABLISH A CONVENTION FOR STORING THE SECTIONS OF ONE LARGE NUMBER IN SEVERAL REGISTERS. FOR THE PURPOSES OF THE CURRENT DISCUSSION, IT WILL BE ASSUMED THAT "TRIPLE-PRECISION" ARITHMETIC IS TO BE PERFORMED. NUMBERS WILL BE STORED IN THREE CONSECUTIVE MEMORY LOCATIONS ACCORDING TO THE FOLLOWING ARRANGEMENT.

MEMORY LOCATION "N"	=	LEAST SIGNIFICANT 8 BITS
MEMORY LOCATION "N+1"	=	NEXT SIGNIFICANT 8 BITS
MEMORY LOCATION "N+2"	=	MOST SIGNIFICANT 7 BITS + SIGN BIT

THUS, THE THREE WORDS IN MEMORY COULD BE MENTALLY VIEWED AS BEING ONE CONTINUOUS LARGE REGISTER CONTAINING 23 BINARY BITS PLUS A SIGN BIT AS SHOWN IN THE DIAGRAM BELOW.



OF COURSE, ONE COULD REVERSE THE ABOVE SEQUENCE, AND STORE THE LEAST SIGNIFICANT BITS IN MEMORY LOCATION "N," THE NEXT GROUP IN "N+1," AND THE MOST SIGNIFICANT BITS PLUS SIGN BIT IN MEMORY LOCATION "N+2." IT MAKES LITTLE DIFFERENCE AS LONG AS ONE REMAINS CONSISTENT WITHIN A PROGRAM. HOWEVER, THE CONVENTION ILLUSTRATED WILL BE THE ONE USED FOR THE DISCUSSION IN THIS SECTION.

ALSO, AS HAS BEEN POINTED OUT, IT IS NOT NECESSARY TO LIMIT THE STORAGE TO JUST THREE WORDS - ADDITIONAL WORDS MAY BE USED IF ADDITIONAL PRECISION IS REQUIRED. FOR MOST OF THE DISCUSSION IN THIS CHAPTER, THREE WORDS WILL BE USED FOR STORING NUMBERS. USING THREE WORDS IN THE ABOVE FASHION WILL ALLOW NUMBERS UP TO A VALUE OF 8,388,647 IN MAGNITUDE TO BE STORED. THIS MEANS THAT 6 TO 7 SIGNIFICANT DIGITS CAN BE MAINTAINED IN CALCULATIONS.

THE FIRST MULTIPLE-PRECISION ROUTINE TO BE ILLUSTRATED WILL BE AN ADDITION ROUTINE THAT WILL ADD TOGETHER TWO MULTIPLE-PRECISON NUMBERS AND LEAVE THE RESULT IN THE LOCATION FORMERLY OCCUPIED BY ONE OF THE NUMBERS. THE ROUTINE TO BE PRESENTED HAS BEEN DEVELOPED AS A "GENERAL PURPOSE" ROUTINE IN THAT, BY PROPERLY SETTING UP MEMORY ADDRESS POINTERS AND LOADING A CPU REGISTER WITH A "PRECISION" VALUE PRIOR TO "CALLING" THE ROUTINE, THE SAME ROUTINE CAN BE USED TO HANDLE MULTIPLE-PRECISION ADDITION OF NUMBERS VARYING IN LENGTH FROM "1 TO N" REGISTERS (AS LONG AS THE REGISTERS CONTAINING A NUMBER ARE IN CONSECUTIVE ORDER IN MEMORY, AND WITH THE RESTRICTION THAT ALL THE REGISTERS CONTAINING A NUMBER ARE ON ONE PAGE - LIMITING "N" TO 255 (DECIMAL WORDS), WHICH IS A LIMITATION FEW PROGRAMMERS WOULD FIND CUMBERSOME)!

THE KEY ELEMENT IN THE ADDITION ROUTINE TO BE ILLUSTRATED IS THE USE OF THE "ACM," OR "ADD WITH CARRY" INSTRUCTION. THE ESSENTIAL DIFFERENCE BETWEEN AN "ADD WITH CARRY" (ACM) INSTRUCTION, AND AN "ADM" (ADD WITHOUT CARRY) COMMAND IS AS FOLLOWS:

AN "ADM" INSTRUCTION SIMPLY ADDS THE CONTENTS OF THE ACCUMULATOR AND THE CONTENTS OF THE MEMORY LOCATION POINTED TO BY THE "H & L" REGISTERS. DURING THE ADDITION PROCESS, THE STATUS OF THE CARRY FLAG IS IGNORED. HOWEVER, IF AT THE END OF THE PROCESS, AN "OVER-FLOW" HAS OCCURED, THE CARRY FLAG WILL BE SET TO A "1" CONDITION FOR EXAMPLE, ADDING THE FOLLOWING BINARY NUMBERS WOULD YIELD:

```
                  1 0  1 0 1  0 1 0
                  0 1  0 1 0  1 0 1
                 -------------------
CARRY = 0 :       1 1  1 1 1  1 1 1
```

AND ADDING THE NEXT TWO NUMBERS WOULD YIELD:

```
                  1 1  1 1 1  1 1 1
                  0 0  0 0 0  0 0 1
                 -------------------
CARRY = 1 :       0 0  0 0 0  0 0 0
```

REGARDLESS OF THE CONDITION OF THE CARRY FLAG AT THE START OF THE ADDITION OPERATION.

AN "ACM" COMMAND, ON THE OTHER HAND, EXAMINES THE CONTENTS OF THE CARRY FLAG PRIOR TO THE START OF THE ADDITION OPERATION AND CONSIDERS IT AS AN OPERATOR ON THE LEAST SIGNIFICANT BIT POSITION. AT THE END OF THE PROCESS, THE CARRY FLAG IS SET OR CLEARED DE-PENDING ON WHETHER OR NOT AN "OVERFLOW" OCCURED, AS IN THE "ADM" CLASS OF INSTRUCTION. FOR EXAMPLE, ADDING THE FOLLOWING BINARY NUMBERS YIELDS RESULTS THAT ARE DEPENDENT ON THE INITIAL STATUS OF THE CARRY FLAG.

```
  CASE #1A        1 0  1 0 1  0 1 0  : 0 = CARRY BIT AT START
                  0 1  0 1 0  1 0 1
                 -------------------
CARRY = 0 :       1 1  1 1 1  1 1 1


  CASE #1B        1 0  1 0 1  0 1 0  : 1 = CARRY BIT AT START
                  0 1  0 1 0  1 0 1
                 -------------------
CARRY = 1 :       0 0  0 0 0  0 0 0
```

```
CASE #2A    1 1   1 1 1   1 1 1   : 0 = CARRY BIT AT START
            0 0   0 0 0   0 0 1
           ------------------
CARRY = 1 : 0 0   0 0 0   0 0 0

CASE #2B    1 1   1 1 1   1 1 1   : 1 = CARRY BIT AT START
            0 0   0 0 0   0 0 1
           ------------------
CARRY = 1 : 0 0   0 0 0   0 0 1
```

IN SUMMARY, ONE CAN SEE THAT AN "ACM" TYPE OF INSTRUCTION MAKES MULTIPLE-PRECISION ADDITION EXTREMELY EASY BECAUSE THE CARRY BIT ACTS AS A LINK BETWEEN ANY "CARRY" FROM THE MOST SIGNIFICANT BIT OF ONE ADDITION OPERATION INTO THE LEAST SIGNIFICANT BIT OF THE NEXT ADDITION OPERATION - JUST AS THOUGH THE ADDITION PROCESS WAS PERFORMED IN ONE LONG REGISTER. FOR COMPARISON, EXAMINE THE EXAMPLE BELOW WHICH FIRST ILLUSTRATES AN ADDITION OPERATION IN A HYPOTHETICAL 16 (DECIMAL) BIT REGISTER, AND THEN SHOWS THE SAME RESULT WHEN TWO "ACM" OPERATIONS ARE PERFORMED ON TWO 8 BIT REGISTERS.

```
HYPOTHETICAL 16 BIT REGISTER:  1 1   1 1 1   1 1 1    1 0   1 0 1   0 1 0
                               0 0   0 0 0   0 0 0    1 1   0 1 0   1 0 1
                              ---------------------------------------------
                    CARRY = 1 : 0 0   0 0 0   0 0 0    0 1   1 1 1   1 1 1

FIRST ACM OPERATION:  1 0   1 0 1   0 1 0   : 0 = ASSUMED FOR CARRY
                      1 1   0 1 0   1 0 1         BIT AT START
                     ----------------
           CARRY = 1 : 0 1   1 1 1   1 1 1   = LSB'S IN MEM LOC "N"

                      1 1   1 1 1   1 1 1   : 1 = CARRY STATUS FROM
                      0 0   0 0 0   0 0 0         OPERATION ABOVE
                     ----------------
           CARRY = 1 : 0 0   0 0 0   0 0 0   =  MSB'S IN MEM LOC "N+1"
```

PLACING THE RESULTS OF THE TWO 8 BIT REGISTERS SIDE-BY-SIDE AFTER USING THE "ACM" TYPE OF INSTRUCTION YIELDS THE SAME RESULT AS THOUGH THE OPERATION HAD BEEN PERFORMED IN A SIXTEEN BIT REGISTER. THE CONCEPT CAN BE APPLIED TO AS MANY 8 BIT REGISTERS AS NECESSARY!

ARMED WITH THE KNOWLEDGE OF HOW THE POWERFUL "ACM" TYPE OF INSTRUCTION OPERATES, ONE CAN PROCEED TO DEVELOP A "N'TH PRECISION" ADDITION SUBROUTINE. EXAMINE THE FOLLOWING ROUTINE.

```
          MNEMONIC                    COMMENTS
          --------                    --------

ADDER,    NDA            /ALWAYS CLEAR CARRY FLAG AT RTN ENTRY
ADDMOR,   LAM     1      /GET FIRST NUMBER INTO ACCUMULATOR
          CAL SWITCH     /CHANGE POINTERS TO SECOND NUMBER
          ACM            /PERFORM "ADDITION WITH CARRY"
          LMA            /PLACE RESULT BACK INTO MEMORY
          DCB            /DECREMENT THE "PRECISION" COUNTER
          RTZ            /EXIT ROUTINE WHEN COUNTER REACHES 000
```

```
MNEMONIC                COMMENTS
----------              ----------

        INL             /ADVANCE SECOND NUMBER POINTER
        CAL SWITCH      /CHANGE POINTER BACK TO FIRST NUMBER
        INL             /ADVANCE FIRST NUMBER POINTER
        JMP ADDMOR      /REPEAT PROCESS FOR NEXT PRECISION
```

NOTE THAT THE ABOVE "ADDER" SUBROUTINE REQUIRES THAT A NUMBER OF THE CPU REGISTERS BE "SET UP" PRIOR TO CALLING THE ROUTINE. THE "H & L" REGISTERS MUST CONTAIN THE ADDRESS OF THE LEAST SIGNIFICANT BITS IN THE FIRST MULTI-WORD NUMBER. REGISTERS "D & E" SIMILARLY MUST BE SET UP TO CONTAIN THE ADDRESS OF THE LEAST SIGNIFICANT PART OF THE SECOND MULTI-PRECISION NUMBER THAT IS TO BE ADDED TO THE FIRST. FINALLY, REGISTER "B" MUST BE INITIALIZED TO THE "PRECISION," OR NUMBER OF MEMORY WORDS USED TO CONTAIN THE MULTI-PRECISION NUMBER. SUPPOSE, FOR EXAMPLE, THAT A NUMBER IN "TRIPLE-PRECISION" FORMAT IS STORED IN THREE WORDS STARTING AT LOCATION 100 ON PAGE 00 AND THAT A SECOND NUMBER IN SIMILAR FORMAT IS STORED AT LOCATION 200 ON PAGE 01. THE FOLLOWING INSTRUCTIONS WOULD BE USED TO SET UP THE CPU REGISTERS PRIOR TO CALLING THE "ADDER" SUBROUTINE.

```
        MNEMONIC                COMMENTS
        ----------              ----------

INIT,   LHI 000         /SET PAGE FOR LSW OF FIRST NUMBER
        LLI 100         /SET LOCATION ON PAGE FOR LSW OF 1ST #
        LDI 001         /SET PAGE FOR LSW OF SECOND NUMBER
        LEI 200         /SET LOCATION ON PAGE FOR LSW OF 2ND #
        LBI 003         /SET PRECISION VALUE (3 WORDS)
        CAL ADDER       /CALL THE N'TH PRECISION ADDITION RTN
          .
          .             /USER ROUTINES TO PROCESS THE ANSWER
          .
```

NOTE TOO, THAT THE "ADDER" SUBROUTINE IS "DESTRUCTIVE" TO THE ORIGINAL VALUE OF THE SECOND NUMBER THAT IS ADDED BECAUSE THE ANSWER IS LEFT IN THAT LOCATION. IF, FOR SOME REASON, THE USER WANTED TO SAVE THE ORIGINAL SECOND NUMBER, THEN IT WOULD HAVE TO BE "SAVED" ELSEWHERE IN MEMORY PRIOR TO PERFORMING THE ADDITION.

JUST AS THERE ARE TWO CLASSES OF INSTRUCTIONS FOR PERFORMING ADDITION WITH THE 8008 CPU, ONE OF WHICH (ACM CATEGORY) IS SUITED FOR MULTIPLE-PRECISION ARITHMETIC, THERE ARE TWO CLASSES OF SUBTRACT COMMANDS. THE "SUM" (SUBTRACT WITHOUT CARRY) AND THE "SBM" (SUBTRACT WITH CARRY - OR MORE APPROPRIATELY "BORROW"). THE "SBM" TYPE WORKS SIMILAR TO THE "ACM" TYPE IN THAT THE CPU FIRST CHECKS THE STATUS OF THE CARRY FLAG BEFOR PERFORMING THE SUBTRACTION OPERATION MAKING IT AN EASY MATTER TO PROCESS MULTIPLE-PRECISION SUBTRACTION OPERATIONS. IN FACT, ONE CAN SET UP AN ALMOST IDENTICAL ROUTINE TO THE ONE USED FOR ADDITION THAT WILL ALLOW PROCESSING "N'TH PRECISION" SUBTRACTION OPERATIONS. AS IN THE PREVIOUS EXAMPLE, ONE WOULD FIRST SET UP CPU REGISTERS AS POINTERS TO THE LEAST SIGNIFICANT PORTIONS OF THE MULTIPLE-PRECISION NUMBERS IN MEMORY AND LOAD REGISTER "B" WITH THE NUMBER OF MEMORY WORDS OCCUPIED BY A "N'TH PRECISION" NUMBER.

WHILE THE ROUTINES PRESENTED HERE ONLY UTILIZE THE "ACM" OR "SBM" INSTRUCTIONS - BECAUSE THE ALGORITHMS HAVE BEEN DEVELOPED AS GENERAL

PURPOSE ROUTINES TO HANDLE STRINGS OF NUMBERS IN MEMORY, THE READER IS REMINDED THAT THERE ARE A WHOLE GROUP OF INSTRUCTIONS THAT HAVE SIMILAR CAPABILITY FOR WORKING WITH DATA IN CPU REGISTERS (SUCH AS "ACB," "ACC," AND THE OTHER CPU REGISTERS PLUS "IMMEDIATE" OPERATIONS). THE READER SHOULD REVIEW CHAPTER ONE OF THIS 8008 PROGRAMERS MANUAL FOR A SUMMARY OF THE POSSIBLE VARIATIONS.

```
                MNEMONIC                    COMMENTS
            --------------              --------------

SUBBER, NDA             /ALWAYS CLEAR CARRY FLAG AT START OF RTN
SUBTRA, LAM             /GET FIRST NUMBER INTO ACCUMULATOR
        CAL SWITCH      /CHANGE POINTERS TO SECOND NUMBER
        SBM             /SUBTRACT 2'ND FROM 1'ST WITH BORROW
        LMA             /PLACE RESULT BACK INTO MEMORY
        DCB             /DECREMENT THE PRECISION COUNTER
        RTZ             /EXIT ROUTINE WHEN COUNTER = 000
        INL             /ADVANCE SECOND NUMBER POINTER
        CAL SWITCH      /CHANGE POINTER BACK TO FIRST NUMBER
        INL             /ADVANCE FIRST NUMBER POINTER
        JMP SUBTRA      /REPEAT PROCESS FOR NEXT PRECISION
```

ONE THING A USER DEALING WITH MATHEMATICAL FUNCTIONS ON A COMPUTER WILL SOON HAVE TO BE CONCERNED WITH IS WHAT HAPPENS WHEN A LARGER NUMBER IS SUBTRACTED FROM A SMALLER NUMBER. THE ANSWER IS NATURALLY A MINUS OR NEGATIVE NUMBER. AS WAS INITIALLY DISCUSSED IN THE CHAPTER ON FUNDAMENTAL PROGRAMMING SKILLS, THE 8008 CPU PROCESSES NEGATIVE NUMBERS UTILIZING THE "TWO'S COMPLEMENT" CONVENTION. THE READER MAY WANT TO REVIEW THE FIRST FEW PAGES OF THAT SECTION AT THIS TIME.

FOR INSTANCE, IF USING SINGLE PRECISION ARITHMETIC, THE NUMBER 8 (DECIMAL) WAS SUBTRACTED FROM 6, THE RESULT WOULD APPEAR IN THE ACCUMULATOR AS SHOWN HERE:

```
6 DECIMAL  =  0 0  0 0 0  1 1 0  IN A BINARY REGISTER
8 DECIMAL  =  0 0  0 0 1  0 0 0  IN A BINARY REGISTER
              -----------------
WHICH IS   =  1 1  1 1 1  1 1 0  WHEN SUBTRACTED
```

NOTE THAT THE MOST SIGNIFICANT BIT IN THE REGISTER CONTAINING THE MINUS ANSWER IS A "1." BY ESTABLISHING A TWO'S COMPLEMENT CONVENTION AND ALWAYS ENSURING THAT THE MAGNITUDE OF ANY NUMBERS HANDLED DO NOT INTERFERE WITH THE MOST SIGNIFICANT BIT, ONE CAN QUICKLY DETERMINE WHETHER A NUMBER IN A REGISTER (OR SERIES OF REGISTERS IN THE CASE OF MULTIPLE-PRECISION FORMATTING) IS POSITIVE OR NEGATIVE BY TESTING TO SEE IF THE MOST SIGNIFICANT BIT IS A 1 (FOR A NEGATIVE) OR 0 (FOR A POSITIVE) VALUE. THIS IS READILY DONE IN A 8008 CPU BY TESTING THE "SIGN" FLAG WITH A "JFS," "CTS" OR SIMILAR INSTRUCTION.

ALSO REMEMBER THAT A NUMBER CAN BE SUBTRACTED FROM ANOTHER NUMBER BY FORMING THE TWO'S COMPLEMENT OF THE NUMBER TO BE SUBTRACTED AND PERFORMING AN ADDITION OPERATION. THUS:

```
    + 8 DECIMAL  =  0 0  0 0 1  0 0 0  IN A BINARY REGISTER

IT'S TWO'S COMPLEMENT IS  =  1 1  1 1 1  0 0 0  IN A BINARY REGISTER
```

AND CONSEQUENTLY:

```
          6 DECIMAL  =  0 0  0 0 0  1 1 0  IN A BINARY REGISTER
TWO'S COMPLEMENT OF 8  =  1 1  1 1 1  0 0 0  IN A BINARY REGISTER
                          ------------------
          WHICH IS   =  1 1  1 1 1  1 1 0  WHEN ADDED!
```

IT IS OFTEN DESIRABLE TO PERFORM A STRAIGHT "TWO'S COMPLEMENT" OPERATION ON A NUMBER IN ORDER TO CHANGE IT FROM A POSITIVE TO A NEGATIVE NUMBER OR THE REVERSE. ONE EASY WAY TO ACCOMPLISH THIS IN A 8008 UNIT IS TO SIMPLY SUBTRACT THE NUMBER FROM A VALUE OF ZERO. FOR MULTIPLE-PRECISION WORK ONE COULD SIMPLY LOAD ONE STRING OF MEMORY LOCATIONS (THE FIRST NUMBER) WITH ZEROS AND PLACE THE NUMBER TO BE NEGATED IN THE SECOND STRING OF MEMORY LOCATIONS (THE SECOND NUMBER) AND CALL THE PREVIOUSLY ILLUSTRATED "SUBBER" ROUTINE. HOWEVER, THERE MAY BE CASES WHERE ONE DOES NOT WANT TO DISTURB VALUES IN MEMORY LOCATIONS OR PERFORM THE TRANSFER OPERATIONS NECESSARY TO SET UP THE NUMBERS FOR THE "SUBBER" ROUTINE. WHAT IS DESIRED IS A "TWO'S COMPLEMENT" ROUTINE THAT WILL OPERATE ON A VALUE IN THE LOCATION(S) IN WHICH IT RESIDES. THE FOLLOWING ROUTINE WILL ACCOMPLISH THAT OBJECTIVE, AND CAN HANDLE "N'TH PRECISION" NUMBERS.

```
          MNEMONIC                    COMMENTS
          --------                    --------

COMPLM,   LAM          /GET LEAST SIGNIFICANT BITS (1ST WORD)
          XRI 377      /EXCLUSIVE "OR" = PURE COMPLEMENT
          ADI 001      /NOW ADD 1 TO FORM TWO'S COMPLEMENT
MORCOM,   LMA          /RETURN 2'S COMPLEMENT VALUE TO MEMORY
          RAR          /GET THE CARRY BIT INTO THE ACCUMULATOR
          LDA          /AND SAVE THE CARRY BIT STATUS
          DCB          /NOW DECREMENT THE "PRECISION" COUNTER
          RTZ          /FINISHED WHEN COUNTER = 000
          INL          /IF NOT DONE, ADVANCE MEMORY POINTER
          LAM          /AND FETCH THE NEXT GROUP OF BITS
          XRI 377      /PRODUCE A PURE COMPLEMENT
          LEA          /SAVE PURE COMPLEMENT TEMPORARILY
          LAD          /GET PREVIOUS CARRY BACK INTO ACCUMULATOR
          RAL          /AND SHIFT IT BACK OUT TO THE CARRY FLAG
          LAI 000      /DO A LOAD SO DOES NOT DISTURB CARRY
          ACE          /ADD COMPLEMENTED VALUE WITH ANY CARRY
          JMP MORCOM   /GO ON TO DO NEXT WORD IN STRING
```

NOTICE THAT IN THE ABOVE ROUTINE IT WAS NECESSARY TO SAVE THE STATUS OF THE CARRY FLAG (BIT) IN A CPU REGISTER BECAUSE AN "XRI" OR ANY BOOLEAN LOGIC INSTRUCTION AUTOMATICALLY "CLEARS" THE CARRY FLAG TO ZERO AND WOULD DESTROY ANY PREVIOUS "1" CONDITION. (ANY READERS WHO FORGOT THAT MIGHT BE WISE TO SPEND A LITTLE MORE TIME STUDYING CHAPTER ONE OF THIS MANUAL!

AS WITH THE "ADDER" AND "SUBBER" ROUTINES IT IS ALSO NECESSARY TO DO SOME PRELIMINARY SETTING UP BEFORE CALLING THE "COMPLM" SUBROUTINE. THE "H & L" REGISTERS MUST BE SET TO THE FIRST WORD (LEAST SIGNIFICANT BITS) OF THE MULTI-PRECISION NUMBER AND REGISTER "B" MUST INDICATE HOW MANY WORDS ARE OCCUPIED BY THE NUMBER.

IT WILL ALSO BE POINTED OUT HERE, THAT AS THE PROGRAMMER GETS INTO

DEVELOPING MORE AND MORE COMPLICATED ROUTINES, THAT UTILIZE A LOT OF SUBROUTINES, THE PROGRAMMER MUST MAINTAIN STRICT CONTROL OVER WHICH CPU REGISTERS ARE AFFECTED AND MAKE SURE THAT THE USE OF SELECTED CPU REGISTERS BY ONE ROUTINE (ESPECIALLY WHEN IT "CALLS" ANOTHER ROUTINE) DO NOT INTERFERE WITH THE OVER-ALL OPERATION OF A PROGRAM. THE BEST RULE OF THUMB IS TO TRY AND LEAVE A SUBROUTINE WITH ALL THE CPU REGISTERS, EXCEPT THOSE TRANSFERRING INFORMATION TO THE NEXT ROUTINE, IN A "FREE" OR "DON'T CARE" STATE. THIS IS NOT ALWAYS POSSIBLE, AND WHEN IT IS NOT, THE PROGRAMMER MUST KEEP TRACK OF WHICH REGISTERS ARE BEING USED FOR A SPECIFIC PURPOSE AND NOT ALLOW THEM TO BE UNINTENTIONALLY ALTERED. FOR INSTANCE, THE ABOVE "COMPLM" ROUTINE REQUIRES THAT THREE OF THE CPU REGISTERS BE SET UP PRIOR TO ENTRY - THE "H," "L," AND "B" REGISTERS. WHEN IT LEAVES THE ROUTINE THOSE ROUTINES ARE ESSENTIALLY FREE FOR USE BY THE NEXT ROUTINE. IT ALSO USES THE "A," "D" AND "E" CPU REGISTERS FOR OPERATIONS THAT IT PERFORMS. IT DOES NOT CARE ABOUT THE STATUS OF THOSE REGISTERS WHEN IT STARTS OPERATIONS BECAUSE IT "LOADS" THEM ITSELF. IT ALSO LEAVES THOSE REGISTERS ESSENTIALLY "FREE" WHEN THE ROUTINE IS EXITED. (ALL THE IMPORTANT OPERATIONS ARE DONE WITH LOCATIONS IN MEMORY). HOWEVER, THE FACT THAT THE ROUTINE USES CERTAIN CPU REGISTERS - SUCH AS REGISTERS "D & E," IS VERY IMPORTANT TO REMEMBER IF ONE WAS USING OTHER ROUTINES THAT MAINTAINED, SAY, MEMORY POINTERS IN REGISTERS "D & E." THE NOVICE PROGRAMMER (AND A LOT OF TIMES THE "NOT-SO-NOVICE" ONES) WILL OFTEN FIND SOME VERY STRANGE OPERATIONS OCCURING IN A NEWLY DEVELOPED PROGRAM BECAUSE OF PROBLEMS RELATED TO JUST THIS ASPECT!

THE ABOVE ROUTINES COULD BE USED BY THEMSELVES TO HANDLE ADDITION AND SUBTRACTION OF LARGE NUMBERS. HOWEVER, A RESTRICTION ON THE TYPES OF NUMBERS THEY COULD HANDLE WOULD BE THAT THE NUMBERS WOULD HAVE TO BE WHOLE NUMBERS. ALSO, AS THE MAGNITUDES OF THE NUMBERS TO BE HANDLED INCREASED, THE NUMBER OF WORDS USED TO STORE A VALUE IN MULTI-PRECISION FORMAT WOULD HAVE TO BE INCREASED. AS WAS POINTED OUT EARLIER, USUALLY, WHEN ONE STARTS DEALING WITH NUMBERS OF LARGE MAGNITUDE, ONE IS PRIMARILY CONCERNED WITH A CERTAIN NUMBER OF "SIGNIFICANT" DIGITS IN A CALCULATION. FOR INSTANCE, ONE COULD REPRESENT THE VALUE ONE MILLION AS 1,000,000. TO STORE THIS NUMBER IN MULTI-PRECISION FORMAT REQUIRES THE USE OF THREE MEMORY WORDS IN A 8008 UNIT. HOWEVER, THE NUMBER 1,000,000 ONLY CONTAINS ONE SIGNIFICANT DIGIT. THE NUMBER COULD JUST AS EASILY BE REPRESENTED AS 1 RAISED TO THE 6TH POWER OF 10, OR 1 E+6 IN WHAT IS OFTEN TERMED FLOATING POINT FORMAT. NOTE THAT IF THE NUMBER WAS STORED IN SUCH A FORMAT, ONE WOULD ONLY NEED TO USE ONE MEMORY REGISTER (FOR THE "1" AS THE SIGNIFICANT DIGIT, AND ANOTHER REGISTER TO HOLD THE POWER TO WHICH THE SIGNIFICANT DIGIT WAS TO BE RAISED. FLOATING POINT FORMAT ALSO ENABLES ANOTHER PROBLEM TO BE READILY HANDLED - THAT OF PROCESSING FRACTIONAL NUMBERS. UP TO THIS POINT, NO DISCUSSION ON REPRESENTING NON-INTEGER NUMBERS HAS BEEN PRESENTED. THIS WILL BE DONE SHORTLY, HOWEVER, AS AN INTRODUCTION, NOTE THAT THE DECIMAL NUMBER 0.1 COULD BE REPRESENTED IN FLOATING POINT FORMAT AS 1 RAISED TO THE MINUS 1 POWER OF 10, OR 1 E-1.

THE READER HAS NOW BEEN INTRODUCED TO MULTI-PRECISION ARITHMETIC AND HOPEFULLY HAS AN UNDERSTANDING OF HOW LARGE NUMBERS CAN BE STORED IN SEVERAL SMALL REGISTERS. THE TERM LARGE NUMBERS CAN BE INTERPRETED AS NUMBERS CONTAINING MORE THAN A COUPLE OF SIGNIFICANT DIGITS. THE READER SHOULD UNDERSTAND THAT INCREASING THE NUMBER OF SIGNIFICANT DIGITS REQUIRES AN INCREASE IN THE NUMBER OF BINARY BITS NEEDED TO STORE A NUMBER AND HENCE INCREASES THE NUMBER OF MEMORY WORDS REQUIRED WHEN THE NUMBER IS STORED IN MULTI-PRECISION FORMAT. ALSO, WHEN THE FORMAT DESCRIBED UP TO NOW IS USED, INCREASING THE MAGNITUDE OF A NUMBER (BY ADDING ZEROS TO THE RIGHT OF THE SIGNIFICANT DIGITS) RAPIDLY INCREASES THE NUMBER OF WORDS OF MEMORY REQUIRED TO HOLD A NUMBER. FINALLY, JUST STORING A NUMBER IN A REGISTER, WITHOUT REGARD TO A "DECIMAL POINT" LOCATION, MAKES

IT IMPOSSIBLE TO PROPERLY MANIPULATE FRACTIONAL NUMBERS.

HOWEVER, THE IDEA THAT NUMBERS CAN BE REPRESENTED AS A SERIES OF SIGNIFICANT DIGITS RAISED TO A POWER PRESENTS A SOLUTION TO THE LIMITATIONS MENTIONED. HANDLING NUMBERS IN SUCH A FASHION IS GENERALLY TERMED "FLOATING-POINT" ARITHMETIC. THE REMAINDER OF THIS CHAPTER WILL BE DEVOTED TO DEVELOPING ROUTINES FOR A "FLOATING-POINT" MATHEMATICAL PROGRAM FOR GENERAL PURPOSE APPLICATIONS.

HOWEVER, BEFORE PROCEEDING INTO THE DEVELOPMENT OF FLOATING-POINT ROUTINES, IT WILL BE NECESSARY TO DISCUSS A MATTER THAT HAS BEEN LEFT ASIDE UP TO THIS POINT - REPRESENTING FRACTIONAL NUMBERS UTILIZING THE LANGUAGE OF THE DIGITAL COMPUTER - BINARY ARITHMETIC.

IN THE DECIMAL NUMBERING SYSTEM WHICH VIRTUALLY EVERYONE HAS BEEN EDUCATED IN, FRACTIONS OF A NUMBER ARE REPRESENTED BY DIGITS PLACED TO THE RIGHT OF A DECIMAL POINT. EACH POSITION TO THE RIGHT OF SUCH A POINT REPRESENTS UNITS OF DECREASING POWERS OF 10. THUS THE NUMBER:

0 . 1 2 5 (DECIMAL)

ACTUALLY REPRESENTS:

	1 . .	TENTH (1/10 OR 10 TO THE -1 POWER)
PLUS:	2 .	HUNDREDTHS (OR 10 TO THE -2 POWER)
PLUS:	5	THOUSANDTHS (OR 10 TO THE -3 POWER)

THE CONCEPT IS EXACTLY THE SAME FOR BINARY ARITHMETIC EXCEPT THAT NOW EACH POSITION TO THE RIGHT OF THE DECIMAL POINT REPRESENTS UNITS OF DECREASING POWERS OF 2! THUS THE NUMBER:

0 . 1 1 1 (BINARY)

REPRESENTS:

	1 . .	HALF (1/2 OR 2 TO THE -1 POWER)
PLUS:	1 .	QUARTER (OR 2 TO THE -2 POWER)
PLUS:	1	EIGHTH (OR 2 TO THE -3 POWER)

THUS THE ABOVE BINARY NUMBER 0.111 REPRESENTS A FRACTIONAL NUMBER WHICH WHEN CONVERTED TO DECIMAL IS EQUAL TO:

1/2 + 1/4 + 1/8 = 7/8 OR .875 (DECIMAL)

THE MANNER IN WHICH FRACTIONAL BINARY NUMBERS ARE REPRESENTED BRINGS OUT AN INTERESTING POINT WHICH MANY READERS MAY HAVE HEARD OF, BUT NOT TRULY UNDERSTOOD - THE INTRODUCTION OF ERRORS INTO CALCULATIONS DONE WITH A DIGITAL COMPUTER DUE TO THE MANIPULATION OF FRACTIONS THAT CAN NOT BE "FINALIZED." AS AN ANALOGY, THERE ARE SIMILAR CASES IN DECIMAL ARITHMETIC, SUCH AS THE CASE WHEN THE NUMBER 1 IS DIVIDED BY 3. THE ANSWER IS:

0.33333333333333333333.....

OR A NON-ENDING SERIES OF 3'S AFTER THE DECIMAL POINT. THE ACCURACY OR "PRECISION" WITH WHICH A CALCULATION INVOLVING SUCH A NUMBER CAN BE CARRIED OUT IS DETERMINED BY HOW MANY "SIGNIFICANT" DIGITS ARE USED IN FURTHER CALCULATIONS INVOLVING THE FRACTION. FOR INSTANCE, THEORETICALLY, IF THE NUMBER 1 IS DIVIDED BY 3 AND THEN MULTIPLIED BY 3, ONE WOULD GET BACK 1 AS A RESULT. HOWEVER, IF THE RESULT OF THE DIVISION IS ACTUALLY MULTIPLIED BY 3, THE ANSWER IS NOT ACTUALLY ONE, BUT APPROACHES

THAT VALUE AS THE NUMBER OF SIGNIFICANT DIGITS USED IN THE CALCULATION IS INCREASED. OBSERVE.

```
   0.3   (ONE SIGNIFICANT DIGIT USED)
 X   3
 ---
  .9   (ANSWER IS OFF BY 10%)

  0.33  (TWO SIGNIFICANT DIGITS USED)
 X   3
 ----
  .99   (ANSWER IS OFF BY 1%)

  0.333 (THREE SIGNIFICANT DIGITS USED)
 X   3
 -----
  .999  (ANSWER IS OFF BY 0.1%)
```

A SIMILAR SITUATION EXISTS WITH BINARY ARITHMETIC EXCEPT THERE ARE NOW MANY MORE CASES WHERE THE "NON-ENDING" FRACTION SITUATION CAN OCCUR. FOR INSTANCE, THE VALUE 0.1 IS TRULY REPRESENTED IN THE DECIMAL SYSTEM, BUT IN THE BINARY SYSTEM, THE DECIMAL VALUE 0.1 CAN ONLY BE APPROXIMATED - AND SIMILARLY TO THE ABOVE, THE MORE BINARY DIGITS USED, THE CLOSER THE VALUE APPROACHES THE TRUE VALUE OF 0.1. OBSERVE.

USING 4 BINARY DIGITS = 0.0001 = 1/16 = .0625 (OFF 37.5%)

9 DIGITS = 0.000110011 = 1/16 + 1/32 + 1/256 + 1/512 = .0996 (OFF .4%)

NOTE TOO, THAT THE BINARY REPRESENTATION IS A NON-ENDING SERIES:

0.1 DECIMAL = 0.0001100110011001100110011001100... (BINARY)

AND CAN NOT REACH THE THEORETICAL TRUE VALUE OF 0.1 AS IN THE DECIMAL SYSTEM. THUS, IF 0.1 AS REPRESENTED IN THE BINARY SYSTEM IS MULTIPLIED BY, SAY 10, (WHICH CAN BE TRULY REPRESENTED IN THE BINARY SYSTEM!) THE THEORETICAL VALUE OF 1.0 CAN ONLY BE APPROACHED, AND THE MORE BITS USED TO HOLD THE BINARY EQUIVELANT, THE CLOSER ONE CAN APPROACH THE TRUE ANSWER. THUS, ONE CAN SEE ANOTHER REASON FOR USING MULTIPLE-PRECISION ARITHMETIC IN A DIGITAL COMPUTER EVEN IF ONE DOES NOT WANT TO HANDLE BIG NUMBERS! THIS IS BECAUSE THE MORE BITS AVAILABLE TO STORE A FRACTIONAL NUMBER - THE MORE "PRECISION" ONE CAN MAINTAIN IN PERFORMING CALCULATIONS. ONE SHOULD NOW ALSO REALIZE, THAT THE MORE COMPLEX A SERIES OF MATHEMATICAL OPERATION BECOMES, IN OTHER WORDS, THE MORE TIMES A NUMBER THAT CAN NOT TRULY BE REPRESENTED IS MULTIPLIED OR DIVIDED, THE WIDER WILL BECOME THE MARGIN OF ERROR IN THE FINAL ANSWER!

NOW THAT ONE HAS A GRASP OF HOW BINARY NUMBERS CAN REPRESENT FRACTIONAL NUMBERS WHEN PLACED TO THE RIGHT OF A DECIMAL POINT, ONE CAN PROCEED TO INVESTIGATE "FLOATING-POINT" ARITHMETIC USING A DIGITAL COMPUTER.

FLOATING-POINT ARITHMETIC

JUST AS ONE CAN REPRESENT DECIMAL NUMBERS IN FLOATING-POINT FORMAT, I.E., A STRING OF SIGNIFICANT DIGITS RAISED TO A POWER OF 10, ONE CAN ALSO TREAT BINARY NUMBERS IN A SIMILAR MANNER AS A STRING OF BINARY DIGITS RAISED TO A POWER OF 2.

WHEN HANDLING NUMBERS IN FLOATING-POINT FORMAT THE NUMBER IS REPRESENTED AS TWO PARTS. THE "SIGNIFICANT DIGITS" PORTION IS REFERRED TO AS THE "MANTISSA" AND THE POWER TO WHICH THE NUMBER IS TO BE RAISED IS REFERRED TO AS THE "EXPONENT." IN DECIMAL FLOATING-POINT FORMAT THE NUMBER "5" COULD BE EXPRESSED AS:

5.0 E+0 = 5 X 1 = 5

OR 50.0 E-1 = 50 X 1/10 = 5

OR 0.5 E+1 = 0.5 X 10 = 5

WHILE IN BINARY FLOATING-POINT FORMAT THE SAME NUMBER COULD BE EXPRESSED AS:

101.0 E+0 = 5 X 1 = 5

OR 101000.0 E-3 = 40 X 1/8 = 5

OR 0.101 E+3 = 5/8 X 8 = 5

NOTE THAT THE "MECHANICS" OF THE CORRESPONDENCE BETWEEN THE EXPONENT AND THE LOCATION OF THE DECIMAL POINT IN THE MANTISSA IS THE SAME FOR BOTH NUMBERING SYSTEMS. IF THE SIGNIFICANT DIGITS ARE MOVED TO THE RIGHT OF THE DECIMAL POINT THEN THE EXPONENT IS INCREASED A UNIT FOR EACH POSITION THE MANTISSA IS SHIFTED. IF THE DIGITS IN THE MANTISSA ARE SHIFTED TO THE LEFT, THEN THE EXPONENT IS DECREASED. THE ONLY DIFFERENCE BETWEEN THE TWO SYSTEMS IS THAT THE EXPONENT IN THE DECIMAL SYSTEM IS SPECIFIED FOR POWERS OF 10, WHILE IN THE BINARY SYSTEM IT IS FOR POWERS OF 2.

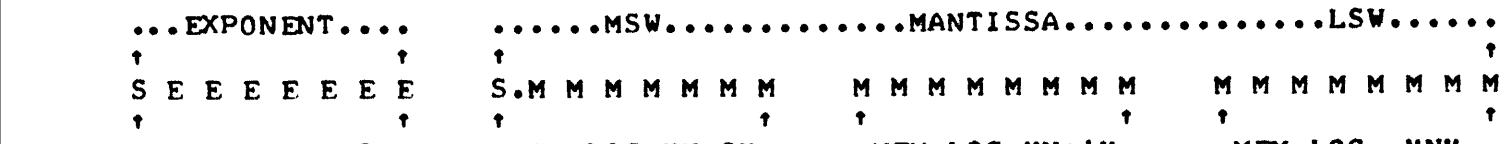
THE READER SHOULD NOW SEE THAT IT CAN BE QUITE A SIMPLE MATTER TO HANDLE BINARY NUMBERS USING A FLOATING-POINT FORMAT IF ONE SIMPLY ARRANGES TO KEEP TABS ON THE "EXPONENT" PORTION IN ONE REGISTER (OR REGISTERS) AND MAINTAINS THE "MANTISSA" PORTION IN ANOTHER REGISTER (OR SEVERAL REGISTERS FOR MORE PRECISION). FURTHERMORE, A VERY SIMPLE RELATIONSHIP CAN BE MAINTAINED BETWEEN THE MANTISSA AND THE EXPONENT TO FACILITATE KEEPING TRACK OF A "DECIMAL" POINT. ONCE ONE HAS SELECTED A GIVEN POSITION AS A REFERENCE JUNCTION IN THE MANTISSA PORTION, ONE HAS ONLY TO OBSERVE THE FOLLOWING PROCEDURE FOR MANIPULATING THE NUMBER AND KEEPING TRACK OF THE "DECIMAL" POINT:

EACH TIME THE MANTISSA IS SHIFTED TO THE RIGHT - INCREMENT THE EXPONENT!

EACH TIME THE MANTISSA IS SHIFTED TO THE LEFT - DECREMENT THE EXPONENT!

FOR THE REMAINDER OF THIS CHAPTER, A CONVENTION FOR STORING NUMBERS IN FLOATING-POINT FORMAT WILL BE ESTABLISHED. NUMBERS WILL BE STORED IN FOUR CONSECUTIVE WORDS IN MEMORY. THE FIRST WORD IN A GROUP WILL BE USED TO STORE THE "EXPONENT" WITH THE MOST SIGNIFICANT BIT IN THE WORD USED TO REPRESENT THE "SIGN" OF THE EXPONENT. A "1" IN THE MOST SIGNIFICANT BIT POSITION MEANS THE NUMBER IS "NEGATIVE." THE NEXT THREE WORDS WILL THEN HOLD THE "MANTISSA" PORTION IN TRIPLE-PRECISION FORMAT, WITH THE FIRST BIT IN THE FIRST WORD OF THE MANTISSA BEING USED AS THE "SIGN" BIT. THE REMAINING BITS IN THAT WORD WILL BE THE MOST SIGNIFI-

CANT BITS OF THE NUMBER. THE REMAINING TWO WORDS IN A GROUP WILL HOLD THE LESS SIGNIFICANT BITS OF THE MANTISSA. FURTHERMORE, THERE WILL BE AN IMPLIED "DECIMAL" POINT IMMEDIATELY TO THE RIGHT OF THE "SIGN" BIT IN THE MANTISSA. THE FORMAT IS ILLUSTRATED HERE:



NOTE THE ORDER OF THE MEMORY ADDRESSES ASSIGNED TO THE STORAGE OF A NUMBER. AS IN THE PAST, THE ORDER OF STORAGE IS AN ARBITRARY ASSIGNMENT BUT ONCE IT HAS BEEN ASSIGNED IT MUST BE ADHERED TO WITHIN A PROGRAM. THE ORDER SHOWN IS THE ONE THAT WILL BE USED IN THE DISCUSSION AND PROGRAM EXAMPLES USED FOR THE REMAINDER OF THIS SECTION.

NOTE TOO, THAT A CONVENTION HAS BEEN ESTABLISHED THAT WILL CONSIDER A "DECIMAL POINT" (ACTUALLY, PERHAPS IT SHOULD BE TERMED A "BINARY" POINT) TO BE LOCATED TO THE RIGHT OF THE DESIGNATED "SIGN" BIT FOR THE MANTISSA. THIS MEANS THAT ALL NUMBERS STORED IN FLOATING-POINT FORMAT WILL BE REPRESENTED AS A FRACTIONAL NUMBER! ALSO, THE READER CAN SEE THAT WITH ONE BIT OUT OF THE THREE WORDS USED TO STORE THE MANTISSA USED TO HOLD THE "SIGN" OF THE MANTISSA, THAT 23 (DECIMAL) BITS ARE LEFT TO HOLD THE ACTUAL MAGNITUDE OF THE MANTISSA. SIMILARLY, THE EXPONENT HAS 7 BITS WITH WHICH TO REPRESENT THE MAGNITUDE OF IT'S VALUE. FURTHERMORE, AN EXPONENT MUST BE AN INTEGER VALUE AS THERE WILL BE NO IMPLIED "DECIMAL POINT" IN THE EXPONENT REGISTER.

FLOATING-POINT NORMALIZATION

"NORMALIZATION" CAN BE CONSIDERED AS A "STANDARDIZING" PROCESS THAT WILL PLACE A NUMBER INTO A FIXED POSITION AS A REFERENCE POINT FROM WHICH TO COMMENCE OPERATIONS. FOR THE PURPOSES OF THIS DISCUSSION, THE TERM "NORMALIZATION" WILL MEAN TO PLACE A NUMBER INTO ITS STORAGE REGISTERS SO THAT THE "MANTISSA" WILL HAVE A VALUE THAT IS GREATER THAN OR EQUAL TO "1/2" BUT LESS THAN "1." PUT ANOTHER WAY, THIS MEANS THAT ANY NUMBER TO BE MANIPULATED BY A "FLOATING-POINT ROUTINE" WILL FIRST BE SHIFTED SO THAT THE MOST SIGNIFICANT BINARY DIGIT IS NEXT TO THE IMPLIED "BINARY" POINT IN THE MOST SIGNIFICANT WORD OF THE MANTISSA STORAGE REGISTERS. FOR INSTANCE IF A BINARY NUMBER SUCH AS:

101.0 E+0 (DECIMAL 5)

WAS RECEIVED BY AN INPUT ROUTINE TO A FLOATING-POINT PROGRAM, THE NUMBER WOULD BE "NORMALIZED" WHEN IT WAS PLACED IN THE FORM:

0.101 E+3 (WHICH IS 5/8 X 8 = 5 DECIMAL!)

SIMILARLY, IF AFTER SAY A BINARY DIVISION OPERATION IN WHICH THE NUMBER 1 HAD BEEN DIVIDED BY 10 (DECIMAL) AND ONE HAD THE ANSWER:

0.0001100110011100... E+0 (DECIMAL 0.1)

THE NUMBER WOULD BE CONSIDERED NORMALIZED WHEN IT WAS IN THE FORMAT:

0.1100110011001100.. E-3 (DECIMAL 0.1)

NOTE THAT "NORMALIZING" A NUMBER IS A PRETTY EASY MATTER. IN THE FIRST EXAMPLE ABOVE THE NUMBER WAS NORMALIZED BY SHIFTING THE ORIGINAL NUMBER TO THE RIGHT UNTIL THE MOST SIGNIFICANT BIT WAS JUST TO THE RIGHT OF THE DECIMAL POINT, WHILE INCREMENTING THE "EXPONENT" FOR EACH SHIFT OPERATION. IN THE SECOND EXAMPLE, THE NUMBER IS SHIFTED IN THE REVERSE DIRECTION WHILE DECREMENTING THE EXPONENT.

THERE ARE SEVERAL REASONS FOR WANTING TO "NORMALIZE" A NUMBER WHEN WORKING WITH A FLOATING-POINT PROGRAM. THE FIRST HAS TO DO WITH THE FACT THAT GENERALLY NUMBERS WILL ORIGINATE FROM A HUMAN WHO WILL BE USING THE COMPUTER TO MANIPULATE NUMBERS IN DECIMAL FORMAT AND THEREFORE THE COMPUTER WILL HAVE TO CONVERT NUMBERS FROM SAY, A DECIMAL FLOATING-POINT FORMAT, TO THE BINARY FORMAT USED BY THE COMPUTER. THERE WILL BE MORE DISCUSSION ON THIS MATTER LATER IN THIS CHAPTER AFTER A NUMBER OF BINARY FLOATING-POINT OPERATIONS HAVE BEEN PRESENTED. THE SECOND REASON FOR NORMALIZING NUMBERS, AND A VERY IMPORTANT ONE, IS BECAUSE THE PROCESS WILL ALLOW MORE SIGNIFICANT BINARY DIGITS TO BE RETAINED IN A FIXED LENGTH REGISTER. THIS CAN BE SEEN BY OBSERVING IN THE ABOVE EXAMPLE OF THE CASE WHERE 0.1 DECIMAL IS NORMALIZED, THAT SHIFTING THE BINARY NUMBER TO THE LEFT THREE PLACES WOULD ALLOW SEVERAL MORE LEAST SIGNIFICANT BITS TO BE PLACED IN A FIXED LENGTH REGISTER FOR THE NON-ENDING BINARY SERIES OF "0.110011001100..." AND THUS ALLOW MORE ACCURACY IN THE BINARY CALCULATIONS THAT MIGHT FOLLOW!

A ROUTINE FOR "NORMALIZING" BINARY NUMBERS WILL BE PRESENTED SHORTLY. IN THE ROUTINE FOR "NORMALIZING" NUMBERS, AND VARIOUS OTHER MATHEMATICAL ROUTINES IN THIS CHAPTER, VARIOUS LOCATIONS ON PAGE 00 WILL BE USED FOR STORING NUMBERS THAT ARE TO BE MANIPULATED BY THE ROUTINES AS WELL AS HOLDING "COUNTERS" AND "POINTERS" IN MEMORY LOCATIONS. A LIST OF THE LOCATIONS USED WILL BE PROVIDED LATER. ALSO, BEFORE GETTING INTO THE ACTUAL BINARY FLOATING-POINT ROUTINES, THE READER SHOULD BE INFORMED THAT IN THE FOLLOWING ROUTINES, REFERENCES WILL BE MADE TO A "FLOATING-POINT ACCUMULATOR" AND "FLOATING-POINT OPERAND." THE FLOATING-POINT ACCUMULATOR AND OPERAND WILL BE SEPARATE GROUPS CONSISTING OF FOUR CONSECUTIVE MEMORY WORDS ON PAGE 00 USED TO STORE THE "ACTIVE" NUMBERS THAT ARE MANIPULATED BY THE FLOATING-POINT ROUTINES. THEY WILL, OF COURSE, BE ARRANGED IN THE FORMAT DESCRIBED EARLIER OF A SINGLE MEMORY WORD "EXPONENT" AND A TRIPLE-PRECISION "MANTISSA." THE "FLOATING-POINT ACCUMULATOR" WILL BE THE FOCAL POINT FOR ANY FLOATING-POINT ROUTINE AS ALL THE RESULTS OF FLOATING-POINT CALCULATIONS WILL BE PLACED THERE. THE "FLOATING-POINT OPERAND" WILL BE USED PRIMARILY FOR HOLDING AND MANIPULATING THE NUMBER THAT THE FLOATING-POINT ACCUMULATOR OPERATES ON. FOR BREVITY IN FURTHER DISCUSSIONS, THE FLOATING-POINT ACCUMULATOR WILL BE ABBREVIATED AS "FPACC" AND THE FLOATING-POINT OPERAND AS "FPOP."

```
                MNEMONIC                    COMMENTS
            --------------              --------------

FPNORM,  LAB            /CHECK REGISTER "B" FOR SPECIAL CASE
         NDA            /SET FLAGS AFTER LOAD OPERATION
         JTZ  NOEXCO    /IF "B" WAS 0, DO STANDARD NORMALIZATION
         LLI  127       /OTHERWISE SET EXPONENT OF FPACC
         LMB            /TO VALUE FOUND IN "B" AT START OF RTN
NOEXCO,  LLI  126       /SET POINTER TO MSW OF FPACC MANTISSA
         LAM            /AND GET MSW OF FPACC MANTISSA INTO ACC
         LLI  100       /CHANGE POINTER TO "SIGN" STORAGE ADDRESS
         NDA            /SET FLAGS AFTER PREVIOUS "LAM" OPERATION
         JTS  ACCMIN    /SEE IF MSB IN MSW = 1, YES = MINUS #
         XRA            /IF MSB = 0, HAVE POSITIVE VALUE MANTISSA
         LMA            /SO SET "SIGN" STORAGE TO 000
```

```
                MNEMONIC                COMMENTS
                --------                --------

                JMP  ACZERT   /PROCEED TO SEE IF FPACC = ZERO
ACCMIN,         LMA           /ORIG FPACC = NEG #, PUT DATA IN "SIGN"
                LBI  004      /SET PRECISION CNTR TO 4 (USE EXTRA WORD)
                LLI  123      /AND PNTR TO FPACC LSW-1 (USE EXTRA WORD)
                CAL  COMPLM   /TWO'S COMPLEMENT FPACC + 1 EXTRA MEM WORD
ACZERT,         LLI  126      /CHECK TO SEE IF FPACC CONTAINS ZERO
                LBI  004      /SET A COUNTER
LOOK0,          LAM           /GET A PART OF FPACC
                NDA           /SET FLAGS AFTER LOAD OPERATION
                JFZ  ACNONZ   /IF FIND ANYTHING THEN FPACC IS NOT ZERO
                DCL           /OTHERWISE MOVE POINTER TO NEXT PART
                DCB           /DECREMENT THE LOOP COUNTER
                JFZ  LOOK0    /AND IF NOT FINISHED CHECK NEXT PART
                LLI  127      /IF REACH HERE FPACC WAS ZERO
                XRA           /SO MAKE SURE EXPONENT OF FPACC IS ALSO
                LMA           /ZERO BY PUTTING ZERO IN IT!
                RET           /CAN THEN EXIT THE NORMALIZATION ROUTINE
ACNONZ,         LLI  123      /IF FPACC HAS VALUE, SET UP POINTERS
                LBI  004      /AND "PRECISION" VALUE (P = 4 TO HANDLE
                CAL  ROTATL   /SPECIAL CASES) AND ROTATE FPACC L E F T
                LAM           /THEN GET MSB OF MSW IN MANTISSA
                NDA           /SET FLAGS AFTER LOAD OPERATION
                JTS  ACCSET   /IF MSB = 1, HAVE FOUND MSB IN FPACC
                INL           /IF NOT, ADVANCE PNTR TO FPACC EXPONENT
                CAL  CNTDWN   /AND DECREMENT THE VALUE OF THE EXPONENT
                JMP  ACNONZ   /THEN CONTINUE IN THE ROTATING LEFT LOOP
ACCSET,         LLI  126      /COMPENSATE FOR LAST ROTATE LEFT WHEN MSB
                LBI  003      /FOUND TO LEAVE ROOM FOR "SIGN" IN MSB OF
                CAL  ROTATR   /FPACC MANTISSA BY DOING ONE ROTATE RIGHT
                LLI  100      /SET POINTER TO ORIGINAL "SIGN" STORAGE
                LAM           /GET ORIGINAL "SIGN" INDICATOR
                NDA           /SET FLAGS AFTER LOAD OPERATION
                RFS           /FINISHED AS VALUE IN FPACC IS POSITIVE
                LLI  124      /ORIG "SIGN" NEGATIVE, SO SET PNTR TO LSW
                LBI  003      /OF FPACC AND SET PRECISION COUNTER
                CAL  COMPLM   /TWO'S COMPLEMENT THE NORMALIZED FPACC
                RET           /THAT'S ALL FOR "NORMALIZATION"
```

THERE ARE SEVERAL ITEMS IN THE ABOVE ROUTINE THAT MIGHT CONFUSE THE READER IF NOT EXPLAINED. FIRST OF ALL, THE ROUTINE FIRST CHECKS CPU REGISTER "B" WHEN IT IS ENTERED. IF "B" CONTAINS 000 THEN THE ROUTINE WILL PROCEED ON TO THE NEXT PART OF THE PROGRAM. IF "B" CONTAINS SOME NON-ZERO VALUE, THEN THAT VALUE WILL BE PLACED IN THE EXPONENT PORTION OF THE FPACC. THIS WAS DONE SO THAT THE "FPNORM" SUBROUTINE COULD HANDLE NUMBERS THAT WERE NOT IN FLOATING-POINT FORM. FOR INSTANCE, WHEN A NUMBER IS FIRST RECEIVED FROM AN INPUT DEVICE IT WILL GENERALLY BE IN A FORM SUCH AS THE EXAMPLE FOR THE BINARY EQUIVELENT OF 5 (DECIMAL) AS ILLUSTRATED:

00 000 000 00 000 000 00 000 101

WHEN IN TRIPLE-PRECISION FORMAT. NOW THE ABOVE FORMAT COULD BE CONVERTED TO THE DESIRED FLOATING-POINT FORMAT BY ASSUMING A "BINARY" POINT EXISTED TO THE RIGHT OF THE LEAST SIGNIFICANT BIT, AND SHIFTING THE ENTIRE NUMBER TO THE RIGHT WHILE INCREMENTING THE BINARY EXPONENT REGISTER. HOWEVER, THE TECHNIQUE WOULD CAUSE A SLIGHT PROBLEM. HOW COULD ONE TELL WHERE THE MOST SIGNIFICANT BIT OF THE BINARY NUMBER WAS? A WAY

AROUND THAT PROBLEM IS TO SIMPLY SHIFT THE REGISTERS TO THE LEFT UNTIL THE FIRST "1" (MOST SIGNIFICANT BIT) IS IN THE DESIRED POSITION. IF THIS IS DONE, ONE MUST FIRST SET THE "EXPONENT" TO THE HIGHEST POSSIBLE VALUE THAT COULD BE CONTAINED IN THE REGISTERS AND THEN DECREMENT THAT VALUE FOR EACH SHIFT TO THE LEFT. REMEMBERING EARLIER THAT THERE ARE 23 (DECIMAL) BITS AVAILABLE FOR STORING THE MANTISSA WHEN TRIPLE-PRECISION FORMATTING IS BEING USED (AS ONE BIT IS RESERVED FOR THE "SIGN" OF THE NUMBER) THEN ONE WOULD SIMPLY LOAD REGISTER "B" WITH 27 (OCTAL WHICH IS 23 DECIMAL) BEFORE CALLING THE "FPNORM" ROUTINE IF THE NUMBER TO BE NORMALIZED WAS NOT IN FLOATING-POINT FORMAT. THE FOLLOWING ILLUSTRATIONS SHOULD CLARIFY THE MATTER.

ORIGINAL NUMBER WHICH IS NOT IN FLOATING-POINT FORMAT

00 000 000 00 000 000 00 000 101

DESIRED FLOATING-POINT FORMAT

SE EEE EEE S.M MMM MMM MM MMM MMM MM MMM MMM

ORIGINAL NUMBER PLACED IN FPACC AND EXPONENT SET TO 27 (OCTAL)

00 010 111 0.0 000 000 00 000 000 00 000 000

ORIGINAL NUMBER IS THEN NORMALIZED BY ROTATING LEFT

00 000 011 0.1 010 000 00 000 000 00 000 000

SINCE THE EXPONENT WAS DECREMENTED EACH TIME THE NUMBER WAS ROTATED LEFT THE FINAL EXPONENT VALUE IS THE SAME AS IF THE NUMBER HAD BEEN ROTATED TO THE RIGHT TO ACCOMPLISH THE NORMALIZATION!

THE READER SHOULD ALSO NOTE THAT THE "FPNORM" ALSO CHECKS TO SEE IF THE NUMBER TO BE NORMALIZED IS NEGATIVE. IF IT IS, THE ROUTINE KEEPS TRACK OF THAT FACT AND MAKES THE NUMBER POSITIVE IN ORDER TO ACCOMPLISH THE NORMALIZATION PROCEDURE. IF IT DID NOT, THE NORMALIZATION ROUTINE WOULD NOT WORK AS CAN BE SEEN WHEN ONE RECALLS WHAT A NUMBER SUCH AS MINUS 5 APPEARS LIKE IN IT'S TWO'S COMPLEMENT FORM:

11 111 111 11 111 111 11 111 011

AFTER THE NUMBER HAS BEEN NORMALIZED IN IT'S POSITIVE FORM, IT IS CONVERTED BACK TO THE NEGATIVE FORM SO THAT THE NUMBER MINUS 5 WOULD APPEAR WHEN NORMALIZED AS:

00 000 011 1.0 110 000 00 000 000 00 000 000

THE READER SHOULD WORK THROUGH THE PROCEDURE USING PENCIL AND PAPER TO MAKE SURE THE PROCESS IS UNDERSTOOD FOR HANDLING NEGATIVE NUMBERS AS IT CAN BE CONFUSING AT FIRST GLANCE. NOTE THAT THE NORMALIZED MINUS VALUE HAS THE MOST SIGNIFICANT BIT POSITION IN THE MANTISSA SET TO A "1" TO INDICATE A NEGATIVE VALUE!

ANOTHER POINT OF INTEREST IN THE "FPNORM" ROUTINE IS THAT THE ROUTINE TESTS TO SEE IF THE FPACC CONTAINS ZERO. NOTE THAT IF THIS TEST WAS NOT MADE AND APPROPRIATE ACTION TAKEN TO EXIT THE ROUTINE, THAT THE ROUTINE WOULD BECOME "HUNG-UP" IN THE ROTATE LEFT LOOP AS IT WOULD FAIL TO EVER SEE A "1" APPEAR IN THE MOST SIGNIFICANT BIT POSITION! WHEN A ZERO CONDITION IS FOUND IN THE MANTISSA, THE ROUTINE SETS THE EXPONENT

PART OF THE FPACC TO ZERO AS AN ADDITIONAL MEASURE.

FINALLY, THE READER WILL NOTE THAT THE FIRST PART OF THE NORMALIZATION ROUTINE ASSUMES THE MANTISSA USES FOUR MEMORY WORDS - THIS WAS DONE SO THAT THE ROUTINE COULD HANDLE SOME SPECIAL CASES THAT CAN OCCUR AFTER OPERATIONS SUCH AS MULTIPLICATION WHERE IT IS NECESSARY TO HAVE SOME ADDITIONAL "PRECISION." IN CASES WHERE THE FEATURE IS NOT NEEDED, THE EXTRA MEMORY WORD SHOULD BE SET TO 000 BEFORE USING THE "FPNORM" ROUTINE.

THE "ROTATL" AND "ROTATR" SUBROUTINES CALLED BY "FPNORM" ARE SHORT ROUTINES THAT HAVE BEEN SET UP FOR "NTH-PRECISION" OPERATION AS WITH OTHER ALGORITHMS DISCUSSED IN THIS CHAPTER. BEFORE ENTERING THE ROUTINES THE CALLING PROGRAM SETS THE STARTING ADDRESS OF THE STRING OF MEMORY WORDS TO BE PROCESSED IN THE "H & L" REGISTERS AND THE NUMBER OF WORDS IN THE STRING IN REGISTER "B." THE TWO ROUTINES ARE SHOWN BELOW.

```
                MNEMONIC                    COMMENTS
            --------------              --------------

ROTATL,  NDA            /CLEAR CARRY FLAG AT THIS ENTRY POINT
ROTL,    LAM            /FETCH WORD FROM MEMORY
         RAL            /ROTATE LEFT (WITH CARRY)
         LMA            /RESTORE ROTATED WORD TO MEMORY
         DCB            /DECREMENT "PRECISION" COUNTER
         RTZ            /RETURN TO CALLING ROUTINE WHEN DONE
         INL            /OTHERWISE ADVANCE PNTR TO NEXT WORD
         JMP ROTL       /AND ROTATE ACROSS THE MEM WORD STRING

ROTATR,  NDA            /CLEAR CARRY FLAG AT THIS ENTRY POINT
ROTR,    LAM            /FETCH WORD FROM MEMORY
         RAR            /ROTATE RIGHT (WITH CARRY)
         LMA            /RESTORE ROTATED WORD TO MEMORY
         DCB            /DECREMENT "PRECISION" COUNTER
         RTZ            /RETURN TO CALLING ROUTINE WHEN DONE
         DCL            /GOING OTHER WAY SO DECREMENT MEM PNTR
         JMP ROTR       /AND ROTATE ACROSS THE MEM WORD STRING
```

FLOATING-POINT ADDITION

FLOATING-POINT ADDITION IS QUITE STRAIGHT FORWARD, AND IN FACT ONE CAN USE THE "ADDER" ROUTINE ALREADY DEVELOPED EARLIER IN THIS CHAPTER FOR THE MANTISSA PORTION OF A SET OF FLOATING-POINT NUMBERS. HOWEVER, THERE ARE A FEW OTHER PARAMETERS THAT MUST BE CONSIDERED IN DEVELOPING THE OVER-ALL ROUTINE.

WHEN TWO NUMBERS ARE TO BE ADDED IT WILL BE ASSUMED THAT THEY HAVE BEEN POSITIONED IN THE "FPACC" AND THE "FPOP" MEMORY STORAGE AREAS. A FEW ITEMS THAT SHOULD BE CONSIDERED IN DEVELOPING THE BASIC FLOATING-POINT ADDITION ROUTINE INCLUDE THE FOLLOWING.

SUPPOSE EITHER THE "FPOP" OR "FPACC" CONTAIN ZERO? OR THEY BOTH CONTAIN ZERO? IN THE LATTER CASE THE ROUTINE COULD BE IMMEDIATELY EXITED AS THE ANSWER IS SITTING IN THE "FPACC." IF THE "FPACC" IS ZERO, BUT THE "FPOP" IS NOT, THEN ONE HAS MERELY TO PLACE THE CONTENTS OF THE "FPOP" INTO THE "FPACC" (AS THE CONVENTION WAS ESTABLISHED EARLIER THAT THE "RESULT" OF AN OPERATION WOULD ALWAYS BE LEFT IN THE "FPACC"). AND, FOR THE CASE WHERE THE "FPACC" CONTAINS A VALUE, BUT THE "FPOP" IS ZERO, ONE CAN AGAIN IMMEDIATELY EXIT THE ROUTINE.

BUT, AS WILL MORE LIKELY BE THE CASE WHEN THE FLOATING-POINT ADD ROUTINE IS CALLED, BOTH THE "FPACC" AND THE "FPOP" WILL CONTAIN SOME NON-ZERO VALUE, AND THUS ONE COULD IMMEDIATELY PROCEED TO PERFORM THE ADDITION OPERATION, RIGHT? WRONG! SINCE FLOATING-POINT OPERATIONS ALLOW THE MANIPULATING OF LARGE MAGNITUDES OF NUMBERS, BECAUSE OF THE EXPONENT METHOD OF MAINTAINING MAGNITUDES, IT IS QUITE POSSIBLE THAT AN OPERATOR MIGHT ASK FOR AN ADDITION OF A VERY SMALL NUMBER TO A VERY LARGE NUMBER (OR THIS MIGHT OCCUR IN THE MIDDLE OF A VERY COMPLEX CALCULATION WHERE AN OPERATOR DOES NOT SEE THE INTERMEDIATE RESULTS). HOWEVER, READERS KNOW THAT IF THE DIFFERENCE BETWEEN THE TWO NUMBERS TO BE ADDED IS SO GREAT THAT THERE CAN BE NO CHANGE IN THE "SIGNIFICANT" DIGITS IN THE CALCULATION (THE VALUE STORED IN THE MANTISSA) THEN THERE IS NO USE IN PERFORMING THE ADDITION PROCESS! SO, THE NEXT STEP IN THE FLOATING-POINT ADDITION ROUTINE WOULD BE TO CHECK TO SEE WHETHER OR NOT THE MAGNITUDES OF THE NUMBERS ARE WITHIN "SIGNIFICANT" RANGE OF ONE ANOTHER. IF THEY ARE NOT, THEN THE LARGEST VALUE SHOULD BE PLACED IN THE "FPACC" AS THE ANSWER!

IF THE MAGNITUDES OF THE TWO NUMBERS ARE WITHIN "SIGNIFICANT" RANGE THEN THE TWO NUMBERS MAY BE ADDED BUT BEFORE THIS CAN BE DONE, THEY MUST FIRST BE "ALIGNED" BY SHIFTING ONE OF THE NUMBERS UNTIL THE "EXPONENT" IS EQUAL IN MAGNITUDE WITH THE SECOND NUMBER. THE "ALIGNMENT" IS ACCOMPLISHED BY FINDING OUT WHICH EXPONENT IS THE SMALLEST AND SHIFTING THE MANTISSA OF THAT NUMBER TO THE RIGHT (WHILE INCREMENTING THE EXPONENT FOR EACH SHIFT) UNTIL IT IS PROPERLY ALIGNED. THE SHIFTING PROCEDURE IS QUITE STRAIGHT-FORWARD SINCE IT CAN BE HANDLED BY A "NTH-PRECISION" REGISTER ROTATE OPERATION. HOWEVER, THERE IS ONE SPECIAL CONSIDERATION FOR THE CASE OF A NEGATIVE NUMBER BEING SHIFTED TO THE RIGHT - ONE MUST INSERT A "1" INTO THE MOST SIGNIFICANT BIT POSITION EACH TIME A SHIFT IS MADE IN ORDER TO MAINTAIN THE "MINUS" VALUE PROPERLY (AND KEEP THE SIGN BIT IN IT'S PROPER STATE). THIS CAN BE ACCOMPLISHED EASILY AS THE READER WILL SEE IN THE "FPADD" ROUTINE BY INSERTING A "1" INTO THE CARRY BIT AND THEN CALLING THE "ROTR" SUBROUTINE WHICH IS SIMPLY ANOTHER ENTRY POINT TO THE "ROTATR" SUBROUTINE PRESENTED EARLIER (AVOIDING THE "NDA" ENTRY POINT IN THE ROUTINE WHICH WOULD CAUSE THE CARRY BIT TO BE SET TO A "0" CONDITION IF EXECUTED).

ONE MORE CONSIDERATION THAT THE READER WILL NOTE IN THE FOLLOWING "FPADD" ROUTINE IS THAT THE TWO NUMBERS TO BE ADDED ARE SHIFTED TO THE RIGHT ONCE BEFORE THE ADDITION IS PERFORMED SO THAT ANY OVER-FLOW FROM THE ADDITION WILL STAY WITHIN THE "FPACC" THUS ALLOWING "NORMALIZATION" TO BE HANDLED BY THE PREVIOUSLY PRESENTED ROUTINE INSTEAD OF HAVING TO BE CONCERNED WITH THE STATUS OF THE CARRY FLAG AT THE END OF THE OPERATION. BECAUSE OF THIS SHIFTING OPERATION, AN ADDITIONAL MEMORY WORD IS USED BY BOTH THE "FPACC" AND "FPOP" AND THE ADDITION IS PERFORMED USING "QUAD-PRECISION." AT THE END OF THE ADDITION PROCESS THE RESULT IS NORMALIZED AND LEFT IN THE "FPACC."

```
                MNEMONIC                    COMMENTS
            --------------              --------------

FPADD,  LLI 126         /SET POINTER TO MSW OF FPACC
        LBI 003         /SET LOOP COUNTER
CKZACC, LAM             /FETCH PART OF FPACC
        NDA             /SET FLAGS AFTER LOADING OPERATION
        JFZ NONZAC      /FINDING ANYTHING MEANS FPACC NOT ZERO
        DCB             /IF THAT PART = 0, DECREMENT LOOP COUNTER
        JTZ MOVOP       /IF FPACC = 0, MOVE FPOP INTO FPACC
        DCL             /NOT FINISHED CHECKING, DECREMENT PNTR
        JMP CKZACC      /AND TEST NEXT PART OF FPACC
```

```
                MNEMONIC                    COMMENTS
        ----------------------      ----------------------

MOVOP,  CAL SWITCH      /SAVE POINTER TO LSW OF FPACC
        LHD             /SET "H" = 000 FOR SURE
        LLI 134         /SET POINTER TO LSW OF FPOP
        LBI 004         /SET A LOOP COUNTER
        CAL MOVEIT      /MOVE FPOP INTO FPACC = ANSWER
        RET             /EXIT FPADD
NONZAC, LLI 136         /SET POINTER TO MSW OF FPOP
        LBI 003         /SET LOOP COUNTER
CKZOP,  LAM             /GET MSW OF FPOP
        NDA             /SET FLAGS AFTER LOAD OPERATION
        JFZ CKEQEX      /IF NOT 0 THEN HAVE A NUMBER!
        DCB             /IF 0, DECREMENT LOOP COUNTER
        RTZ             /EXIT RTN IF FPOP = ZERO
        DCL             /ELSE DECREMENT PNTR TO NEXT PART OF FPOP
        JMP CKZOP       /AND CONTINUE TESTING FOR ZERO FPOP
CKEQEX, LLI 127         /CHECK FOR EQUAL EXPONENTS
        LAM             /GET FPACC EXPONENT
        LLI 137         /CHANGE POINTER TO FPOP EXPONENT
        CPM             /COMPARE EXPONENTS
        JTZ SHACOP      /IF SAME CAN SET UP FOR ADD OPERATION
        XRI 377         /IF NOT SAME, TWO'S COMPLEMENT THE VALUE
        ADI 001         /OF THE FPACC EXPONENT
        ADM             /AND ADD IN FPOP EXPONENT
        JFS SKPNEG      /IF + GO DIRECTLY TO ALIGNMENT TEST
        XRI 377         /IF NEGATIVE PERFORM TWO'S COMPLEMENT
        ADI 001         /ON THE RESULT
SKPNEG, CPI 030         /NOW SEE IF RESULT GREATER THAN 27 OCTAL
        JTS LINEUP      /IF NOT CAN PERFORM ALIGNMENT
        LAM             /IF NOT ALIGNABLE GET FPOP EXPONENT
        LLI 127         /SET POINTER TO FPACC EXPONENT
        SUM             /SUBTRACT FPACC EXPONENT FROM FPOP EXP
        RTS             /FPACC EXP GREATER SO JUST EXIT RTN
        LLI 124         /FPOP WAS GREATER, SET PNTR TO FPACC LSW
        JMP MOVOP       /GO PUT FPOP INTO FPACC & THEN EXIT RTN
LINEUP, LAM             /ALIGN FPACC AND FPOP, GET FPOP EXP
        LLI 127         /CHANGE POINTER TO FPACC EXP
        SUM             /SUBTRACT FPACC EXP FROM FPOP EXP
        JTS SHIFTO      /FPACC GREATER SO GO TO SHIFT OPERAND
        LCA             /FPOP GREATER - SAVE DIFFERENCE
MORACC, LLI 127         /POINTER TO FPACC EXP
        CAL SHLOOP      /CALL SHIFT LOOP
        DCC             /DECREMENT DIFFERENCE COUNTER
        JFZ MORACC      /CONTINUE ALIGNING IF NOT DONE
        JMP SHACOP      /SET UP FOR ADD OPERATION
SHIFTO, LCA             /SHIFT FPOP RTN, SAVE DIFF CNT (NEG VAL)
MOROP,  LLI 137         /SET POINTER TO FPOP EXPONENT
        CAL SHLOOP      /CALL SHIFT LOOP
        INC             /INCREMENT DIFFERENCE COUNTER
        JFZ MOROP       /SHIFT AGAIN IF NOT DONE
SHACOP, LLI 127         /SHIFT FPACC RIGHT ONCE - SET POINTER
        CAL SHLOOP      /CALL SHIFT LOOP
        LLI 137         /SHIFT FPOP RIGHT ONCE - SET POINTER
        CAL SHLOOP      /CALL SHIFT LOOP
        LDH             /SET UP POINTERS - "D" = 0 FOR SURE
        LEI 123         /POINTER TO LSW OF FPACC
        LBI 004         /SET PRECISION COUNTER
        CAL ADDER       /ADD FPACC TO FPOP QUAD-PRECISION
        LBI 000         /SET "B" FOR STANDARD NORMALIZATION
```

MNEMONIC		COMMENTS
	CAL FPNORM	/NORMALIZE THE RESULT OF THE ADDITION
	RET	/EXIT FPADD RTN WITH RESULT IN FPACC
SHLOOP,	LBM	/SHIFTING LOOP FOR ALIGNMENT
	INB	/FETCH EXPONENT INTO "B" AND INCREMENT IT
	LMB	/RETURN INCREMENTED VALUE TO MEMORY
	DCL	/DECREMENT THE POINTER
	LBI 004	/SET A COUNTER
	CAL FSHIFT	/CALL SPECIAL SHIFT ROUTINE
	RET	/EXIT "SHLOOP"
FSHIFT,	LAM	/GET MSW OF FLOATING-POINT NUMBER
	NDA	/SET FLAGS AFTER LOADING OPERATION
	JTS BRING1	/IF # IS MINUS, NEED TO SHIFT IN A "1"
	CAL ROTATR	/OTHERWISE PERFORM NTH-PRECISION ROTATE
	RET	/EXIT "FSHIFT"
BRING1,	RAL	/SAVE "1" IN CARRY BIT
	CAL ROTR	/DO ROTATE WITHOUT CLEARING CARRY BIT
	RET	/EXIT "FSHIFT"
MOVEIT,	LAM	/FETCH A WORD FROM MEMORY STRING "A"
	INL	/ADVANCE "A" STRING POINTER
	CAL SWITCH	/SWITCH POINTERS TO STRING "B"
	LMA	/PUT WORD FROM STRING "A" INTO STRING "B"
	INL	/ADVANCE "B" STRING POINTER
	CAL SWITCH	/SWITCH POINTERS BACK TO STRING "A"
	DCB	/DECREMENT COUNTER
	RTZ	/RETURN TO CALLING RTN WHEN COUNTER = 0
	JMP MOVEIT	/OTHERWISE CONTINUE MOVING OPERATION

FLOATING-POINT SUBTRACTION

NOW THAT ONE HAS A FLOATING-POINT ADDITION ROUTINE, FLOATING-POINT SUBTRACTION IS A "SNAP." ALL ONE REALLY HAS TO DO IS NEGATE THE NUMBER IN THE "FPACC" AND JUMP TO THE FLOATING-POINT ADDITION ROUTINE!

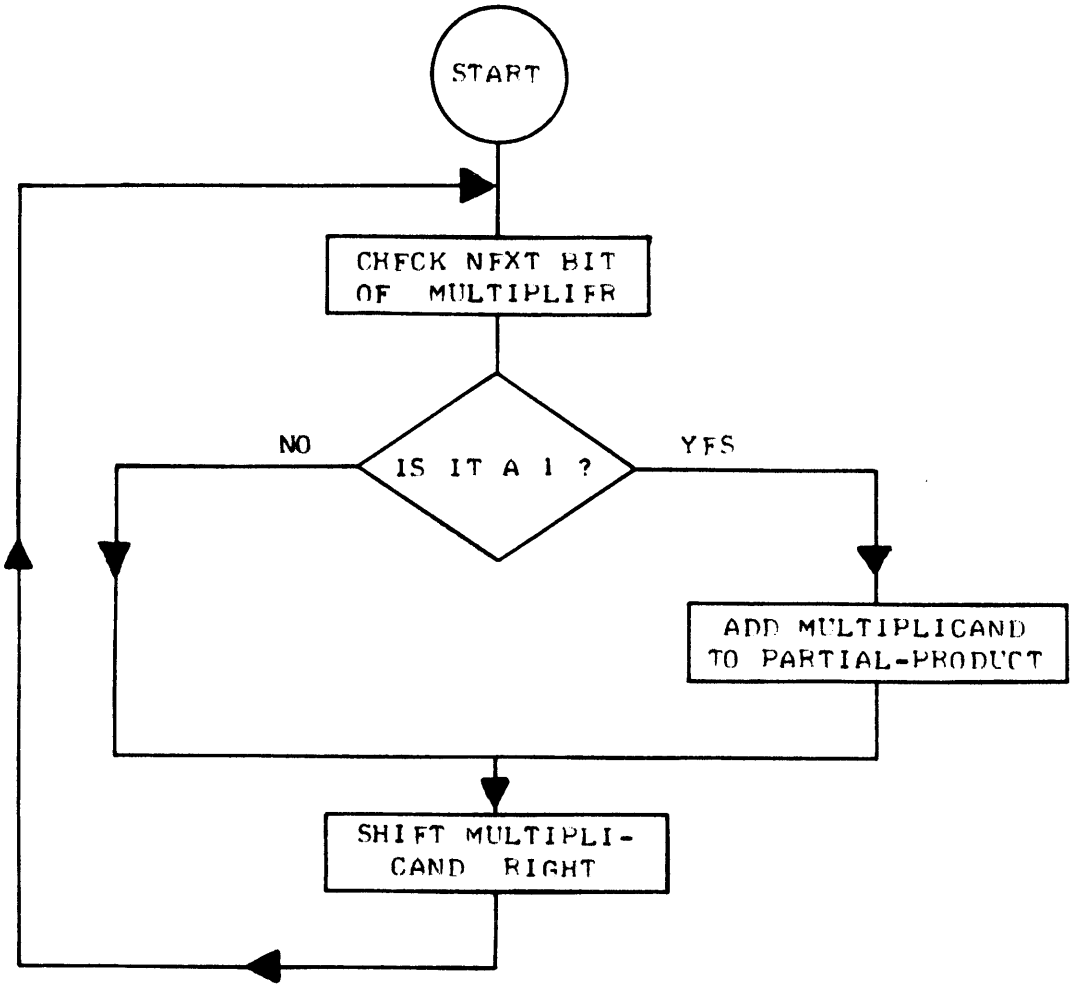
MNEMONIC		COMMENTS
FSUB,	LLI 123	/SET POINTER TO LSW OF FPACC
	LBI 004	/SET PRECISION COUNTER
	CAL COMPLM	/PERFORM TWO'S COMPLEMENT OF FPACC
	JMP FPADD	/SUBTRACTION ACCOMPLISHED NOW BY ADDING!

FLOATING-POINT MULTIPLICATION

FLOATING-POINT MULTIPLICATION CAN BE ACCOMPLISHED BY UTILIZING A "SHIFTING AND ADDING" ALGORITHM FOR THE MANTISSA PORTION OF THE NUMBERS. AS POINTED OUT EARLIER, SHIFTING A BINARY NUMBER TO THE RIGHT SERVES TO ESSENTIALLY "DOUBLE" IT'S VALUE. AN ALGORITHM THAT TAKES ADVANTAGE OF THAT FACT CAN BE DESCRIBED AS FOLLOWS.

CONSIDER THE TWO NUMBERS AS A "MULTIPLIER" AND A "MULTIPLICAND." EXAMINE THE LEAST SIGNIFICANT BIT OF THE "MULTIPLIER." IF IT IS A "1," ADD THE CURRENT VALUE OF THE "MULTIPLICAND" TO A THIRD REGISTER (WHICH INITIALLY STARTS WITH A VALUE OF ZERO). NOW, SHIFT THE MULTIPLICAND ONE

POSITION TO THE LEFT. EXAMINE THE NEXT BIT TO THE LEFT OF THE LEAST SIGNIFICANT BIT IN THE MULTIPLIER. IF IT IS A "1," ADD THE CURRENT VALUE OF THE "MULTIPLICAND" TO THE THIRD REGISTER (WHICH COULD BE CALLED THE "PARTIAL-PRODUCT" REGISTER). SHIFT THE MULTIPLICAND TO THE RIGHT AGAIN. CONTINUE THE PROCESS BY EXAMINING ALL THE BITS IN THE MULTIPLIER FOR A "1" CONDITION. WHENEVER THE MULTIPLIER CONTAINS A "1" ADD THE CURRENT VALUE OF THE MULTIPLICAND TO THE PARTIAL-PRODUCT REGISTER. AFTER EACH EXAMINATION OF A BIT IN THE MULTIPLIER (AND ADDITION OF THE MULTIPLIER TO THE PARTIAL-PRODUCT REGISTER IF A "1" WAS OBSERVED) SHIFT THE MULTIPLICAND RIGHT. CONTINUE UNTIL ALL BITS IN THE MULTIPLIER HAVE BEEN EXAMINED. THE RESULT OF THE MULTIPLICATION WILL BE IN THE PARTIAL-PRODUCTS REGISTER AT THE COMPLETION OF THE ABOVE PROCESS. THE ALGORITHM CAN PERHAPS BE SEEN A LITTLE MORE CLEARLY BY STUDYING THE FLOW-CHART PRESENTED BELOW.



THE READER CAN VERIFY THE ALGORITHM BY FOLLOWING THE EXAMPLE BELOW FOR TWO SMALL NUMBERS - THE NUMBER 3 (DECIMAL) AS THE MULTIPLICAND AND THE NUMBER 5 AS THE MULTIPLIER.

```
   00 000 011      (MULTIPLICAND AT START OF OPERATIONS)

   00 000 101      (MULTIPLIER)
 --------------
   00 000 000      (PARTIAL PRODUCT BEFORE OPERATIONS START)
```

```
  00 000 011      (MULTIPLICAND WHEN 1ST BIT OF MULTIPLIER
                        IS EXAMINED)
  00 000 101      (LEAST SIGNFICANT BIT OF MULTIPLIER = 1)
--------------
  00 000 011      (MULTIPLICAND IS ADDED TO PARTIAL-PRODUCT)


  00 000 110      (MULTIPLICAND IS SHIFTED TO THE RIGHT BE-
                     FORE SECOND BIT OF MULTIPLIER EXAMINED
  00 000 101      (SECOND BIT OF MULTIPLIER IS ZERO)
--------------
  00 000 011      (SO NOTHING IS ADDED TO PARTIAL-PRODUCT)


  00 001 100      (MULTIPLICAND IS SHIFTED TO RIGHT AGAIN
                  BEFORE NEXT BIT OF MULTIPLIER IS EXAMINED
  00 000 101      (THIRD BIT OF MULTIPLIER IS A "1")
--------------
  00 001 111      (SO MULTIPLICAND'S CURRENT VALUE IS ADDED
                  INTO THE PARTIAL-PRODUCT REGISTER.  SINCE
                  ALL THE REMAINING BITS IN THE MULTIPLIER
                  ARE "0" NOTHING MORE WILL BE ADDED TO THE
                  PARTIAL-PRODUCT REGISTER WHICH THUS HOLDS
                  THE FINAL ANSWER!)
```

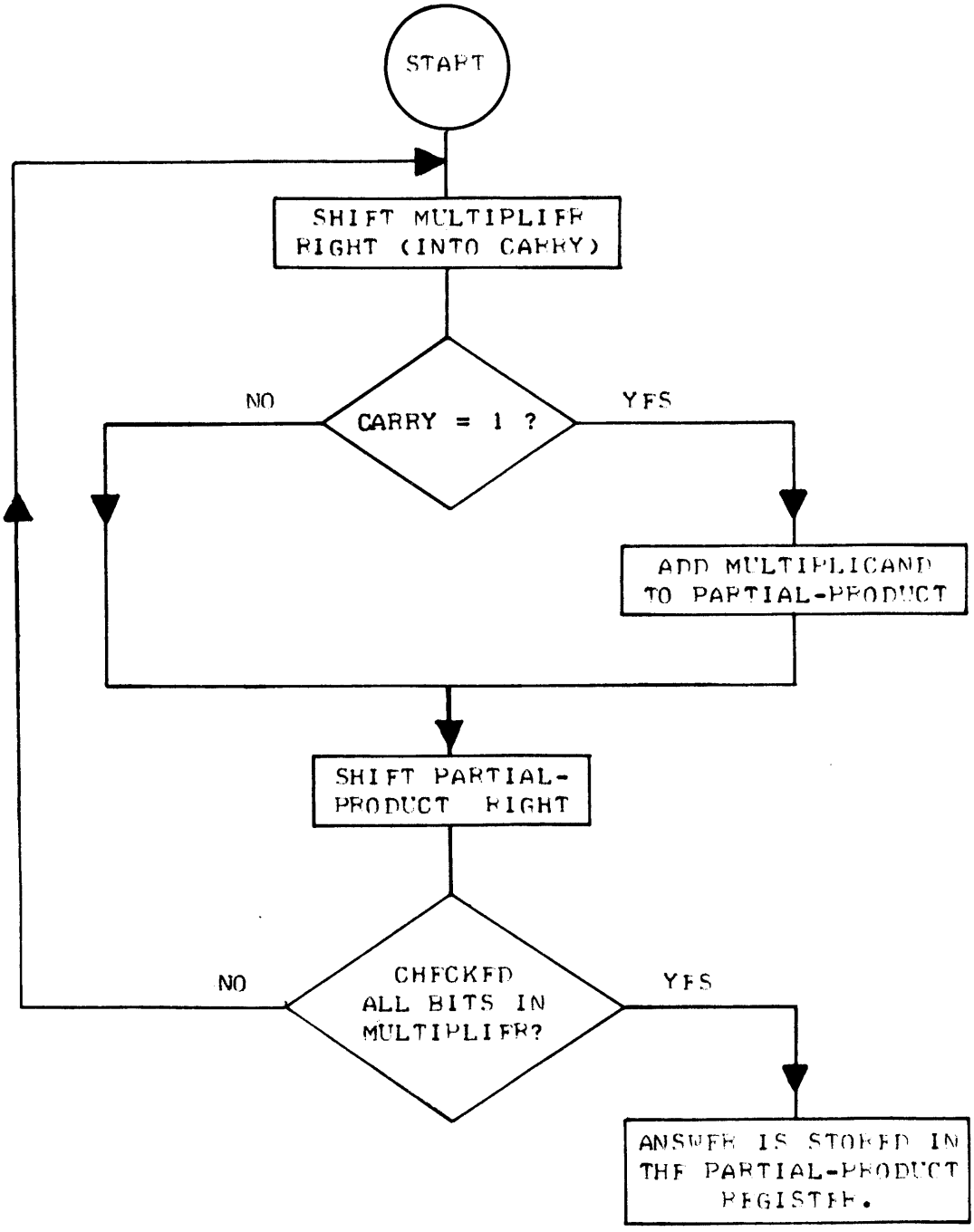
WHILE THE ABOVE ALGORITHM WAS PRESENTED FOR HANDLING NUMBERS IN REGULAR FORMAT, WITH JUST A LITTLE VARIATION, THE BASIC PROCEDURE CAN BE IMPLEMENTED FOR MULTIPLYING THE MANTISSA PORTION OF NUMBERS STORED IN FLOATING-POINT FORMAT. A FLOW CHART OF THE MANTISSA MULTIPLYING PORTION OF THE "FPMULT" ROUTINE TO BE PRESENTED SHORTLY IS SHOWN ON THE NEXT PAGE. NOTE THAT IT IS EASY TO TEST EACH BIT OF THE "MULTIPLIER" BY SIMPLY ROTATING IT RIGHT AND TESTING THE STATUS OF THE CARRY FLAG AFTER THE ROTATE OPERATION!

HANDLING THE EXPONENT PORTION WHEN MULTIPLYING TWO NUMBERS STORED IN BINARY FLOATING-POINT FORMAT IS ACCOMPLISHED THE SAME WAY ONE WOULD HANDLE EXPONENTS IN DECIMAL FLOATING-POINT FORMAT. THE EXPONENTS ARE SIMPLY ADDED TOGETHER.

THERE ARE SEVERAL OTHER PARAMETERS TO CONSIDER WHEN MULTIPLYING NUMBERS. FIRST, THE ALGORITHM PRESENTED MAY ONLY BE USED WHEN THE NUMBERS ARE POSITIVE IN VALUE. THUS, ANY NEGATIVE NUMBERS MUST FIRST BE "NEGATED" BEFORE USING THE ALGORITHM. FURTHERMORE, THE READER KNOWS THAT IF TWO NUMBERS OF THE SAME "SIGN" ARE MULTIPLIED TOGETHER THE ANSWER WILL BE A POSITIVE VALUE, BUT IF THE "SIGNS" ARE DIFFERENT, THE ANSWER WILL BE A NEGATIVE NUMBER. THEREFORE, ONE MUST KEEP ACCOUNT OF THE INITIAL "SIGNS" OF THE NUMBERS BEING MULTIPLIED, AND IF THE ANSWER DICTATES, THE FINAL VALUE MUST BE NEGATED AFTER USING THE ALGORITHM. AS THE READER WILL OBSERVE IN THE "FPMULT" ROUTINE, HANDLING THIS TASK IS QUITE EASY.

SECONDLY, THE ALERT READER MAY HAVE OBSERVED THAT SINCE THE MULTIPLICAND IS SHIFTED IN THE ABOVE ALGORITHM (THE PARTIAL-PRODUCT REGISTER IS SHIFTED IN THE FLOATING-POINT ALGORITHM TO ACCOMPLISH THE SAME PURPOSE) ONE POSITION FOR EACH BIT IN THE MULTIPLIER, THEN IT IS NECESSARY TO MAINTAIN "WORKING" REGISTERS THAT ARE TWICE AS LONG AS THE ORIGINAL NUMBERS TO BE MULTIPLIED. THUS, THE FINAL ANSWER MAY CONTAIN MORE BITS OF PRECISION THEN THE OVER-ALL PROGRAM IS DESIGNED TO HANDLE. IN THE "FPMULT" ROUTINE, THE MULTIPLICATION OF THE MANTISSAS IS ACCOMPLISHED USING SIX MEMORY WORDS PER REGISTER. BUT, AT THE CONCLUSION OF THE ROUTINE, THE 23RD BINARY BIT IS "ROUNDED" OFF (DEPENDING ON THE STATUS OF

THE 24TH LEAST SIGNIFICANT BIT) AND THE ANSWER IS NORMALIZED BACK TO A 23 BIT BINARY NUMBER WHICH IS THE LARGEST NUMBER THE PACKAGE BEING DISCUSSED IS DESIGNED TO HANDLE. THE METHOD ALLOWS MAXIMUM PRECISION TO BE MAINTAINED DURING THE MULTIPLICATION PROCESS.



FLOATING-POINT MULTIPLICATION ALGORITHM FLOW CHART

```
        MNEMONIC                    COMMENTS
     -------------              -------------

FPMULT, CAL CKSIGN      /SET UP ROUTINE AND CHECK SIGN OF #'S
ADDEXP, LLI 137         /SET POINTER TO FPOP EXPONENT
        LAM             /FETCH FPOP EXPONENT INTO ACCUMULATOR
        LLI 127         /SET POINTER TO FPACC EXPONENT
        ADM             /ADD FPACC EXP TO FPOP EXP
```

```
                MNEMONIC                    COMMENTS
        ----------------------      ----------------------

                ADI 001             /ADD ONE FOR ALGORITHM COMPENSATION
                LMA                 /STORE RESULT IN FPACC EXPONENT
SETMCT,         LLI 102             /SET BIT COUNTER STORAGE POINTER
                LMI 027             /SET BIT CNTR TO 23 DECIMAL (27 OCTAL)
MULTIP,         LLI 126             /BASIC "X" ALGORITHM - PNTR TO MSW FPACC
                LBI 003             /SET PRECISION COUNTER
                CAL ROTATR          /ROTATE MULTIPLIER RIGHT INTO CARRY FLAG
                CTC ADOPPP          /IF CARRY=1, ADD M'CAND TO PARTIAL-PROD
                LLI 146             /SET PNTR TO PARTIAL-PRODUCT MSW
                LBI 006             /SET PRECISION COUNTER
                CAL ROTATR          /SHIFT PARTIAL-PRODUCT RIGHT
                LLI 102             /SET POINTER TO BIT COUNTER
                CAL CNTDWN          /DECREMENT VALUE IN BIT COUNTER
                JFZ MULTIP          /IF BIT CNTR NOT ZERO, REPEAT ALGORITHM
                LLI 146             /SET POINTER TO PARTIAL-PRODUCT MSW
                LBI 006             /SET PRECISION COUNTER - ROTATE P/P ONCE
                CAL ROTATR          /MORE TO MAKE ROOM FOR POSSIBLE ROUNDING
                LLI 143             /SET PNTR TO ACCESS 24TH BIT IN P/P
                LAM                 /FETCH 24TH BIT
                RAL                 /POSITION IT TO MSB POSITION
                RAL                 /   "     "  "   "     "
                NDA                 /SET FLAGS AFTER ROTATE OPERATION
                CTS MROUND          /IF 24TH BIT = 1, DO ROUNDING PROCEDURE
                LLI 123             /NOW SET PNTR TO FPACC
                CAL SWITCH          /SAVE FPACC POINTER
                LHD                 /ENSURE "H" IS 000
                LLI 143             /SET POINTER TO PARTIAL-PRODUCT
                LBI 004             /SET PRECISION COUNTER
EXMLDV,         CAL MOVEIT          /MOVE ANSWER FROM P/P INTO FPACC
                LBI 000             /SET "B" FOR STANDARD NORMALIZATION
                CAL FPNORM          /NORMALIZE THE ANSWER
                LLI 101             /SET POINTER TO "SIGNS" INDICATOR
                LAM                 /FETCH "SIGNS" INDICATOR
                NDA                 /SET FLAGS AFTER LOAD OPERATION
                RFZ                 /IF "SIGNS" HAS VALUE, RESULT IS +, EXIT
                LLI 124             /BUT IF "SIGNS" = 0, SET FPACC LSW PNTR
                LBI 003             /AND PRECISION COUNTER
                CAL COMPLM          /AND NEGATE THE ANSWER
                RET                 /BEFORE EXITING "FPMULT" ROUTINE
CKSIGN,         CAL CLRWRK          /CLEAR WORKING LOC'S FOR MULTIPLICATION
                LLI 101             /SET POINTER TO "SIGNS" STORAGE
                LMI 001             /PLACE THE INITIAL VALUE "1" IN "SIGNS"
                LLI 126             /SET POINTER TO MSW OF FPACC
                LAM                 /FETCH MSW OF FPACC
                NDA                 /SET FLAGS AFTER LOAD OPERATION
                JTS NEGFPA          /IF # IS MINUS, NEED TO DO 2'S COMPLEMENT
OPSGNT,         LLI 136             /SET POINTER TO MSW OF FPOP
                LAM                 /FETCH MSW OF FPOP
                NDA                 /SET FLAGS AFTER LOAD OPERATION
                RFS                 /IF # IS +, RETURN TO CALLING ROUTINE
                LLI 101             /IF # MINUS, SET POINTER TO "SIGNS"
                CAL CNTDWN          /DECREMENT VALUE IN "SIGNS"
                LLI 134             /SET POINTER TO LSW OF FPOP
                LBI 003             /SET PRECISION COUNTER
                CAL COMPLM          /PERFORM TWO'S COMPLEMENT OF # IN FPOP
                RET                 /GO BACK TO CALLING ROUTINE
NEGFPA,         LLI 101             /SET POINTER TO "SIGNS" STORAGE
                CAL CNTDWN          /DECREMENT VALUE OF "SIGNS"
```

```
          LLI 124      /SET POINTER TO LSW OF FPACC
          LBI 003      /SET PRECISION COUNTER
          CAL COMPLM   /NEGATE THE VALUE IN THE FPACC
          JMP OPSCNT   /GO CHECK SIGN OF FPOP
CLRWRK.   LLI 140      /CLEAR PART-PROD'S WORK AREA (140-147)
          LBI 010      /SET POINTER AND COUNTER
          XRA          /SET ACCUMULATOR = 000
CLRNEX.   LMA          /DEPOSIT ACCUMULATOR CONTENTS INTO MEM
          DCB          /DECREMENT COUNTER
          JTZ CLROPL   /WHEN DONE GO TO NEXT AREA
          INL          /ELSE CONTINUE CLEARING P/P WORKING AREA
          JMP CLRNEX   /BY STUFFING 000 IN NEXT MEM LOCATION
CLROPL.   LBI 004      /CLEAR ADDITIONAL ROOM FOR MULTIPLICAND
          LLI 130      /AT 130 TO 133 - SET COUNTER & POINTER
CLRNX1.   LMA          /PUT 000 IN MEMORY
          DCB          /DECREMENT COUNTER
          RTZ          /RETURN TO CALLING PROGRAM WHEN DONE
          INL          /ELSE ADVANCE POINTER
          JMP CLRNX1   /AND CONTINUE CLEARING OPERATIONS
ADDPPP.   LEI 141      /POINTER TO LSW OF PARTIAL-PRODUCT
          LDM          /GM PG 00 IN "D & E"
          LLI 131      /PNTR TO LSW OF MULTIPLICAND
          LBI 006      /SET PRECISION COUNTER
          CAL ADDER    /PERFORM ADDITION
          RET
MROUND.   LBI 003      /SET PRECISION COUNTER
          LAI 100      /ADD "1" TO 23'RD BIT OF PARTIAL-PROD
          ADM          /HERE
GROUND.   LMA          /RESTORE TO MEMORY
          INL          /ADVANCE POINTER
          LAI 000      /CLEAR ACC WITHOUT DISTURBING CARRY
          ACM          /AND PROPOGATE ROUNDING
          DCB          /IN PARTIAL-PRODUCT
          JFZ GROUND   /FINISHED WHEN CNTR = 000
          LMA          /RESTORE LAST WORD OF P-P
          RET
```

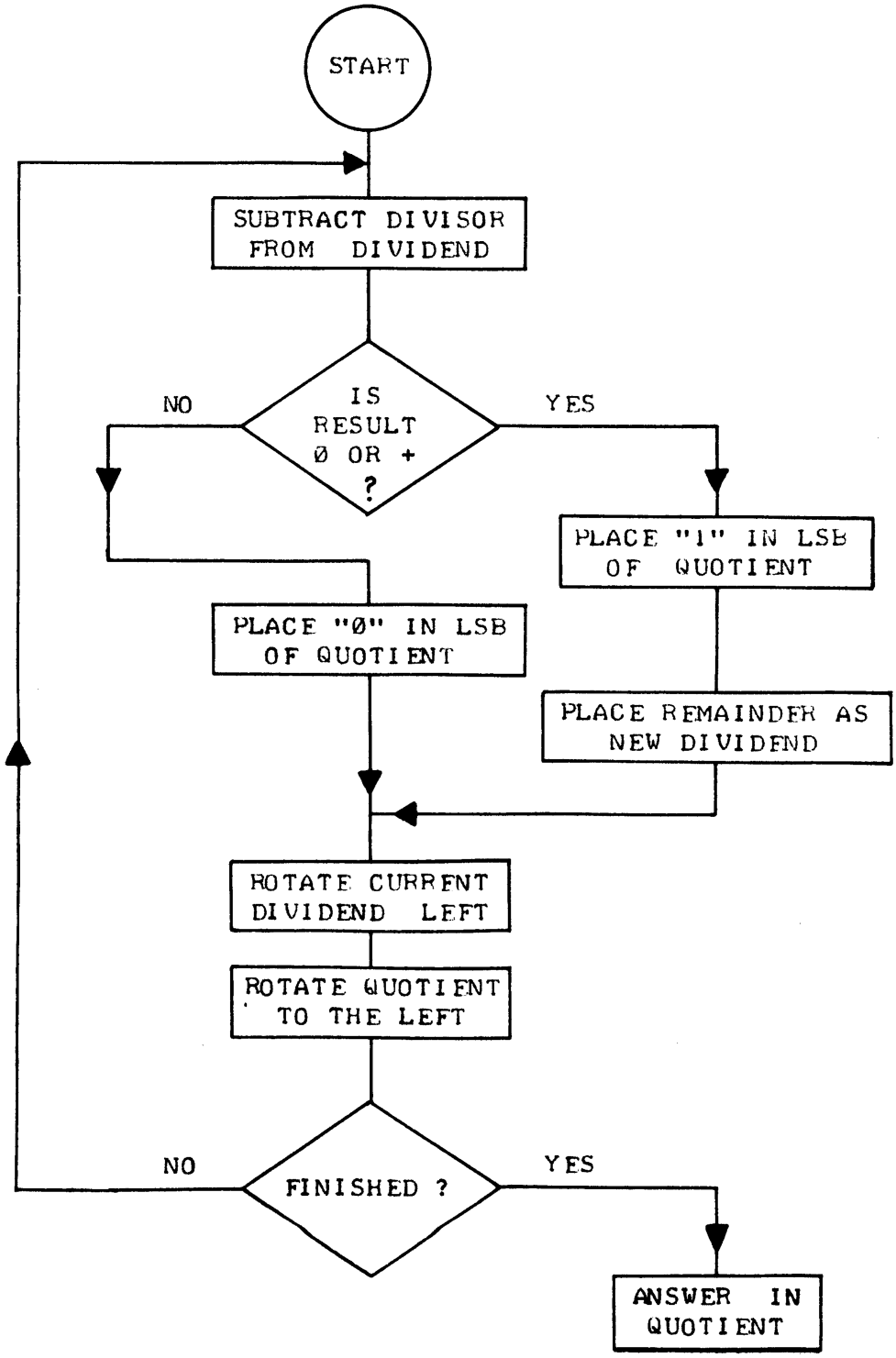
FLOATING-POINT DIVISION

IN A MANNER THAT IS SORT OF THE REVERSE OF MULTIPLICATION (WHICH USED ADDITION AND ROTATE OPERATIONS) ONE CAN PERFORM DIVISION USING AN ALGORITHM THAT UTILIZES SUBTRACTION AND ROTATE OPERATIONS. THE ALGORITHM WILL BE PRESENTED DIRECTLY IN THE FORM USED IN FLOATING-POINT OPERATIONS BECAUSE IN THIS CASE IT IS SIMPLER THAN DESCRIBING IT FOR NUMBERS THAT ARE NOT IN FLOATING-POINT FORM. HOWEVER, THE ALERT READER SHOULD HAVE LITTLE DIFFICULTY OBSERVING THAT THE ALGORITHM COULD BE USED FOR NUMBERS THAT ARE NOT IN FLOATING-POINT FORMAT IF ONE FIRST ALIGNED THE MOST SIGNIFICANT BITS OF THE DIVISOR AND DIVIDEND, AND TOOK APPROPRIATE ACTION TO HANDLE THE LOCATION OF A "BINARY" POINT FOR CASES WHERE THE RESULT WAS NOT A PURE INTEGER.

IN RAMBLING ENGLISH, THE ALGORITHM COULD BE STATED AS FOLLOWS: SUBTRACT THE VALUE OF THE DIVISOR FROM THE VALUE OF THE ORIGINAL DIVIDEND. TEST THE RESULT OF THE SUBTRACTION. IF THE RESULT IS NEGATIVE, MEANING THE ENTIRE DIVISOR COULD NOT BE SUBTRACTED, PLACE A "0" IN THE LEAST SIGNIFICANT BIT OF A REGISTER TERMED THE "QUOTIENT." LEAVE THE CURRENT DIVIDEND ALONE. IF THE RESULT OF THE SUBTRACTION IS POSITIVE, OR ZERO, INDICATING THE DIVIDEND WAS LARGER THAN THE DIVISOR, PLACE A "1" IN THE LEAST SIGNIFICANT BIT OF THE "QUOTIENT" REGISTER AND CHANGE THE DIVIDEND TO BE THE VALUE OF THE "REMAINDER" (OR RESULT) OF THE SUBTRACTION OPERATION. NEXT, ONCE THE APPROPRIATE ACTION HAS BEEN TAKEN AS A

FUNCTION OF THE RESULT OF THE SUBTRACTION OPERATION, ROTATE THE CONTENTS OF THE DIVIDEND (WHETHER IT'S ORIGINAL VALUE OR THE NEW "REMAINDER") ONE POSITION TO THE RIGHT, AND SIMILARLY ROTATE THE QUOTIENT ONCE TO THE RIGHT TO ALLOW ROOM FOR THE NEXT LEAST SIGNIFICANT BIT. NOW REPEAT THE ENTIRE PROCEDURE UNTIL ONE HAS PERFORMED THE ABOVE OPERATIONS AS MANY TIMES AS THERE ARE BIT POSITIONS IN THE REGISTER USED TO HOLD THE ORIGINAL DIVIDEND! (THAT WOULD BE 23 (DECIMAL) TIMES FOR THE FLOATING-POINT PACKAGE BEING DISCUSSED HERE.)

THE ALGORITHM MAY BE VISUALIZED A LITTLE MORE CLEARLY BY STUDYING THE FLOW CHART PRESENTED BELOW. ADDITIONALLY, A STEP-BY-STEP PRESENTATION ILLUSTRATING THE ALGORITHM BEING USED TO DIVIDE THE BINARY EQUIVALENT OF 15 (DECIMAL) BY 5 IS PRESENTED ON THE NEXT PAGE. THE LENGTH OF THE REGISTERS HAVE BEEN REDUCED TO SHORTEN THE ILLUSTRATION. REMEMBER, THE ALGORITHM SHOWN IS FOR THE MANTISSA PORTION OF NUMBERS ALREADY STORED IN "NORMALIZED" FLOATING-POINT FORMAT.



```
      0 . 1 1 1 1     ORIGINAL DIVIDEND AT START OF ROUTINE

      0 . 1 0 1 0     DIVISOR (NOTE FLOATING-POINT FORMAT!)
    ---------------
      0 . 0 1 0 1     RESULT OF FIRST SUBTRACTION OPERATION
                      THIS IS THE "REMAINDER" FROM THE SUB-
                      TRACTION OPERATION.  SINCE RESULT WAS
                      "POSITIVE" A "1" IS PLACED IN THE LSB
                      OF THE QUOTIENT REGISTER:

                      0 . 0 0 0 1  QUOTIENT AFTER 1ST LOOP

NOW BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER) ARE ROTATED LEFT

      0 . 1 0 1 0     NEW DIVIDEND (WHICH IS THE LAST REMAIN-
                      DER ROTATED ONCE TO THE LEFT)
      0 . 1 0 1 0     DIVISOR (DOES NOT CHANGE DURING ROUTINE)
    ---------------
      0 . 0 0 0 0     RESULT OF THIS SUBTRACTION IS ZERO AND
                      THUS QUALIFIES TO BECOME NEW DIVIDEND.
                      QUOTIENT LSB GETS A "1" FOR THIS CASE!

                      0 . 0 0 1 1  QUOTIENT AFTER 2ND LOOP

AGAIN BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER) ARE ROTATED LEFT

      0 . 0 0 0 0     NEW DIVIDEND (WHICH IS THE LAST REMAIN-
                      DER ROTATED ONCE TO THE LEFT)
      0 . 1 0 1 0     DIVISOR - STILL SAME OLD NUMBER
    ---------------
      1 . 0 1 1 0     RESULT OF THIS SUBTRACTION IS A MINUS
                      NUMBER (NOTE THAT THE "SIGN" BIT CHANG-
                      ED!)  THUS, OLD DIVIDEND STAYS IN PLACE
                      AND QUOTIENT GETS A "0" IN LSB POSITION.

                      0 . 0 1 1 0  QUOTIENT AFTER 3RD LOOP

NOW BOTH QUOTIENT, AND IN THIS CASE THE OLD DIVIDEND ARE ROTATED LEFT

      0 . 0 0 0 0     OLD DIVIDEND ROTATED ONCE TO THE LEFT

      0 . 1 0 1 0     SAME OLD DIVISOR
    ---------------
      1 . 0 1 1 0     RESULT OF THIS SUBTRACTION IS AGAIN A
                      MINUS.  OLD DIVIDEND STAYS IN PLACE.
                      QUOTIENT GETS ANOTHER "0" IN LSB.

                      0 . 1 1 0 0  QUOTIENT AFTER 4TH LOOP
```

SINCE THERE WERE JUST 4 BITS IN THE MULTIPLICAND REGISTER, THE ALGORITHM WOULD BE COMPLETED AT THE END OF THE FOURTH LOOP AND THE ANSWER WOULD BE THAT SHOWN IN THE QUOTIENT. REMEMBER, THAT SINCE FLOATING-POINT FORMAT IS BEING USED, THAT THERE WOULD BE BINARY EXPONENTS INVOLVED. SIMILAR TO THE WAY ONE WOULD HANDLE EXPONENTS IN DECIMAL FLOATING-POINT NOTATION, ONE SUBTRACTS THE EXPONENTS FOR THE TWO NUMBERS (DIVISOR EXPONENT FROM THE DIVIDEND EXPONENT) TO OBTAIN THE EXPONENT VALUE FOR A DIVISION OPERATION. IN THE ABOVE EXAMPLE, THE MULTIPLICAND WOULD HAVE HAD THE BINARY EXPONENT 4 (DECIMAL) TO REPRESENT THE NORMALIZED STORING OF 15 AND THE DIVISOR WOULD HAVE HAD A BINARY EXPONENT OF 3. THE ABOVE ALGORITHM REQUIRES A COMPENSATION FACTOR OF + 1 AFTER SUBTRACTING THE

EXPONENTS (CAN THE READER THINK OF WAYS IN WHICH THIS COULD BE AVOIDED?) IN ORDER TO HAVE THE CORRECT FLOATING-POINT RESULT. IN THE SAMPLE ILLUSTRATED ABOVE, (4 - 3) + 1 = 2, AND INDEED IF THE ANSWER SHOWN WAS MOVED TWO PLACES TO THE LEFT (OF THE IMPLIED "BINARY POINT") ONE CAN VERIFY THAT THE RESULT IS THE BINARY EQUIVELANT OF 3 (DECIMAL)! THE READER MIGHT WANT TO TRY USING OTHER SMALL VALUED NUMBERS TO TEST THE VALIDITY OF THE ALGORITHM AND DEVELOP A THOROUGH UNDERSTANDING OF THE PROCESS. A GOOD CASE TO EXAMINE IS ONE WHERE THE RESULT IS "NON-ENDING" SUCH AS THE NUMBER 1 DIVIDED, SAY, BY 3.

JUST AS IN THE MULTIPLICATION ROUTINE, THERE ARE SEVERAL OTHER PARAMETERS THAT MUST BE CONSIDERED IN DEVELOPING THE DIVISION ROUTINE. FOR INSTANCE, THERE IS AGAIN THE MATTER OF THE SIGNS OF THE NUMBERS. THE ALGORITHM REQUIRES THAT THE NUMBERS BE IN POSITIVE FORMAT SO AGAIN ONE MUST KEEP TRACK OF THE SIGNS OF THE ORIGINAL NUMBERS AND CONVERT ANY NEGATIVE ONES TO POSITIVE FOR THE ROUTINE. IF THE SIGNS OF THE TWO NUMBERS INVOLVED ARE IDENTICAL, THE RESULT MUST BE POSITIVE, IF THEY ARE DIFFERENT, THEN THE PROGRAM MUST NEGATE THE ANSWER OBTAINED FROM THE ACTUAL DIVISION PROCESS. AND, BECAUSE SOME CALCULATIONS WILL RESULT IN A NON-ENDING SERIES FOR AN ANSWER, SOME "ROUNDING" CAPABILITY MUST BE INCLUDED IN THE PROGRAM. THEN, THERE IS A SPECIAL CASE IN DIVISION THAT ONE MUST CHECK FOR: DIVISION BY ZERO! IN THAT CASE THE PROGRAM MIGHT BRANCH OFF TO TELL THE OPERATOR A THING OR TWO. THE FLOATING-POINT DIVISION ROUTINE SHOWN BELOW CONSIDERS THESE MATTERS AS THE READER WILL OBSERVE.

```
                MNEMONIC                    COMMENTS
            ----------------            ----------------

FPDIV,   CAL CKSIGN   /SET UP REG'S AND CHECK SIGN OF NUMBERS
         LLI 126      /SET POINTER TO MSW OF FPACC (DIVISOR)
         LAI 000      /CLEAR ACCUMULATOR
         CPM          /SEE IF MSW OF FPACC = ZERO
         JFZ SUBEXP   /IF FIND ANYTHING - PROCEED TO DIVIDE
         DCL          /DECREMENT POINTER
         CPM          /SEE IF NSW OF DIVISOR = ZERO
         JFZ SUBEXP   /IF FIND ANYTHING - PROCEED TO DIVIDE
         DCL          /DECREMENT POINTER
         CPM          /SEE IF LSW OF DIVISOR = ZERO
         JTZ DERROR   /IF DIVISOR = ZERO, TELL SOMEBODY!
SUBEXP,  LLI 137      /SET POINTER TO DIVIDEND (FPOP) EXPONENT
         LAM          /FETCH DIVIDEND EXPONENT
         LLI 127      /SET POINTER TO DIVISOR (FPACC) EXPONENT
         SUM          /SUBTRACT DIVISOR EXP FM DIVIDEND EXP
         ADI 001      /COMPENSATE FOR DIVISION ALGORITHM
         LMA          /STORE EXPONENT RESULT IN FPACC EXP
SETDCT,  LLI 102      /SET POINTER TO BIT COUNTER STORAGE
         LMI 027      /SET IT TO 27 OCTAL (23 DECIMAL)
DIVIDE,  CAL SETSUB   /MAIN DIVISION RTN - SUB DIVIS FM DIVID
         JTS NOGO     /IF RESULT IS NEGATIVE - PUT 0 IN QUOT
         LEI 134      /IF + OR 0, MOVE REMAINDER INTO DIVIDEND
         LLI 131      /SET POINTERS
         LBI 003      /AND PRECISION COUNTER
         CAL MOVEIT   /AND MOVE REMAINDER INTO DIVIDEND
         LAI 001      /PUT A "1" INTO ACCUMULATOR
         RAR          /AND MOVE IT INTO THE CARRY BIT
         JMP QUOROT   /PROCEED TO ROTATE IT INTO THE QUOTIENT
NOGO,    LAI 000      /WHEN RESULT IS NEG, PUT "0" INTO ACC
         RAR          /AND MOVE IT INTO CARRY BIT
QUOROT,  LLI 144      /SET POINTER TO LSW OF QUOTIENT
```

```
                MNEMONIC                COMMENTS
                --------                --------

                LBI 003         /SET PRECISION COUNTER
                CAL ROTL        /MOVE CARRY BIT INTO LSB OF QUOTIENT
                LLI 134         /SET POINTER TO DIVIDEND LSW
                LBI 003         /SET PRECISION COUNTER
                CAL ROTATL      /ROTATE DIVIDEND LEFT
                LLI 102         /SET POINTER TO BITS COUNTER
                CAL CNTDWN      /DECREMENT BITS COUNTER
                JFZ DIVIDE      /IF NOT FINISHED - CONTINUE ALGORITHM
                CAL SETSUB      /DO ONE MORE DIVIDE FOR ROUNDING OPS
                JFS DVEXIT      /24TH BIT = 0, NO ROUNDING
                LLI 144         /24TH BIT = 1, SET PNTR TO QUOTIENT LSW
                LAM             /FETCH LSW OF QUOTIENT
                ADI 001         /ADD "1" TO 23RD BIT
                LMA             /RESTORE LSW
                LAI 000         /CLEAR ACCUMULATOR WHILE SAVING CARRY
                INL             /ADVANCE POINTER TO NSW OF QUOTIENT
                ACM             /ADD WITH CARRY
                LMA             /RESTORE NSW
                LAI 000         /CLEAR ACCUMULATOR WHILE SAVING CARRY
                INL             /ADVANCE POINTER TO MSW OF QUOTIENT
                ACM             /ADD WITH CARRY
                LMA             /RESTORE MSW
                JFS DVEXIT      /IF MSB OF MSW = 0, PREPARE TO EXIT
                LBI 003         /OTHERWISE SET PRECISION COUNTER
                CAL ROTATR      /MOVE QUOT RIGHT TO CLEAR SIGN BIT
                LLI 127         /SET POINTER TO FPACC EXPONENT
                LBM             /FETCH EXPONENT
                INL             /INCREMENT IT FOR ROTATE RIGHT OP ABOVE
                LMB             /RESTORE EXPONENT
DVEXIT,         LLI 144         /SET POINTERS TO TRANSFER
                LEI 124         /QUOTIENT TO FPACC
                LBI 003         /SET PRECISION COUNTER
                JMP EXMLDV      /EXIT THRU FPMULT RTN AT "EXMLDV"
SETSUB,         LLI 131         /SET PNTR TO LSW OF WORKING REGISTER
                CAL SWITCH      /SAVE POINTER
                LHD             /SET H = 0 FOR SURE
                LLI 124         /SET POINTER TO LSW FPACC
                LBI 003         /SET PRECISION COUNTER
                CAL MOVEIT      /MOVE FPACC VALUE TO WORKING REGISTER
                LEI 131         /RESET PNTR TO WORKING REG'S LSW (DIVISOR)
                LLI 134         /SET PNTR TO LSW OF FPOP (DIVIDEND)
                LBI 003         /SET PRECISION COUNTER
                CAL SUBBER      /SUBTRACT DIVISOR FROM DIVIDEND
                LAM             /GET MSW OF RESULT FROM SUBTRACTION OPS
                NDA             /AND SET FLAGS AFTER LOAD OPERATION
                RET             /BEFORE RETURNING TO CALLING ROUTINE
DERROR,         CAL DERMSG      /**USER DEFINED ERROR ROUTINE FOR ATTEMPT-
                JMP USERDF      /ING DIVISION BY 0 - EXIT AS DIRECTED**
```

THE FIVE FUNDAMENTAL FLOATING-POINT ROUTINES, "FPNORM," "FPADD," "FPSUB," "FPMULT" AND "FPDIV," WHEN ASSEMBLED INTO OBJECT CODE WILL FIT WITHIN THREE PAGES OF MEMORY IN AN 8008 SYSTEM. ADDITIONALLY, THE ROUTINES AS PRESENTED IN THIS CHAPTER USE SOME SPACE ON PAGE 00 FOR STORING DATA AND COUNTERS. NEEDLESS TO SAY, THE PROGRAMS AS DEVELOPED FOR DISCUSSION COULD BE MODIFIED TO USE OTHER MEMORY LOCATIONS WITH LITTLE DIFFICULTY. FOR REFERENCE PURPOSES, THE LOCATIONS USED ON PAGE 00 BY THE FUNDAMENTAL FLOATING-POINT ROUTINES JUST PRESENTED ARE LISTED

HERE:

LOCATION(S)	USAGE
100	SIGN INDICATOR
101	SIGNS INDICATOR (MULT & DIVIDE)
102	BITS COUNTER
123	FPACC EXTENSION
124	FPACC LEAST SIGNIFICANT WORD
125	FPACC NEXT SIGNIFICANT WORD
126	FPACC MOST SIGNIFICANT WORD
127	FPACC EXPONENT
130 - 133	WORKING AREA
134	FPOP LEAST SIGNIFICANT WORD
135	FPOP NEXT SIGNIFICANT WORD
136	FPOP MOST SIGNIFICANT WORD
137	FPOP EXPONENT
140 - 147	WORKING AREA

THE FUNDAMENTAL FLOATING-POINT ROUTINES WHICH HAVE BEEN PRESENTED AND DISCUSSED ARE EXTREMELY POWERFUL ROUTINES WHICH SHOULD BE OF CONSIDERABLE VALUE TO ANYONE DESIRING TO MANIPULATE MATHEMATICAL DATA WITH AN 8008 SYSTEM. THE ROUTINES IN THE FORM PRESENTED FOR ILLUSTRATIVE PURPOSES ARE CAPABLE OF HANDLING BINARY NUMBERS THAT ARE THE DECIMAL EQUIVALENT OF 6 TO 7 SIGNIFICANT DIGITS RAISED TO APPROXIMATELY THE PLUS OR MINUS 38TH POWER OF TEN! THE ROUTINES CAN BE USED TO SOLVE A WIDE VARIETY OF MATHEMATICAL FORMULAS BY SIMPLY CALLING THE APPROPRIATE SUBROUTINES AFTER LOADING THE "FPOP" AND "FPACC" REGISTERS WITH THE VALUES THAT ARE TO BE MANIPULATED (WHEN THEY ARE IN NORMALIZED FLOATING-POINT FORMAT). FURTHERMORE, THE BASIC ROUTINES ILLUSTRATED CAN BECOME THE FUNDAMENTAL ROUTINES IN MORE SOPHISTICATED PROGRAMS THAT MIGHT BE DEVELOPED TO CALCULATE SUCH FUNCTIONS AS "SINES" AND "COSINES" USING NUMERICAL TECHNIQUES THAT CLOSELY APPROXIMATE THOSE FUNCTIONS BY TECHNIQUES SUCH AS "EXPANSION SERIES" FORMULAS.

THE INTERESTED PROGRAMMER SHOULD HAVE LITTLE DIFFICULTY IN MODIFYING THE ROUTINES ILLUSTRATED TO UPGRADE THEIR CAPABILITY TO PROVIDE MORE SIGNIFICANT DIGITS (BY INCREASING THE LENGTH OF THE MANTISSA) OR TO EXTEND THE "EXPONENTS" CAPABILITY BY PROVIDING DOUBLE OR TRIPLE-PRECISION REGISTERS FOR THE EXPONENT PORTION. FOR MANY APPLICATIONS, HOWEVER, THE USER MAY BE WELL SATISFIED WITH THE CAPABILITY PROVIDED BY THE ROUTINES AS THEY HAVE BEEN PRESENTED FOR EDUCATIONAL PURPOSES.

THE FLOATING-POINT ROUTINES WHICH HAVE BEEN PRESENTED CAN BE USED TO MANIPULATE NUMBERS ONCE THEY ARE IN BINARY FORMAT. IN SOME APPLICATIONS SUCH AS WHEN FORMULAS ARE BEING SOLVED BY THE COMPUTER TO CONTROL THE OPERATION OF A MACHINE, OR TYPES OF APPLICATIONS WHERE THERE IS LITTLE OR NO NEED TO COMMUNICATE WITH HUMANS, THE ABOVE ROUTINES COUPLED WITH SOME I/O ROUTINES AND WHATEVER OTHER OPERATING PROGRAMS ARE DICTATED BY THE APPLICATION WOULD BE SUFFICIENT FOR HANDLING THE MATHEMATICAL OPERATIONS. HOWEVER, IN PROBABLY THE MAJORITY OF APPLICATIONS, AT SOME TIME OR OTHER IT WILL BE DESIRABLE FOR HUMANS TO COMMUNICATE WITH THE COMPUTER AND FOR THE COMPUTER TO PRESENT INFORMATION BACK TO HUMANS. NOW, IT SEEMS THAT THE VAST MAJORITY OF PEOPLE PREFER TO MANIPULATE MATHEMATICAL

DATA USING DECIMAL NOTATION AND WOULD NOT WANT TO CHANGE THEIR WAYS BY WORKING IN FLOATING-POINT BINARY NOTATION. SO, MOST PROGRAMMERS WOULD FIND IT BENEFICIAL TO HAVE SOME CONVERSION ROUTINES THAT WOULD CONVERT NUMBERS FROM DECIMAL FLOATING-POINT NOTATION TO BINARY FLOATING-POINT NOTATION AS WELL AS THE REVERSE. THE NEXT SECTION OF THIS CHAPTER IS DEVOTED TO DISCUSSING AND DEVELOPING ROUTINES THAT ACCOMPLISH SUCH A WORTHWHILE OBJECTIVE!

CONVERTING FLOATING-POINT DECIMAL TO FLOATING-POINT BINARY

MOST USER'S OF A COMPUTER FOR MATHEMATICAL FUNCTIONS WOULD PROBABLY DESIRE TO INPUT DATA IN THE FORM:

1234.567

OR

1.234 E+15

USING AN INPUT DEVICE SUCH AS A KEYBOARD OR TELETYPE MACHINE. IN ORDER TO ACCEPT DATA IN SUCH FORMAT ONE NEEDS TO DEVELOP A PROGRAM THAT WILL FIRST CONVERT THE INFORMATION FROM THE DECIMAL MANTISSA AND EXPONENT FORM OVER TO THE BINARY EQUIVALENT. THE PROCESS IS FAIRLY STRAIGHT-FORWARD CONCEPTUALLY.

FIRST, ONE NEEDS TO DEVELOP A METHOD FOR BREAKING DOWN THE MANTISSA PORTION INTO A "DECIMAL NORMALIZED" FORMAT. THIS CAN BE DONE QUITE READILY BECAUSE:

1234.567 = 1234567.0 E-3

AND

1.234 E+15 = 1234.0 E+12

THUS, TO EFFECTIVELY "NORMALIZE" A DECIMAL NUMBER ONE HAS TO SIMPLY KEEP TRACK OF WHERE THE DECIMAL POINT IS PLACED BY THE OPERATOR IN THE MANTISSA AND COMPENSATE FOR THAT FACTOR BY REMOVING THE DECIMAL POINT (MAKING THE MANTISSA AN INTEGER VALUE) AND CHANGING THE EXPONENT PORTION TO ACCOUNT FOR THE REMOVAL OF THE DECIMAL POINT!

NEXT, ONE NEEDS TO CONVERT THE MANTISSA PORTION OF THE NUMBER FROM DECIMAL TO ITS BINARY EQUAL. THAT CONVERSION PROCESS CAN ACTUALLY BE ACCOMPLISHED AS EACH DECIMAL NUMBER IS INPUTTED BY THE OPERATOR BY USING THE ALGORITHM DESCRIBED BELOW.

> DECIMAL TO BINARY CONVERSION: EACH TIME A DIGIT IS RECEIVED IN DECIMAL FORM, IMMEDIATELY CONVERT IT TO IT'S BINARY EQUIVELANT. IN MANY CASES THIS CONSISTS OF SIMPLY "MASKING OFF" EXTRA BITS TO LEAVE A VALUE IN BCD FORMAT. NEXT, IN ORDER TO COMPENSATE FOR THE POWERS OF TEN DENOTED BY THE POSITIONAL WEIGHT OF DECIMAL NUMBERS, MULTIPLY ANY PREVIOUS NUMBER(S) THAT ARE ALREADY STORED IN BINARY FORM BY MULTIPLYING THEM BY 10 (DECIMAL). THEN ADD IN THE BINARY EQUIVALENT OF THE NUMBER THAT HAS JUST BEEN RECEIVED.

THE ALGORITHM CAN BE ILLUSTRATED BY CONSIDERING THE FOLLOWING EXAMPLE WHERE AN OPERATOR ENTERS THE DECIMAL NUMBER "63" BY FIRST ENTERING

THE NUMBER "6" AND THEN "3" FROM AN INPUT DEVICE SUCH AS AN ASCII CODED KEYBOARD:

```
0 0  0 0 0  0 0 0    INPUT REGISTER INITIALLY CLEARED
```

OPERATOR INITIALLY TYPES IN THE CHARACTER FOR A "6." THIS IS IMMEDIATELY CONVERTED TO 1 1 0 AS IT'S BINARY EQUIVALENT. SINCE IT IS THE FIRST CHARACTER RECEIVED IT IS NOT NECESSARY TO MULTIPLY THE PRESENT VALUE OF THE STORAGE REGISTER BY TEN. THE BINARY VALUE 1 1 0 CAN SIMPLY BE PLACED IN THE INPUT REGISTER GIVING:

```
0 0  0 0 0  1 1 0    INPUT REGISTER AFTER 1ST # RECEIVED
```

THE OPERATOR THEN ENTERS THE CHARACTER FOR A "3." ONCE AGAIN THIS IS IMMEDIATELY CONVERTED TO 0 1 1 AS IT'S BINARY EQUIVALENT. BUT, BEFORE THIS NEW DIGIT IS ADDED TO THE BINARY STORAGE REGISTER, THE CONTENTS OF THE REGISTER MUST BE MULTIPLIED BY TEN TO ACCOUNT FOR THE POSITIONAL VALUE OF THE PREVIOUS DIGIT. A SIMPLE WAY TO MULTIPLY A BINARY REGISTER BY TEN IS TO PERFORM THE FOLLOWING STEPS:

```
0 0  0 0 0  1 1 0    INPUT REGISTER CONTAINS 1ST # "6."

0 0  0 0 1  1 0 0    ROTATE LEFT = MULTIPLY BY 2

0 0  0 1 1  0 0 0    ROTATE LEFT = MULTIPLY BY 4

0 0  0 1 1  1 1 0    ADD IN ORIGINAL VALUE = MULT BY 5

0 0  1 1 1  1 0 0    ROTATE LEFT = MULTIPLY BY 10
```

WITH THE PREVIOUS VALUE OF "6" NOW MULTIPLIED BY TEN TO REPRESENT "60" IN THE BINARY REGISTER, THE NEW VALUE OF "3" CAN NOW BE ADDED IN TO YIELD:

```
0 0  1 1 1  1 1 1    BINARY EQUIVELANT OF "63" DECIMAL.
```

THE ABOVE ALGORITHM IS THUS REPEATED EACH TIME AN ADDITIONAL DECIMAL CHARACTER IS RECEIVED TO MAINTAIN THE BINARY EQUIVALENT. NATURALLY THE ALGORITHM IS VALID FOR MULTIPLE-PRECISION STORAGE OF NUMBERS.

FINALLY, IT IS NECESSARY TO CONVERT THE DECIMAL EXPONENT VALUE (WHICH AGAIN IS IMMEDIATELY CONVERTED TO A BINARY NUMBER AS IT IS RECEIVED FROM THE INPUT DEVICE) TO REPRESENT THE BINARY NUMBER RAISED TO AN EQUIVALENT VALUE. CONVERSION AT THIS POINT MAY BE ACCOMPLISHED BY FIRST CONVERTING THE BINARY REPRESENTATION OF THE MANTISSA TO IT'S "NORMALIZED" FORMAT (USING THE SPECIAL CAPABILITY OF THE "FPNORM" ROUTINE TO CONVERT THE REGULAR FORMATTED BINARY NUMBER TO IT'S NORMALIZED FORM) AND THEN MULTIPLYING THE NORMALIZED FLOATING-POINT BINARY NUMBER BY 10 (DECIMAL) FOR EACH UNIT OF A POSITIVE DECIMAL EXPONENT OR MULTIPLYING IT BY 0.1 FOR EACH UNIT OF A MINUS DECIMAL EXPONENT. THIS CAN BE ACCOMPLISHED BY USING THE "FPMULT" ROUTINE PREVIOUSLY DESCRIBED!

THE DECIMAL TO BINARY INPUT PROGRAM TO BE PRESENTED SHORTLY HANDLES THE ABOVE CONSIDERATIONS PLUS ALLOWS SEVERAL OTHER FUNCTIONS TO BE PERFORMED. THE ROUTINE WILL ALLOW AN OPERATOR TO SPECIFY THE SIGN OF THE DECIMAL MANTISSA AND EXPONENT AND TAKES APPROPRIATE ACTION TO NEGATE

NUMBERS DESIGNATED AS BEING MINUS IN VALUE. IT ALSO ALLOWS FOR ERASURE OF THE CURRENT INPUT STRING BY TYPING A SPECIAL CHARACTER. THE ROUTINE ASSUMES THAT CHARACTERS ARE RECEIVED FROM AN INPUT DEVICE THAT USES ASCII CODE AND THAT AN OUTPUT DEVICE USING ASCII CODE IS USED TO "ECHO" INFORMATION RECEIVED BACK TO THE OPERATOR. NEITHER THE ACTUAL INPUT OR OUTPUT ROUTINES ARE SHOWN IN THE SAMPLE PROGRAM. (INFORMATION ON ACTUAL I/O ROUTINES WILL BE PRESENTED IN A LATER CHAPTER). THE ROUTINE ALSO ASSUMES THAT CERTAIN LOCATIONS ON PAGE 00 WILL BE USED FOR STORAGE OF NUMBERS RECEIVED AND FOR MAINTAINING COUNTERS AND INDICATORS. A LISTING OF THE LOCATIONS USED WILL BE PROVIDED LATER. ADDITIONALLY, THE PROGRAM CALLS ON OTHER ROUTINES PREVIOUSLY DETAILED IN THIS MANUAL SUCH AS "FPNORM" AND "FPMULT."

```
          MNEMONIC                    COMMENTS
     ----------------          ----------------

DINPUT,  LHI 000        /SET POINTERS TO INPUT
         LLI 150        /STORAGE REGISTERS
         XRA            /CLEAR ACCUMULATOR
         LBI 010        /SET A COUNTER
CLRNX2,  LMA            /AND CLEAR MEMORY LOCATIONS 150 - 157
         INL            /BY DEPOSITING 0'S AND ADVANCING PNTR
         DCB            /AND DECREMENTING LOOP COUNTER
         JFZ CLRNX2     /UNTIL FINISHED
         LLI 103        /SET POINTERS TO CNTR/INDICATOR STORAGE
         LBI 004        /SET A COUNTER
CLRNX3,  LMA            /AND CLEAR MEMORY LOCATIONS 103 - 106
         INL            /IN A SIMILAR FASHION BY DEPOSITING 0'S
         DCB            /AND DECREMENTING LOOP COUNTER
         JFZ CLRNX3     /UNTIL FINISHED
         CAL INPUT      /NOW BRING IN A CHARACTER FROM I/O DEVICE
         CPI 253        /TEST TO SEE IF IT IS A "+" SIGN
         JTZ SECHO      /IF YES, GO TO ECHO AND CONTINUE
         CPI 255        /IF NOT "+" SEE IF "-" SIGN
         JFZ NOTPLM     /IF NOT "+" OR "-" TEST FOR VALID CHAR
         LLI 103        /IF MINUS, SET POINTER TO "INPUT SIGN"
         LMA            /AND MAKE IT NON-ZERO BY DEPOSITING CHAR
SECHO,   CAL ECHO       /OUTPUT CHAR IN ACC AS ECHO TO OPERATOR
NINPUT,  CAL INPUT      /FETCH A NEW CHARACTER FROM I/O DEVICE
NOTPLM,  CPI 377        /SEE IF CHARACTER IS CODE FOR "RUBOUT"
         JTZ ERASE      /IF YES, PREPARE TO START OVER
         CPI 256        /IF NOT, SEE IF CHARACTER IS A PERIOD "."
         JTZ PERIOD     /IF "." PROCESS AS DECIMAL POINT
         CPI 305        /IF NOT, SEE IF CHAR IS "E" FOR EXPONENT
         JTZ FNDEXP     /IF "E" PROCESS AS EXPONENT INDICATOR
         CPI 260        /IF NOT, SEE IF CHAR A VALID NUMBER
         JTS ENDINP     /IF NONE OF ABOVE, TERMINATE INPUT STRING
         CPI 272        /STILL CHECKING FOR VALID NUMBER
         JFS ENDINP     /IF NOT, TERMINATE INPUT STRING
         LLI 156        /HAVE A #, SET PNTR TO MSW OF INPUT REG'S
         LBA            /SAVE CHARACTER IN REGISTER "B"
         LAI 370        /FORM A MASK AND CHECK TO SEE IF INPUT
         NDM            /REG'S CAN ACCEPT LARGER NUMBER
         JFZ NINPUT     /IF NOT, IGNORE PRESENT INPUT
         LAB            /IF O.K., RESTORE CHARACTER TO ACC
         CAL ECHO       /AND ECHO # BACK TO OPERATOR
         LLI 105        /SET POINTER TO DIGIT COUNTER
         LCM            /FETCH DIGIT COUNTER
         INC            /INCREMENT IT'S VALUE
         LMC            /AND RESTORE IT TO STORAGE
```

```
          MNEMONIC                COMMENTS
        -------------           -------------

          CAL DECBIN    /PERFORM DECIMAL TO BINARY CONVERSION
          JMP NINPUT    /GET NEXT CHARACTER FOR MANTISSA
PERIOD,   LBA           /SUBRTN TO PROCESS "." - SAVE IN "B"
          LLI 106       /SET POINTER TO "." STORAGE INDICATOR
          LAM           /FETCH CONTENTS
          NDA           /SET FLAGS AFTER LOAD OPERATION
          JFZ ENDINP    /IF "." ALREADY PRESENT, END INPUT STRING
          LLI 105       /OTHERWISE SET PNTR TO DIGIT COUNTER
          LMA           /AND RESET DIGIT COUNTER TO 0
          INL           /ADVANCE POINTER BACK TO "." STORAGE
          LMB           /AND PUT A "." THERE
          LAB           /RESTORE "." TO ACCUMULATOR
          CAL ECHO      /AND ECHO IT BACK TO OPERATOR
          JMP NINPUT    /GET NEXT CHARACTER IN NUMBER STRING
ERASE,    LAI 274       /PUT ASCII CODE FOR "<" IN ACCUMULATOR
          CAL ECHO      /DISPLAY IT
          LAI 240       /PUT ASCII CODE FOR "SPACE" IN ACC
          CAL ECHO      /AND LEAVE A COUPLE OF SPACES
          CAL ECHO      /BEFORE GOING BACK TO
          JMP DINPUT    /START THE INPUT STRING OVER
FNDEXP,   CAL ECHO      /SUBRTN TO PROCESS EXPONENT, ECHO "E"
          CAL INPUT     /GET NEXT PART OF EXPONENT
          CPI 253       /TEST FOR A "+" SIGN
          JTZ EXECHO    /IF YES, PROCEED TO ECHO IT
          CPI 255       /IF NOT, TEST FOR A "-" SIGN
          JFZ NOEXPS    /IF NOT, SEE IF A VALID CHARACTER
          LLI 104       /IF HAVE "-" THEN SET PNTR TO "EXP SIGN"
          LMA           /SET "EXP SIGN" MINUS INDICATOR
EXECHO,   CAL ECHO      /ECHO CHARACTER BACK TO OPERATOR
EXPINP,   CAL INPUT     /GET NEXT CHARACTER FOR EXPONENT PORTION
NOEXPS,   CPI 377       /SEE IF CODE FOR "RUBOUT"
          JTZ ERASE     /IF YES, PREPARE TO RE-ENTER ENTIRE STRING
          CPI 260       /OTHERWISE CHECK FOR VALID DECIMAL NUMBER
          JTS ENDINP    /IF NOT, END INPUT STRING
          CPI 272       /STILL TESTING FOR VALID NUMBER
          JFS ENDINP    /IF NOT, END INPUT STRING
          NDI 017       /HAVE VALID #, FORM MASK AND STRIP ASCII
          LBA           /CHARACTER TO PURE BCD, SAVE IN REG "B"
          LLI 157       /SET PNTR TO INPUT EXPONENT STORAGE LOC
          LAI 003       /SET ACCUMULATOR = 3
          CPM           /SEE IF 1ST EXPONENT # WAS GREATER THAN 3
          JTS EXPINP    /IF YES, IGNORE INPUT (LIMITS EXP TO < 40)
          LCM           /IF O.K., SAVE PREV EXP VALUE IN "C"
          LAM           /AND ALSO PLACE IT IN ACCUMULATOR
          NDA           /CLEAR THE CARRY BIT
          RAL           /MULT X 10 ALGORITHM, 1ST MULT X 2
          RAL           /MULT X 2 AGAIN
          ADC           /ADD IN ORIGINAL VALUE
          RAL           /MULT X 2 ONCE MORE
          ADB           /ADD IN NEW # TO COMPLETE THE DECIMAL TO
          LMA           /BINARY CONV FOR EXP AND RESTORE TO MEMORY
          LAI 260       /RESTORE ASCII CODE BY ADDING "260"
          ADB           /TO BCD VALUE OF THE NUMBER
          JMP EXECHO    /AND ECHO # THEN LOOK FOR NEXT INPUT
ENDINP,   LLI 103       /SET POINTER TO MANTISSA "SIGN" INDICATOR
          LAM           /FETCH SIGN INDICATOR
          NDA           /SET FLAGS AFTER LOAD OPERATION
          JTZ FININP    /IF NOTHING IN INDICATOR, # IS POSITIVE
```

```
        MNEMONIC              COMMENTS
     -------------         -------------

        LLI 154       /SET PNTR TO LSW OF INPUT MANTISSA
        LBI 003       /SET PRECISION
        CAL COMPLM    /PERFORM 2'S COMPLEMENT TO NEGATE NUMBER
FININP, LLI 153       /SET PNTR TO INPUT STORAGE LSW-1
        XRA           /CLEAR ACCUMULATOR
        LDA           /CLEAR REG "D"
        LMA           /CLEAR INPUT STORAGE LOC LSW-1
        LEI 123       /SET PNTR TO FPACC LSW-1
        LBI 004       /SET PRECISION COUNTER
        CAL MOVEIT    /MOVE INPUT & LSW-1 TO FPACC & LSW-1
        LBI 027       /SET SPEC FPNORM MODE BY SETTING BIT CNT
        CAL FPNORM    /IN REG "B" AND CALL NORMALIZATION ROUTINE
        LLI 104       /SET POINTER TO EXPONENT SIGN INDICATOR
        LAM           /FETCH EXPONENT SIGN INDICATOR TO ACC
        NDA           /SET FLAGS AFTER LOAD OPERATION
        LLI 157       /SET POINTER TO DECIMAL EXP STORAGE
        JTZ POSEXP    /IF EXP POSITIVE, JUMP AHEAD
        LAM           /IF EXP NEGATIVE, FETCH IT INTO ACC
        XRI 377       /AND PERFORM TWO'S
        ADI 001       /COMPLEMENT
        LMA           /THEN RESTORE TO STORAGE LOCATION
POSEXP, LLI 106       /SET POINTER TO PERIOD INDICATOR
        LAM           /FETCH CONTENTS TO ACCUMULATOR
        NDA           /SET FLAGS AFTER LOAD OPERATION
        JTZ EXPOK     /IF NOTHING, NO DECIMAL POINT INVOLVED
        LLI 105       /IF HAVE DECIMAL POINT, SET PTR TO DIGIT
        XRA           /COUNTER THEN CLEAR ACCUMULATOR
        SUM           /SUBTRACT DIGIT CNTR FROM 0 TO GIVE NEG
EXPOK,  LLI 157       /SET POINTER TO DECIMAL EXPONENT STORAGE
        ADM           /ADD IN COMPENSATION FOR DECIMAL POINT
        LMA           /RESTORE COMPENSATED VALUE TO STORAGE
        JTS MINEXP    /IF COMPENSATED VALUE MINUS, JUMP AHEAD
        RTZ           /IF COMPENSATED VALUE ZERO: FINISHED!
EXPFIX, CAL FPX10     /COMPEN DEC EXP IS +, MULT FPACC X 10
        JFZ EXPFIX    /LOOP TIL DECIMAL EXPONENT = 0
        RET           /EXIT WITH CONVERTED VALUE IN FPACC
FPX10,  LEI 134       /MULT FPACC X 10 RTN, SET PNTR TO FPOP LSW
        LDH           /SET D = 0 FOR SURE
        LLI 124       /SET PNTR TO FPACC LSW
        LBI 004       /SET PRECISION COUNTER
        CAL MOVEIT    /MOVE FPACC TO FPOP (INCLUDING EXPONENTS)
        LLI 127       /SET PNTR TO FPACC EXPONENT
        LMI 004       /PLACE FP FORM OF 10 (DECIMAL) IN FPACC
        DCL           /  "   "   "   "  "     "       "   "
        LMI 120       /  "   "   "   "  "     "       "   "
        DCL           /  "   "   "   "  "     "       "   "
        XRA           /  "   "   "   "  "     "       "   "
        LMA           /  "   "   "   "  "     "       "   "
        DCL           /  "   "   "   "  "     "       "   "
        LMA           /  "   "   "   "  "     "       "   "
        CAL FPMULT    /NOW MULTIPLY ORIG BIN # (IN FPOP) X 10
        LLI 157       /SET POINTER TO DECIMAL EXPONENT STORAGE
        CAL CNTDWN    /DECREMENT DEC EXP VALUE
        RET           /RETURN TO CALLING PROGRAM
MINEXP, CAL FPD10     /COMPEN DEC EXP IS -, MULT FPACC X 0.1
        JFZ MINEXP    /LOOP TIL DECIMAL EXPONENT = 0
        RET           /EXIT WITH CONVERTED VALUE IN FPACC
FPD10,  LEI 134       /MULT FPACC X 0.1 RTN, PNTR TO FPOP LSW
```

```
          MNEMONIC                    COMMENTS
          --------                    --------
               LDM          /SET D = 0 FOR SURE
               LLI 124      /SET POINTER TO FPACC
               LBI 004      /SET PRECISION COUNTER
               CAL MOVEIT   /MOVE FPACC TO FPOP (INCLUDING EXPONENT)
               LLI 127      /SET POINTER TO FPACC EXPONENT
               LMI 375      /PLACE FP FORM OF 0.1 (DECIMAL) IN FPACC
               DCL          /  "   "   "   "   "   "   "   "   "
               LMI 146      /  "   "   "   "   "   "   "   "   "
               DCL          /  "   "   "   "   "   "   "   "   "
               LMI 146      /  "   "   "   "   "   "   "   "   "
               DCL          /  "   "   "   "   "   "   "   "   "
               LMI 147      /  "   "   "   "   "   "   "   "   "
               CAL FPMULT   /NOW MULTIPLY ORIG BIN # (IN FPOP) X 0.1
               LLI 157      /SET POINTER TO DECIMAL EXPONENT STORAGE
               LBM          /FETCH VALUE
               INB          /INCREMENT IT
               LMB          /RESTORE IT TO MEMORY
               RET          /RETURN TO CALLING PROGRAM
DECBIN.        LLI 153      /DEC TO BIN CONV. SET PNTR TO TEMP STORAGE
               LAB          /RESTORE CHARACTER TO ACCUMULATOR
               NDI 017      /MASK OFF ASCII BITS TO LEAVE PURE BCD #
               LMA          /PLACE CURRENT BCD # IN TEMP STORAGE
               LEI 150      /SET POINTER TO WORKING AREA LSW
               LLI 154      /SET ANOTHER PNTR TO LSB OF INPUT REG'S
               LDM          /SET D = 0 FOR SURE
               LBI 003      /SET PRECISION COUNTER
               CAL MOVEIT   /MOVE ORIGINAL VALUE TO WORKING AREA
               LLI 154      /SET PNTR TO LSW OF INPUT STORAGE
               LBI 003      /SET PRECISION COUNTER
               CAL ROTATL   /ROTATE LEFT (X 2)          (TOTAL = X 2)
               LLI 154      /SET PNTR TO LSW AGAIN
               LBI 003      /SET PRECISION COUNTER
               CAL ROTATL   /ROTATE LEFT (X 2)     (TOTAL NOW = X 4)
               LEI 154      /SET PNTR TO LSW OF ROTATED VALUE
               LLI 150      /AND ANOTHER TO LSW OF ORIGINAL VALUE
               LBI 003      /SET PRECISION COUNTER
               CAL ADDER    /ADD ORIG TO ROTATED  (TOTAL NOW = X 5)
               LLI 154      /SET PNTR TO LSW AGAIN
               LBI 003      /SET PRECISION COUNTER
               CAL ROTATL   /ROTATE LEFT (X 2)     (TOTAL NOW = X 10)
               LLI 152      /SET POINTER TO CLEAR WORKING AREA
               XRA          /CLEAR ACCUMULATOR
               LMA          /DEPOSIT IN MSW OF WORKING AREA
               DCL          /DECREMENT PNTR TO MSW
               LMA          /PUT ZERO THERE TOO
               LLI 153      /SET PNTR TO CURRENT DIGIT STORAGE
               LAM          /FETCH LATEST BCD NUMBER
               LLI 150      /SET PNTR TO LSW OF WORKING AREA
               LMA          /DEPOSIT LATEST BCD NUMBER IN LSW
               LEI 154      /SET UP POINTER
               LBI 003      /SET PRECISION COUNTER
               CAL ADDER    /ADD IN LATEST # TO COMPLETE DECBIN CONV
               RET          /RETURN TO CALLING PROGRAM
```

CONVERTING FLOATING-POINT BINARY TO FLOATING-POINT DECIMAL

THE FOLLOWING PROGRAM WILL CONVERT BINARY NUMBERS STORED IN FLOATING-POINT FORMAT TO DECIMAL FLOATING-POINT FORMAT AND DISPLAY THEM ON AN

OUTPUT DEVICE SUCH AS A TELETYPE MACHINE IN THE FOLLOWING FORMAT:

+0.1234567 E+07

THE ROUTINE WHICH IS SHOWN BELOW OPERATES ESSENTIALLY IN THE REVERSE MANNER TO THE INPUT ROUTINE. FIRST THE FLOATING-POINT BINARY NUMBER IS CONVERTED TO A REGULARLY FORMATTED BINARY NUMBER, AND THEN THE NUMBER IS CONVERTED TO A DECIMAL NUMBER USING A MULTIPLY BY TEN ALGORITHM. SINCE THE READER SHOULD NOW BE QUITE ADEPT AT FOLLOWING THE OPERATION OF A PROGRAM FROM THE COMMENTED SOURCE LISTING, THE FLOATING-POINT BINARY TO FLOATING-POINT DECIMAL CONVERSION ROUTINE WILL BE PRESENTED WITHOUT FURTHER DISCUSSION AT THIS POINT. IT SHOULD BE REMEMBERED THAT THE ROUTINE ILLUSTRATED ASSUMES AN ASCII CODED OUTPUT DEVICE IS BEING UTILIZED. IN ADDITION, SEVERAL SUBROUTINES USED BY THE PREVIOUSLY ILLUSTRATED INPUT PROGRAM ARE CALLED BY THE ROUTINE.

	MNEMONIC		COMMENTS
FPOUT,	LLI	157	/SET POINTER TO DECIMAL EXPONENT STORAGE
	LMI	000	/CLEAR DECIMAL EXPONENT STORAGE LOCATION
	LLI	126	/SET POINTER TO MSW FPACC MANTISSA
	LAM		/FETCH MSW FPACC MANTISSA TO ACCUMULATOR
	NDA		/SET FLAGS AFTER LOAD OPERATION
	JTS	OUTNEG	/IF MSB = 1 HAVE NEGATIVE NUMBER
	LAI	253	/OTHERWISE # IS POS, SET ASCII CODE FOR +
	JMP	AHEAD1	/GO TO DISPLAY "+" SIGN
OUTNEG,	LLI	124	/HAVE NEG #, SET PNTR TO LSW FPACC MANT
	LBI	003	/SET PRECISION COUNTER
	CAL	COMPLM	/PERFORM TWO'S COMPLEMENT ON FPACC
	LAI	255	/SET ASCII CODE FOR "-" SIGN
AHEAD1,	CAL	ECHO	/DISPLAY SIGN OF MANTISSA
	LAI	260	/SET ASCII CODE FOR "0"
	CAL	ECHO	/DISPLAY "0"
	LAI	256	/SET ASCII CODE FOR "."
	CAL	ECHO	/DISPLAY "."
	LLI	127	/SET POINTER TO FPACC EXPONENT
	LAI	377	/PUT -1 IN ACCUMULATOR
	ADM		/EFFECTIVELY SUBTRACT "1" FROM EXPONENT
	LMA		/RESTORE COMPENSATED EXPONENT
DECEXT,	JFS	DECEXD	/IF COMPEN EXP 0 OR POS, MULT MANT X 0.1
	LAI	004	/IF COMPEN EXP NEGATIVE
	ADM		/ADD "4" (DECIMAL) TO THAT VALUE
	JFS	DECOUT	/IF EXPONENT 0 OR POS NOW, OUTPUT MANTISSA
	CAL	FPX10	/OTHERWISE, MULT MANTISSA BY 10
DECREP,	LLI	127	/SET POINTER TO FPACC EXPONENT
	LAM		/GET EXPONENT AFTER MULTIPLICATION RTN
	NDA		/SET FLAGS AFTER LOAD OPERATION
	JMP	DECEXT	/REPEAT ABOVE TEST FOR 0 OR POS CONDITION
DECEXD,	CAL	FPD10	/MULTIPLY FPACC X 0.1
	JMP	DECREP	/CHECK STATUS OF FPACC EXP AFTER MULTIP
DECOUT,	LEI	164	/SET POINTER TO LSW OF OUTPUT REGISTERS
	LDH		/MAKE D = 0 FOR SURE
	LLI	124	/SET POINTERS TO LSW OF FPACC
	LBI	003	/SET PRECISION COUNTER
	CAL	MOVEIT	/MOVE FPACC TO OUTPUT REGISTERS
	LLI	167	/SET PNTR TO MSW+1 OF OUTPUT REGISTER
	LMI	000	/AND CLEAR THAT LOCATION
	LLI	164	/NOW SET POINTER TO LSW OF OUTPUT REG'S
	LBI	003	/SET PRECISION COUNTER - PERFORM ONE

```
                MNEMONIC                COMMENTS
                --------                --------

        CAL ROTATL      /ROTATE OP TO COMPEN FOR SPACE OF SIGN BIT
        CAL OUTX10      /MULT OUTPUT REG X 10, OVERFLOW INTO MSW+1
COMPEN, LLI 127         /SET PNTR TO FPACC EXPONENT
        LBM             /COMPENSATE FOR ANY REMAINDER IN BINARY
        INB             /EXPONENT BY PERFORMING A ROTATE RIGHT ON
        LMB             /OUTPUT REG'S UNTIL BIN EXP BECOMES ZERO
        JTZ OUTDIG      /GO TO OUTPUT DIGITS WHEN COMPEN DONE
        LLI 167         /BIN EXP COMPENSATION ROTATE LOOP
        LBI 004         /SET PNTR TO OUT MSW+1 AND SET COUNTER
        CAL ROTATR      /PERFORM COMPENSATING ROTATE RIGHT OP
        JMP COMPEN      /REPEAT LOOP UNTIL BIN EXP = 0
OUTDIG, LLI 107         /SET PNTR TO OUTPUT DIGIT COUNTER
        LMI 007         /SET DIGIT COUNTER TO "7" TO INITIALIZE
        LLI 167         /SET PNTR TO MSD IN OUT REG MSW+1
        LAM             /FETCH BCD FORM OF DIGIT TO BE DISPLAYED
        NDA             /SET FLAGS AFTER LOAD OPERATION
        JTZ ZERODG      /SEE IF 1ST DIGIT WOULD BE A "0"
OUTDGS, LLI 167         /IF NOT, SET PNTR TO MSW+1 (BCD CODE)
        LAI 260         /FORM ASCII NUMBER CODE BY ADDING 260
        ADM             /TO BCD CODE
        CAL ECHO        /AND DISPLAY THE DECIMAL NUMBER
DECRDG, LLI 107         /SET POINTER TO OUTPUT DIGIT COUNTER
        CAL CNTDWN      /DECREMENT VALUE OF OUTPUT DIGIT CNTR
        JTZ EXPOUT      /WHEN = 0, GO DO EXPONENT OUTPUT RTN
        CAL OUTX10      /OTHERWISE MULT OUTPUT REG'S X 10
        JMP OUTDGS      /AND OUTPUT NEXT DECIMAL DIGIT
ZERODG, LLI 157         /IF 1ST DIGIT = 0, SET PNTR TO DEC EXP
        CAL CNTDWN      /DECR VALUE TO COMPEN FOR SKIPPING DISPLAY
        LLI 166         /OF 1ST DIGIT, THEN SET POINTER TO MSW
        LAM             /OF OUTPUT REG'S - FETCH CONTENTS
        NDA             /SET FLAGS AFTER LOAD OPERATIONS
        JFZ DECRDG      /CHECK TO SEE IF ENTIRE MANTISSA IS "0"
        DCL             /  "   "   "   "   "   "   "   "   "
        LAM             /  "   "   "   "   "   "   "   "   "
        NDA             /  "   "   "   "   "   "   "   "   "
        JFZ DECRDG      /  "   "   "   "   "   "   "   "   "
        DCL             /  "   "   "   "   "   "   "   "   "
        LAM             /  "   "   "   "   "   "   "   "   "
        NDA             /  "   "   "   "   "   "   "   "   "
        JFZ DECRDG      /  "   "   "   "   "   "   "   "   "
        LLI 157         /IF ENTIRE MANTISSA IS ZERO, SET PNTR TO
        LMA             /DECIMAL EXPONENT STORAGE AND SET IT TO 0
        JMP DECRDG      /BEFORE PROCEEDING TO FINISH DISPLAY
OUTX10, LLI 167         /MULTIPLY OUTPUT REG'S BY 10 TO PUSH OUT
        LMI 000         /BCD CODE OF MSD, 1ST CLEAR OUTPUT MSW+1
        LLI 164         /SET PNTR TO LSW OF OUTPUT REGISTERS
        LDH             /MAKE SURE D = 0
        LEI 160         /SET ANOTHER PNTR TO WORKING AREA
        LBI 004         /SET PRECISION COUNTER
        CAL MOVEIT      /MOVE ORIGINAL VALUE TO WORKING AREA
        LLI 164         /SET POINTER TO ORIGINAL VALUE LSW
        LBI 004         /SET PRECISION COUNTER
        CAL ROTATL      /START MULT X 10 ROUTINE    (TOTAL = X 2)
        LLI 164         /RESET PNTR
        LBI 004         /AND COUNTER
        CAL ROTATL      /MULT X 2 AGAIN             (TOTAL = X 4)
        LLI 160         /SET POINTER TO LSW OF ORIG VALUE
        LEI 164         /AND ANOTHER TO LSW OF ROTATED VALUE
```

```
                MNEMONIC                    COMMENTS
            --------------              --------------
                LBI 004         /SET PRECISION COUNTER
                CAL ADDER       /ADD ORIG VALUE TO ROTATED  (TOTAL = X 5)
                LLI 164         /RESET PNTR
                LBI 004         /AND COUNTER
                CAL ROTATL      /MULT X 2 ONCE MORE         (TOTAL = X 10)
                RET             /FINISHED MULT OUTPUT REG'S X 10
EXPOUT,         LAI 305         /SET ASCII CODE FOR "E"
                CAL ECHO        /DISPLAY "E" FOR "EXPONENT"
                LLI 157         /SET POINTER TO DECIMAL EXP STORAGE LOC
                LAM             /FETCH DECIMAL EXPONENT TO ACC
                NDA             /SET FLAGS AFTER LOAD OPERATION
                JTS EXOUTN      /IF MSB = 1, VALUE IS NEGATIVE
                LAI 253         /IF VALUE IS POS, SET ASCII CODE FOR "+"
                JMP AHEAD2      /GO TO DISPLAY SIGN
EXOUTN,         XRI 377         /FOR NEG EXP, PERFORM TWO'S COMPLEMENT
                ADI 001         /IN STANDARD MANNER
                LMA             /AND RESTORE TO STORAGE LOCATION
                LAI 255         /SET ASCII CODE FOR "-"
AHEAD2,         CAL ECHO        /DISPLAY SIGN OF EXPONENT
                LBI 000         /CLEAR REGISTER "B" FOR COUNTER
                LAM             /FETCH DECIMAL EXPONENT VALUE
SUB12,          SUI 12          /SUBTRACT 10 (DECIMAL)
                JTS TOMUCH      /LOOK FOR NEGATIVE RESULT
                LMA             /RESTORE POS RESULT, MAINTAIN COUNT OF HOW
                INB             /MANY TIMES 10 (DECIMAL) CAN BE SUBTRACTED
                JMP SUB12       /TO OBTAIN MOST SIG DIGIT OF EXPONENT
TOMUCH,         LAI 260         /FORM ASCII CHAR FOR MSD OF EXPONENT BY
                ADB             /ADDING 260 TO COUNT IN REGISTER "B"
                CAL ECHO        /AND DISPLAY MOST SIGNIFICANT DIGIT OF EXP
                LAM             /FETCH REMAINDER IN DEC EXP STORAGE LOC
                ADI 260         /AND FORM ASCII CHAR FOR LSD OF EXPONENT
                CAL ECHO        /DISPLAY LEAST SIGNIFICANT DIGIT OF EXP
                RET             /EXIT "FPOUT" ROUTINE
```

ONCE ONE HAS A DECIMAL TO BINARY INPUT ROUTINE, AND BINARY TO DECIMAL OUTPUT ROUTINE TO WORK WITH THE FUNDAMENTAL FLOATING-POINT ROUTINES IT IS A RELATIVELY SIMPLE MATTER TO TIE THEM ALL TOGETHER TO FORM AN "OPERATING PACKAGE" THAT WOULD ALLOW AN OPERATOR TO SPECIFY NUMERICAL VALUES IN FLOATING-POINT DECIMAL NOTATION AND INDICATE WHETHER ADDITION, SUBTRACTION, MULTIPLICATION OR DIVISION WAS DESIRED, THEN OBTAIN AN ANSWER FROM THE COMPUTER. AN ILLUSTRATIVE "OPERATING PROGRAM" THAT UTILIZES ALL THE DEMONSTRATION ROUTINES PRESENTED IN THIS SECTION IS SHOWN BELOW. THE PROGRAM WILL ALLOW AN OPERATOR TO MAKE ENTRIES AND RECEIVE RESULTS IN THE FORMAT SHOWN HERE:

+33.0E+3 X -4 = -0.1320000E+6

```
                MNEMONIC                    COMMENTS
            --------------              --------------

FPCONT,         CAL CRLF2       /DISPLAY A FEW CR & LF'S FOR I/O DEVICE
                CAL INPUT       /LET OPERATOR ENTER A FP DECIMAL NUMBER
                CAL SPACES      /DISPLAY A FEW SPACES AFTER NUMBER
                LLI 124         /SET PNTR TO LSW OF FPACC
                LDH             /SET D = 0 FOR SURE
                LEI 170         /SET PNTR TO TEMP # STORAGE AREA
                LBI 004         /SET PRECISION COUNTER
```

```
                MNEMONIC                    COMMENTS
                --------                    --------

          CAL MOVEIT    /MOVE FPACC TO TEMP STORAGE AREA
NVALID,   CAL INPUT     /FETCH "OPERATOR" FROM INPUT DEVICE
          LBI 000       /CLEAR REGISTER "B"
          CPI 253       /TEST FOR "+" SIGN
          JTZ OPERA1    /GO SET UP FOR "+" SIGN
          CPI 255       /IF NOT "+," TEST FOR "-" SIGN
          JTZ OPERA2    /GO SET UP FOR "-" SIGN
          CPI 330       /IF NOT ABOVE, TEST FOR "X" (MULT) SIGN
          JTZ OPERA3    /GO SET UP FOR "X" SIGN
          CPI 257       /IF NOT ABOVE, TEST FOR "/" (DIV) SIGN
          JTZ OPERA4    /GO SET UPF FOR "/" SIGN
          CPI 377       /IF NOT ABOVE, TEST FOR "RUBOUT"
          JFZ NVALID    /IF NONE OF ABOVE, IGNORE INPUT
          JMP FPCONT    /IF "RUBOUT" START NEW INPUT SEQUENCE
OPERA1,   DCB           /SET UP REGISTER "B" BASED ON ABOVE
          DCB           / "  "    "     "     "    "    "
OPERA2,   DCB           / "  "    "     "     "    "    "
          DCB           / "  "    "     "     "    "    "
OPERA3,   DCB           / "  "    "     "     "    "    "
          DCB           / "  "    "     "     "    "    "
OPERA4,   LCA           /SAVE "OPERATOR" CHARACTER IN REG "C"
          LAI ***       /*** = NEXT TO LAST LOC IN "LOOKUP" TABLE
          ADB           /MODIFY "***" BY CONTENTS OF "B"
          LLI 110       /SET PNTR TO "LOOKUP" TABLE ADDR STORAGE
          LMA           /PLACE "LOOKUP" ADDR IN STORAGE LOCATION
          LAC           /RESTORE "OPERATOR" CHARACTER TO ACC
          CAL ECHO      /DISPLAY THE "OPERATOR" SIGN
          CAL SPACES    /DISPLAY FEW SPACES AFTER "OPERATOR" SIGN
          CAL DINPUT    /LET OPERATOR ENTER 2ND FP DECIMAL NUMBER
          CAL SPACES    /PROVIDE FEW SPACES AFTER 2ND NUMBER
          LAI 275       /PLACE ASCII CODE FOR "=" IN ACCUMULATOR
          CAL ECHO      /DISPLAY "=" SIGN
          CAL SPACES    /DISPLAY FEW SPACES AFTER "=" SIGN
          LLI 170       /SET POINTER TO TEMP NUMBER STORAGE AREA
          LDH           /SET D = 0 FOR SURE
          LEI 134       /SET ANOTHER POINTER TO LSW FPOP
          LBI 004       /SET PRECISION COUNTER
          CAL MOVEIT    /MOVE 1ST NUMBER INPUTTED TO FPOP
          LLI 110       /SET PNTR TO "LOOKUP" TABLE ADDR STORAGE
          LLM           /BRING IN LOW ORDER ADDR OF "LOOKUP" TABLE
          LHI XXX       /XXX = PAGE THIS PROGRAM LOCATED ON
          LEM           /BRING IN AN ADDR STORED IN "LOOKUP" TABLE
          INL           /RESIDING ON THIS PAGE (XXX) AT LOCATIONS
          LDM           /"*** + B" AND "*** + B + 1" AND PLACE IT
          LLI Z+1       /IN REGS "D & E" THEN CHANGE PNTR TO ADDR
          LME           /PART OF INSTRUCTION LABELED "RESULT" BE-
          INL           /LOW AND TRANSFER THE "LOOKUP" TABLE CON-
          LMD           /TENTS TO BECOME THE ADDRESS FOR THE IN-
          LHI 000       /STRUCTION LABELED "RESULT."  THEN RESTORE
          LDH           /REGISTERS "D" AND "H" BACK TO "0"
          JMP RESULT    /NOW JUMP TO COMMAND LABELED "RESULT"
CRLF2,    LAI 215       /SUBRTN TO PROVIDE CR & LF'S
          CAL ECHO      /PLACE ASCII CODE FOR CR IN ACC & DISPLAY
          LAI 212       /PLACE ASCII CODE FOR LINE FEED IN ACC
          CAL ECHO      /AND DISPLAY
          LAI 215       /DO IT AGAIN - CODE FOR CR IN ACC
          CAL ECHO      /DISPLAY IT
          LAI 212       /CODE FOR LF
```

```
                    MNEMONIC                    COMMENTS
                    --------                    --------

                        CAL  ECHO     /DISPLAY IT
                        RET           /RETURN TO CALLING ROUTINE
                SPACES, LAI  240      /SET UP ASCII CODE FOR SPACE IN ACC
                        CAL  ECHO     /DISPLAY A SPACE
                        LAI  240      /DO IT AGAIN - CODE FOR SPACE IN ACC
                        CAL  ECHO     /DISPLAY SPACE
                        RET           /RETURN TO CALLING ROUTINE
    "Z"         RESULT, CAL  DUMMY    /CAL RTN AT ADDRESS IN NEXT TWO BYTES!
                        CAL  FPOUT    /DISPLAY RESULT
                        JMP  FPCONT   /GO BACK AND GET NEXT PROBLEM!
"LOOKUP TABLE"          AAA           /LOW ADDRESS FOR START OF "FPADD" RTN
                        BBB           /PAGE ADDRESS FOR START OF "FPADD" RTN
                        CCC           /LOW ADDRESS FOR START OF "FPSUB" RTN
                        DDD           /PAGE ADDRESS FOR START OF "FPSUB" RTN
                        EEE           /LOW ADDRESS FOR START OF "FPMULT" RTN
                        FFF           /PAGE ADDRESS FOR START OF "FPMULT" RTN
    ***                 GGG           /LOW ADDRESS FOR START OF "FPDIV" RTN
                        HHH           /PAGE ADDRESS FOR START OF "FPDIV" RTN
```

THE THREE ROUTINES, "FPINP," "FPOUT," AND "FPCONT" AS PRESENTED WOULD REQUIRE ABOUT THREE PAGE OF MEMORY FOR STORAGE. HOWEVER, AS WILL BE DISCUSSED SHORTLY, THE ROUTINES COULD BE MODIFIED TO FIT INTO A CONSIDERABLY LESS AMOUNT OF MEMORY. THE DEMONSTRATION ROUTINES ALSO USED CERTAIN LOCATIONS ON PAGE 00 FOR STORAGE OF TRANSIENT DATA AND THESE ARE LISTED BELOW FOR REFERENCE. NATURALLY, THE ROUTINES COULD BE EASILY ALTERED TO USE OTHER TEMPORARY STORAGE LOCATIONS.

LOCATION(S)	USAGE
103	INPUT MANTISSA SIGN STORAGE
104	INPUT EXPONENT SIGN STORAGE
105	INPUT DIGIT COUNTER
106	INPUT "PERIOD" INDICATOR
107	OUTPUT DIGIT COUNTER
110	TEMP STORAGE FOR CONTROL "OPERATOR"
150 - 153	INPUT WORKING AREA
154 - 156	INPUT STORAGE REGISTERS (FOR DECBIN CONV)
157	INPUT EXPONENT (DECIMAL EQUIVELANT)
160 - 163	OUTPUT WORKING AREA
164 - 167	OUTPUT STORAGE REGISTERS (FOR BINDEC CONV)
170 - 173	TEMPORARY NUMBER STORAGE

TECHNIQUES FOR SHORTENING LENGTHY PROGRAMS

THE "FPINP," "FPOUT," AND "FPCONT" ROUTINES DESCRIBED PREVIOUSLY MIGHT APPEAR SOMEWHAT LENGTHY TO THE READER. INDEED THEY ARE BECAUSE MANY OF THE SECTIONS WERE DEVELOPED IN A MANNER THAT WOULD ENABLE ONE TO MORE EASILY FOLLOW THE LOGIC OF THE PROGRAM RATHER THAN TO SAVE MEMORY SPACE IN A COMPUTER SYSTEM. AS READERS KNOW, HOWEVER, IT IS OFTEN DESIRABLE TO REDUCE PROGRAMS TO FORMS THAT USE LESS MEMORY STORAGE. BUT, THERE ARE TRADE-OFFS TO CONSIDER. DESIGNING A PROGRAM TO MINIMIZE THE AMOUNT OF MEMORY USED GENERALLY REQUIRES SIGNIFICANTLY MORE HUMAN

PROGRAM DEVELOPMENT TIME, AND IT GENERALLY MAKES THE PROGRAM MORE "COMPLEX" OR DIFFICULT FOR SOMEONE ELSE TO UNDERSTAND, BECAUSE ONE OF THE FUNDAMENTAL TECHNIQUES IN REDUCING A PROGRAM'S LENGTH IS TO CAPITALIZE ON MAKING AS MANY "SUBROUTINES" OUT OF DIFFERENT SECTIONS OF THE PROGRAM AS POSSIBLE. THERE IS ALSO ANOTHER PARAMETER THAT CAN BE AFFECTED BY DESIGNING A PROGRAM TO USE LESS MEMORY - THE SPEED AT WHICH THE PROGRAM IS EXECUTED IS GENERALLY DECREASED BECAUSE A LOT OF EXTRA TIME IS SPENT EXECUTING TIME CONSUMING "CALL" INSTRUCTIONS. MORE DISCUSSION ON THE CONSIDERATIONS OF A PROGRAM'S OPERATING SPEED WILL BE PRESENTED IN A LATER CHAPTER.

PERHAPS THE FIRST RULE OF THUMB TO APPLY TOWARDS REDUCING THE AMOUNT OF MEMORY A PROGRAM REQUIRES IS TO MAXIMIZE THE AMOUNT OF SUBROUTINING UTILIZED PROVIDED THAT THE SUBROUTINING MEETS THE FOLLOWING SIMPLE MATHEMATICAL RELATIONSHIP:

$$B \times N > 3 \times N + B + 1$$

WHERE: "B" = THE NUMBER OF BYTES IN A REPEATED INSTRUCTION SEQUENCE
AND: "N" = THE NUMBER OF TIMES THE SEQUENCE IS USED IN THE PROGRAM

EXAMINING THE FORMULA ABOVE WILL SHOW THAT IT DOES NO GOOD IN TERMS OF CONSERVING MEMORY SPACE TO CALL A ROUTINE THAT UTILIZES ONLY 3 BYTES OF MEMORY. THIS IS BECAUSE A "CAL" INSTRUCTION ITSELF REQUIRES 3 BYTES OF MEMORY! HOWEVER, ONCE AN INSTRUCTION SEQUENCE EXCEEDS 3 BYTES OF MEMORY THE POINT AT WHICH SUBROUTINING BECOMES PROFITABLE FOR CONSERVING MEMORY SPACE IS A FUNCTION OF "N," THE NUMBER OF TIMES THE INSTRUCTION SEQUENCE NEEDS TO BE REPEATED IN A PROGRAM. FOR EXAMPLE, IF "B" = 4, ONE STARTS SAVING MEMORY SPACE BY SUBROUTINING WHEN "N" = 6. THE ABOVE FORMULA SHOWS THAT THE VALUE OF "N" REQUIRED TO MEET THE CONDITION WHERE MEMORY SPACE IS SAVED BY SUBROUTINING DROPS QUITE RAPIDLY AS "B" IS INCREASED SO THAT BY THE TIME ONE IS DEALING WITH INSTRUCTIONAL SEQUENCES WHICH USE 8 OR MORE BYTES OF MEMORY, ONE CAN SAVE MEMORY SPACE BY FORMING A SUBROUTINE IF THAT SAME SEQUENCE IS USED MORE THAN ONCE IN A PROGRAM! A SUMMARY OF THE MINIMUM VALUES OF "B" AND "N" THAT WILL RESULT IN MEMORY SPACE BEING SAVED BY SUBROUTINING BASED ON THE ABOVE FORMULA IS PROVIDED BELOW.

$B = 4$ AND $N = 6$
$B = 5$ AND $N = 5$
$B = 6$ AND $N = 3$
$B = 8$ AND $N = 2$

THE AMOUNT OF MEMORY SPACE THAT ONE SAVES BY APPROPRIATE SUBROUTINING CAN BE CHECKED BY REARRANGING THE ABOVE FORMULA:

$$B \times N - (3 \times N + B + 1) = Z$$

AND SOLVING FOR "Z," THE AMOUNT OF BYTES SAVED. FOR EXAMPLE, IF "B" IS 8 AND "N" IS 3, THEN "Z" IS:

$$8 \times 3 - (3 \times 3 + 8 + 1) = 6$$

WHEN DEVELOPING SUBROUTINES, ONE CAN OFTEN USE ONE ROUTINE TO SERVE SEVERAL FUNCTIONS BY ALLOWING FOR MULTIPLE ENTRY POINTS TO THE SUBROUTINE. AN EXAMPLE OF THIS METHOD WAS USED IN THE FLOATING-POINT PACKAGE DISCUSSED WHERE TWO ENTRY POINTS TO THE ROTATE SUBROUTINES WERE PROVIDED, SUCH AS THE "ROTATL" SUBROUTINE WHICH HAD A SECOND ENTRY POINT LABELED "ROTL" WHICH ALLOWED ONE TO ENTER THE ROUTINE BY "SKIPPING" THE "NDA" INSTRUCTION WHICH RESIDED IN THE LOCATION LABELED "ROTATL."

ANOTHER WAY TO OFTEN SAVE SIGNIFICANT AMOUNTS OF MEMORY IS BY CAREFUL ORGANIZATION OF THE PROGRAM AND ASSIGNMENT OF DATA STORAGE AREAS IN MEMORY. FOR EXAMPLE, THE READER MAY HAVE NOTED THAT ALL THE NUMERICAL DATA STORAGE AREAS USED IN THE FLOATING-POINT ROUTINES ALONG WITH THE COUNTERS AND INDICATORS STORED IN MEMORY WERE LOCATED ON PAGE ØØ. THIS WAS DONE TO MINIMIZE THE RESETTING OF THE PAGE POINTER (REGISTER "H"). SCATTERING DATA ON DIFFERENT PAGES OF MEMORY IN A LARGE PROGRAM CAN RESULT IN QUITE A BIT OF WASTED MEMORY BECAUSE REGISTER "H" MUST BE FREQUENTLY ALTERED (WHICH REQUIRES A TWO BYTE INSTRUCTION) TO CHANGE THE MEMORY POINTER ADDRESS. CAREFUL ORGANIZATION OF DATA STORAGE CAN EVEN BE HELPFUL IN MINIMIZING THE AMOUNT OF TIMES THAT REGISTER "L" MUST BE LOADED WITH A NEW ADDRESS (REQUIRING A TWO BYTE INSTRUCTION) BY LOCATING STORAGE AREAS IN ACCORDANCE WITH HOW THEY ARE ACCESSED IN A PROGRAM SEQUENCE SO THAT AN "INL" OR "DCL" (ONE BYTE COMMAND) MAY BE USED TO ACCESS A STORAGE LOCATION RATHER THAN AN "LLI XXX" INSTRUCTION.

IN LINE WITH THE ABOVE CONSIDERATIONS IS THE SIMPLE RULE TO MAINTAIN POINTERS AND COUNTERS AND OTHER FREQUENTLY USED "INDICATORS" IN CPU REGISTERS AS MUCH AS POSSIBLE. THIS CONSIDERABLY REDUCES THE NUMBER OF TIMES THAT THE "H & L" REGISTERS HAVE TO BE CHANGED TO "POINT" TO LOCATIONS THAT CONTAIN SUCH INFORMATION AND THEN CHANGED BACK TO HANDLE THE CURRENT DATA THAT IS BEING MANIPULATED.

ANOTHER GENERAL RULE OF THUMB TO FOLLOW FOR REDUCING PROGRAM MEMORY USAGE IS TO CAPITALIZE ON "LOOPS." A FORMULA FOR DETERMINING WHEN ONE CAN SAVE MEMORY SPACE BY USING A "LOOP" (ASSUMING THE LOOP COUNTER IS STORED IN A CPU REGISTER") IS PRESENTED HERE:

$$B \times N > B + 6$$

WHERE: "B" = THE NUMBER OF BYTES FORMING THE "REPEATED" PORTION OF THE SEQUENCE THAT MUST BE CONSECUTIVELY REPEATED.
AND: "N" = THE NUMBER OF TIMES THE SEQUENCE MUST BE CONSECUTIVELY REPEATED.

THUS, BY USING THE FORMULA, ONE CAN SEE THAT IF A PROGRAMMER HAS A FOUR BYTE INSTRUCTION THAT MUST BE CONSECUTIVELY REPEATED THE PROGRAMMER CAN SAVE MEMORY BY SETTING UP A "LOOP" IF THE SEQUENCE MUST BE CONSECUTIVELY REPEATED THREE OR MORE TIMES. IF "B" IS ONLY TWO, THEN A "LOOP" CONSERVES MEMORY IF IT MUST BE CONSECUTIVELY PERFORMED FIVE OR MORE TIMES. (THE ABOVE FORMULA IS DERIVED FROM THE FACT THAT IT REQUIRES SIX BYTES TO SET UP A "COUNTER," INCREMENT OR DECREMENT THE COUNTER EACH TIME A "LOOP" IS COMPLETED, AND MAKE A "CONDITIONAL" BRANCHING TEST).

A SUBTLE CONCEPT THAT CAN SAVE MEMORY SPACE INVOLVES THE POSSIBILITY OF INCLUDING A FEW CAREFULLY CHOSEN INSTRUCTIONS IN SUBROUTINES TO INCREASE THEIR GENERAL USEFULNESS. FOR EXAMPLE, CONSIDER THE SUBROUTINE ILLUSTRATED BELOW:

```
SAMPLE, LCH      /SAVE VALUE OF "H" IN "C"
        LHI XXX  /SET PNTR TO "DATA" PAGE
        LAM      /FETCH A BYTE OF "DATA"
        LHC      /RESTORE ORIG VALUE OF "H"
        NDA      /SET FLAGS FOR ACC CONTENTS
        RET
```

SUCH A SUBROUTINE MIGHT BE EXTREMELY VALUABLE IN A LARGE PROGRAM WHERE "DATA" WAS STORED ON ONE PAGE, BUT "COUNTERS" AND "INDICATORS" HAD TO BE STORED ON ANOTHER. BEFORE CALLING THE ABOVE ROUTINE, THE PROGRAM WOULD HAVE SET REGISTER "L" TO THE APPROPRIATE ADDRESS ON THE PAGE WHERE "DATA" WAS TO BE OBTAINED. SUPPOSE THAT SOMETIMES THE MAIN PRO-

GRAM NEEDED TO SIMPLY TRANSFER DATA FROM ONE LOCATION TO ANOTHER, AND AT OTHER TIMES IT MADE "TESTS" ON THE DATA IT OBTAINED. THE SIMPLE INCLUSION OF THE "NDA" INSTRUCTION IN THE ABOVE ROUTINE DOES NO HARM IN CASES WHERE DATA IS TO BE SIMPLY TRANSFERRED, BUT IT CAN SAVE VALUABLE MEMORY STORAGE IF THERE ARE TWO OR MORE TIMES IN WHICH THE DATA MUST BE "TESTED" IN THE MAIN PROGRAM BY HAVING THE "NDA" IN THE SUBROUTINE! FOR, THE "NDA" SETS UP THE FLAGS ALLOWING ONE TO IMMEDIATELY EXECUTE A CONDITIONAL BRANCHING INSTRUCTION UPON RETURN FROM THE SUBROUTINE WHEN DESIRED BASED ON THE "DATA" LOADED INTO THE ACCUMULATOR BY THE SUBROUTINE. TO PUSH THE POINT BEING MADE ONE STEP FURTHER - ADDING ONE MORE INSTRUCTION TO THE ABOVE SUBROUTINE - AN "INL" PLACED JUST BEFORE THE "NDA" INSTRUCTION COULD MAKE THE ROUTINE EVEN MORE "GENERAL PURPOSE." FOR INSTANCE, IN A TYPICAL DATA MANIPULATING PROGRAM ONE MIGHT BE SEQUENTIALLY ACCESSING LOCATIONS IN THE "DATA" STORAGE AREA WHILE POSSIBLY SEARCHING FOR A CERTAIN "CODE." AT OTHER TIMES ONE MIGHT BRANCH OFF TO PERFORM WORK IN ANOTHER AREA OF MEMORY IN WHICH CASE ONE WOULD PROBABLY HAVE TO PERFORM AN "LLI XXX" INSTRUCTION. THUS, THE INCLUSION OF THE "INL" COMMAND IN THE SUBROUTINE TAKES CARE OF ALL THE TIMES THAT ONE NEEDS TO ACCESS THE NEXT LOCATION IN THE "DATA" AREA, YET DOES NO HARM IF THE PROGRAM WILL BE DIRECTED TO A DIFFERENT MEMORY AREA! (NOTE, HOWEVER, THAT ONE WOULD HAVE TO EXAMINE CAREFULLY, HOW OFTEN THE MAIN PROGRAM MIGHT BE REQUIRED TO ACCESS THE EXACT SAME LOCATION AGAIN, THUS REQUIRING A COMPENSATING "DCL" INSTRUCTION IN THE MAIN PORTION OF THE PROGRAM!)

HOWEVER, ONE OF THE MOST POWERFUL MEMORY SAVING TECHNIQUES FOR 8008 SYSTEMS IS BASED ON THE USE OF A CLASS OF INSTRUCTIONS THAT MANY NOVICE PROGRAMMERS COMPLETELY OVERLOOK! THIS CLASS OF INSTRUCTIONS IS THE "RESTART" (RST XXX) GROUP. FOR, WHILE THE MNEMONIC FOR A "RESTART" INSTRUCTION IS SHOWN AS CONSISTING OF TWO PARTS, THE ACTUAL COMMAND IS AN EFFECTIVE ONE BYTE "CALL" INSTRUCTION! WHILE THE "RST" COMMANDS WERE INCLUDED IN THE 8008 INSTRUCTION SET TO FACILITATE IMPLEMENTING "START-UP" OPERATIONS IN CONJUNCTION WITH THE "INTERRUPT" FACILITY ON TYPICAL 8008 SYSTEMS, THEY MAY ALSO BE PUT TO EXTREMELY EFFECTIVE USAGE IN GENERAL PROGRAMMING APPLICATIONS. THE REASON IS EASY TO UNDERSTAND ONCE IT HAS BEEN POINTED OUT - BEING ABLE TO "CALL" A SUBROUTINE WITH A ONE BYTE INSTRUCTION INSTEAD OF A THREE BYTE INSTRUCTION CAN SAVE A LARGE AMOUNT OF MEMORY SPACE IF A ROUTINE HAS TO BE "CALLED" FREQUENTLY IN A PROGRAM.

THE READER SHOULD REVIEW THE MATERIAL ON PAGE 17 OF THE CHAPTER WHICH EXPLAINS THE 8008 INSTRUCTION SET IN THIS MANUAL PERTAINING TO THE "RESTART" INSTRUCTIONS. SINCE THERE ARE 8 "RESTART" LOCATIONS ON PAGE 00, THAT MEANS THAT ONE CAN HAVE UP TO EIGHT DIFFERENT SUBROUTINES IN A PROGRAM THAT CAN BE ACCESSED WITH BUT A ONE BYTE CALL! WHILE THE "RESTART" LOCATIONS ARE SPACED BUT 8 (DECIMAL) LOCATIONS APART, ONE CAN STILL USE THE "RESTART" LOCATIONS FOR REACHING THE DESIRED OBJECTIVE OF SAVING MEMORY SPACE EVEN IF THE DESIRED SUBROUTINE WILL NOT FIT IN THE 8 LOCATIONS BY SIMPLY HAVING A "JUMP" INSTRUCTION AT A RESTART LOCATION THAT DIRECTS THE PROGRAM TO THE ACTUAL SUBROUTINE!

TO SEE THE IMPORTANCE OF USING "RST" COMMANDS IN LARGE PROGRAMS CONSIDER THE FACT THAT IT MAY OFTEN BE NECESSARY TO CALL A PARTICULAR SUBROUTINE 30 OR 40 (DECIMAL) TIMES. USING A ONE BYTE "RESTART" INSTRUCTION INSTEAD OF A THREE BYTE "CAL" COMMAND CAN THUS SAVE 60 TO 80 (DECIMAL) MEMORY LOCATIONS. THAT IS ROUGHLY ONE-FOURTH OF A "PAGE" OF MEMORY IN AN 8008 SYSTEM! MULTIPLY THAT BY A FACTOR OF 8 - THE NUMBER OF "RST" LOCATIONS AVAILABLE - AND ONE CAN SEE A VERY CONSIDERABLE SAVINGS IN MEMORY USAGE! THE PERSON WHO HAS DEVELOPED FAIRLY DECENT SIZED PROGRAMS FOR AN 8008 SYSTEM WITHOUT TAKING ADVANTAGE OF THE "RST" COMMANDS TO CONSERVE MEMORY IS OFTEN AMAZED WHEN SUCH PROGRAMS ARE RE-WRITTEN TO

UTILIZE THE TECHNIQUE AND THE PROGRAMMER FINDS MEMORY USAGE CUT BY A CONSIDERABLE PERCENTAGE!

AS A CHALLENGE TO THE READER WHO IS INTERESTED IN DOING A LITTLE CREATIVE "TRIMMING" OF A PROGRAM, WHY NOT GO TO WORK ON REDUCING THE SIZE OF THE "FPINP," "FPOUT," AND "FPCONT" ROUTINES PRESENTED IN THIS CHAPTER? USING THE TECHNIQUES DESCRIBED IN THE LAST SEVERAL PAGES, ONE SHOULD BE ABLE TO WORK THOSE ROUTINES DOWN FROM THE ROUGHLY THREE PAGES OF MEMORY THEY REQUIRE AS PRESENTED, TO WITHIN ABOUT TWO PAGES!

INPUT/OUTPUT PROGRAMMING

THIS CHAPTER WILL BE CONCERNED WITH DISCUSSING PROGRAMMING TECHNIQUES FOR TRANSFERRING INFORMATION TO AND FROM THE COMPUTER AND EXTERNAL DEVICES. EXTERNAL DEVICES ARE CONNECTED TO THE COMPUTER IN AN 8008 SYSTEM VIA PHYSICAL CONNECTIONS WHICH CARRY ELECTRONIC SIGNALS. SINCE IT IS OFTEN DESIRABLE TO HAVE A NUMBER OF DIFFERENT DEVICES CONNECTED TO A SYSTEM AT ONE TIME, A HARDWARE ARRANGEMENT IS GENERALLY PROVIDED THAT ENABLES A NUMBER OF DEVICES TO BE CONNECTED AT ONE TIME, BUT ONLY ONE SUCH DEVICE MAY ACTUALLY "COMMUNICATE" WITH THE COMPUTER AT ANY GIVEN INSTANT OF TIME. TO ALLOW CONTROL OF WHICH DEVICE IS ABLE TO COMMUNICATE WITH THE COMPUTER, AN ELECTRONIC ARRANGEMENT IS PROVIDED THAT ALLOWS "SOFTWARE" SELECTION OF INPUT AND OUTPUT "PORTS." AS FAR AS A PROGRAMMER IS CONCERNED, A "PORT" CONSISTS OF EIGHT SEPARATE ELECTRONIC SIGNALS THAT CAN BE IN A "1" OR "0" STATE. THE EIGHT SIGNALS CORRESPOND TO THE EIGHT BIT POSITIONS AVAILABLE IN THE ACCUMULATOR OF THE CPU. AN "INPUT" PORT ACCEPTS INFORMATION FROM AN EXTERNAL DEVICE AND PRESENTS IT TO THE ACCUMULATOR OF AN 8008. AN "OUTPUT" PORT TAKES INFORMATION FROM THE ACCUMULATOR AND PASSES IT TO AN OUTPUT DEVICE. THE SELECTION OF A PARTICULAR INPUT OR OUTPUT PORT IS SPECIFIED BY THE PROGRAMMER WHEN UTILIZING AN I/O COMMAND. THE READER MAY DESIRE TO REVIEW THE DISCUSSION OF THE I/O INSTRUCTIONS PRESENTED ON PAGE 18 OF THE CHAPTER DESCRIBING THE INSTRUCTION SET FOR THE 8008 CPU AT THIS TIME.

> NOTE: FOR THE PURPOSES OF THE DISCUSSION IN THIS CHAPTER, ALL I/O OPERATIONS WILL BE ASSUMED TO TAKE PLACE BETWEEN THE I/O "PORTS" AND THE ACCUMULATOR OF THE CPU. WHILE SOME READERS MAY BE AWARE THAT IT IS POSSIBLE TO COMMUNICATE WITH A COMPUTER VIA TECHNIQUES KNOWN AS "DIRECT MEMORY ACCESS, WHEREBY AN EXTERNAL DEVICE PLACES DATA DIRECTLY INTO AREAS IN MEMORY, OR VICE-VERSA, SUCH CAPABILITY IS RARELY FOUND ON 8008 BASED SYSTEMS. FURTHERMORE, SUCH TRANSFER TECHNIQUES ARE ESSENTIALLY "HARDWARE CONTROLLED" AND ARE OUTSIDE THE PURELY PROGRAMMING REALM TO WHICH THIS MANUAL IS DEVOTED.

THE BASIC CONCEPT BEHIND COMMUNICATING WITH A COMPUTER LIES IN PROVIDING SOME FORM OF SYSTEMATIC SYSTEM FOR ENCODING INFORMATION FROM AN EXTERNAL DEVICE THAT WILL ALLOW A PROGRAM TO DECODE THE INFORMATION AND TAKE APPROPRIATE ACTION, AND TO ALLOW A PROGRAM TO SEND CODES TO AN EXTERNAL DEVICE THAT WILL DIRECT IT TO PERFORM IN A PRESCRIBED MANNER.

SUCH A SYSTEM CAN BE CREATED ENTIRELY BY THE PROGRAMMER. INDEED, IN MANY SPECIAL APPLICATIONS, SUCH AS CONTROLLING A UNIQUE PIECE OF MACHINERY, THAT IS JUST THE APPROACH TAKEN. FOR EXAMPLE, SUPPOSE SOME MANUFACTURER HAD A MACHINE THAT WAS TO BE CONTROLLED BY THE COMPUTER. THE MACHINE COULD BE CONSTRUCTED SO THAT WHEN IT WAS PERFORMING A CERTAIN TYPE OF FUNCTION IT WOULD CLOSE A PARTICULAR ELECTRICAL SWITCH. THERE MIGHT BE A NUMBER OF SUCH SWITCHES ON THE MACHINE AND EACH ONE COULD BE CONNECTED TO AN INPUT LINE, REPRESENTING ONE "BIT" OF AN INPUT PORT. FOR THE SAKE OF DISCUSSION, SUPPOSE A MACHINE HAD EIGHT SUCH INPUT SWITCHES, ONE CONNECTED TO EACH POSSIBLE LINE MAKING UP AN INPUT PORT. WHEN THE SWITCH WAS "CLOSED" A "1" CONDITION WOULD BE PLACED ON THE LINE AND WHEN IT WAS "OPEN" THE LINE WOULD REPRESENT A "0" CONDITION. FOR THE SAKE OF SIMPLICITY, IT COULD ALSO BE ASSUMED THAT ONLY ONE SWITCH COULD BE CLOSED AT ANY GIVEN TIME.

NOW, ASSUME THE COMPUTER WAS TO MONITOR THE STATUS OF THE SWITCHES

BY PERIODICALLY EXECUTING AN INPUT INSTRUCTION FOR THE INPUT PORT TO WHICH THE SWITCHES WERE ATTACHED. THEN, DEPENDING ON WHICH SWITCH WAS IN THE CLOSED CONDITION, THE COMPUTER WOULD DIRECT INFORMATION TO BE OUTPUTTED ON AN OUTPUT PORT, SAY, TO DIRECT ANOTHER PART OF THE MACHINE TO PERFORM A SPECIFIC OPERATION. A PROGRAMMER MIGHT MAKE UP AN "INPUT" PROGRAM IN THE FOLLOWING MANNER.

```
                MNEMONIC                    COMMENTS
            --------------              --------------

INCTRL,  INP X          /READ DATA FROM PORT X INTO ACCUMULATOR
         NDA            /SET FLAGS AFTER INPUT OPERATION
         JTZ INCTRL     /NO SWITCHES CLOSED - KEEP LOOKING
         CPI 001        /IS IT SWITCH #1?
         JTZ START1     /YES, DO REQUIRED ROUTINE
         CPI 002        /IS IT SWITCH #2?
         JTZ START2     /YES, DO REQUIRED ROUTINE
         CPI 004        /IS IT SWITCH #3?
         JTZ START3     /YES, DO REQUIRED ROUTINE
         CPI 010        /IS IT SWITCH #4?
         JTZ START4     /YES, DO REQUIRED ROUTINE
          .
          .
          .
         CPI 200        /IS IT SWITCH #8?
         JTZ START8     /YES, DO REQUIRED ROUTINE
         JMP ERROR      /IF PROGRAM EVER GETS HERE SOMETHING WRONG
```

THE ABOVE INPUT ROUTINE IS QUITE SIMPLE AND LACKS A TECHNICAL CONSIDERATION THAT MIGHT BE NECESSARY IN A REAL SYSTEM (HOW CAN THE ROUTINE TELL WHETHER A READING INDICATES A "NEW" SWITCH CLOSURE OR A "PREVIOUS" CONDITION STILL PRESENT?) HOWEVER, IT DOES ILLUSTRATE THE CONCEPT OF INPUTTING INFORMATION AND HAVING THE COMPUTER INTERPRET THAT INFORMATION.

IN A SIMILAR MANNER TO THE INPUT ROUTINE, ONE COULD CONNECT, SAY, THE COILS OF ELECTRONIC RELAYS TO THE OUTPUT LINES OF A SPECIFIC OUTPUT PORT. EACH OF THE EIGHT POSSIBLE LINES CONNECTED TO AN OUTPUT PORT COULD ACTIVATE THE ASSOCIATED RELAY WHEN A "1" CONDITION WAS PRESENT, BUT NOT WHEN A "0" CONDITION EXISTED. SINCE EACH LINE CORRESPONDS TO ONE "BIT" IN THE ACCUMULATOR, ONE COULD EASILY DEVELOP A PROGRAM TO CONTROL THE OPERATION OF THE RELAYS BY PLACING APPROPRIATE CODES IN THE ACCUMULATOR OF THE CPU AND THEN EXECUTING AN "OUT Z" INSTRUCTION WHERE "Z" REPRESENTED THE OUTPUT PORT WHOSE LINES WERE CONNECTED TO THE RELAYS.

IN THE ABOVE EXAMPLE INPUT PROGRAM TO MONITOR THE STATUS OF A SET OF SWITCHES IT WAS ASSUMED THAT ONLY ONE SWITCH COULD BE CLOSED AT A GIVEN TIME. THUS, THERE WERE ONLY NINE POSSIBLE SIGNAL CONDITIONS THAT COULD BE RECEIVED BY THE COMPUTER - ANY ONE OF THE EIGHT SWITCHES, EACH REPRESENTED BY THE STATUS OF A PARTICULAR BIT IN THE ACCUMULATOR, COULD BE "ON," OR NONE OF THEM WERE ACTIVATED. THUS, THE PARTICULAR CODING TECHNIQUE FOR THE EXAMPLE WAS REALLY QUITE LIMITED. HAD IT BEEN STATED THAT ANY NUMBER OF THE SWITCHES COULD BE "ON" AT ANY GIVEN TIME, THEN THERE WOULD BE 256 DIFFERENT CODES POSSIBLE ON THE 8 INPUT LINES AT ANY GIVEN TIME! SUCH AN ENCODING SCHEME WOULD ALLOW QUITE A LOT MORE INFORMATION TO BE CONVEYED TO THE COMPUTER ON ONE INPUT PORT. ONE COULD READILY ENVISION COMING UP WITH A SYSTEM WHEREBY AN EXTERNAL MACHINE COULD USE THE 256 POSSIBLE STATES AVAILABLE ON ONE INPUT PORT TO PROVIDE A LOT OF INFORMATION TO THE COMPUTER. BY ASSIGNING DIFFERENT CODES TO REPRESENT DIFFERENT "ARTIFACTS" ONE COULD EASILY COME UP WITH A DEVICE THAT COULD ESSENTIALLY ENCODE ALL THE LETTERS OF THE ALPHABET, THE NUMBERS 0 - 9,

AND A LOT OF SPECIAL SYMBOLS AND STILL HAVE UNUSED STATES! WELL, AS THE READER UNDOUBTABLY KNOWS, PEOPLE DEVELOPED SUCH ENCODING SYSTEMS QUITE SOME TIME AGO. IN FACT, A NUMBER OF DIFFERENT "STANDARDIZED" ENCODING SYSTEMS HAVE BEEN DEVELOPED OVER THE YEARS. ONE OF THE MOST POPULAR ENCODING SYSTEMS, ONE THAT IS USED ON MANY KINDS OF MACHINES SUCH AS ELECTRONIC KEYBOARDS, TYPEWRITER, NUMBERICAL CONTROL MACHINES AND IN A VARIETY OF COMMUNICATION DEVICES, IS COMMONLY ABBREVIATED AND REFERRED TO AS THE "ASCII" CODE. "ASCII" IS THE ABBREVIATION FOR "AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE." "ASCII" CODE ITSELF IS ACTUALLY DESIGNED TO USE JUST 7 BITS OF INFORMATION (THUS ALLOWING FOR THE ENCODING OF 128 DIFFERENT "SYMBOLS"), HOWEVER, "ASCII" CODE IS OFTEN USED IN DEVICES THAT USE 8 BITS BECAUSE THE LAST BIT OF DATA CAN BE USED TO TEST FOR TRANSMISSION ERRORS BY SERVING AS A "PARITY" INDICATOR. MORE WILL BE SAID ABOUT "PARITY" A LITTLE LATER.

WHILE THE ENTIRE "ASCII" CODE IS BASED ON THE DIFFERENT PATTERNS THAT WILL FIT IN SEVEN BITS OF A REGISTER, THUS YIELDING 128 (DECIMAL) DIFFERENT "CODES," A COMMONLY USED "SUBSET" OF THE ASCII CODE IS OFTEN UTILIZED. THE "SUBSET" DOES NOT USE EVERY POSSIBLE PATTERN BUT ONLY THOSE PATTERNS DESIRED. THE "SUBSET" REFERRED TO IS FREQUENTLY USED IN "ASCII" CODED KEYBOARDS, TELETYPE MACHINES, AND OTHER DEVICES. IN THE LISTING SHOWN BELOW, THE 8'TH BIT NOT USED BY THE "ASCII" CODE WILL BE SHOWN AS A "1" CONDITION AND THE CODES WILL BE PRESENTED AS THEY COULD APPEAR IN THE REGISTERS OF AN 8008 CPU.

CHARACTERS SYMBOLIZED	BINARY CODE	OCTAL REP	CHARACTERS SYMBOLIZED	BINARY CODE	OCTAL REP
A	11 000 001	301	!	10 100 001	241
B	11 000 010	302	"	10 100 010	242
C	11 000 011	303	#	10 100 011	243
D	11 000 100	304	$	10 100 100	244
E	11 000 101	305	%	10 100 101	245
F	11 000 110	306	&	10 100 110	246
G	11 000 111	307	'	10 100 111	247
H	11 001 000	310	(	10 101 000	250
I	11 001 001	311	)	10 101 001	251
J	11 001 010	312	*	10 101 010	252
K	11 001 011	313	+	10 101 011	253
L	11 001 100	314	,	10 101 100	254
M	11 001 101	315	-	10 101 101	255
N	11 001 110	316	.	10 101 110	256
O	11 001 111	317	/	10 101 111	257
P	11 010 000	320	0	10 110 000	260
Q	11 010 001	321	1	10 110 001	261
R	11 010 010	322	2	10 110 010	262
S	11 010 011	323	3	10 110 011	263
T	11 010 100	324	4	10 110 100	264
U	11 010 101	325	5	10 110 101	265
V	11 010 110	326	6	10 110 110	266
W	11 010 111	327	7	10 110 111	267
X	11 011 000	330	8	10 111 000	270
Y	11 011 001	331	9	10 111 001	271
Z	11 011 010	332	:	10 111 010	272
[	11 011 011	333	;	10 111 011	273
\	11 011 100	334	<	10 111 100	274
]	11 011 101	335	=	10 111 101	275
↑	11 011 110	336	>	10 111 110	276
←	11 011 111	337	?	10 111 111	277
SPACE	11 100 000	240	@	11 000 000	300

THE SUBSET OF THE "ASCII" CODE JUST PRESENTED HAS SEVERAL NICE FEATURES WORTH NOTING. FOR INSTANCE, THE 26 LETTERS OF THE ALPHABET ARE ALL ENCODED IN A SEQUENCE STARTING WITH 301 (OCTAL) AND ENDING WITH 332 (OCTAL). THUS ONE CAN EASILY CHECK DATA, FOR EXAMPLE, BEING INPUTTED BY AN OPERATOR TO SEE IF THE CODE BEING RECEIVED REPRESENTS A LETTER OF THE ALPHABET BY PERFORMING A "RANGE TEST" AS ILLUSTRATED BELOW.

MNEMONIC		COMMENTS
CKALFA,	INP X	;ACCEPT A CHARACTER FROM INPUT DEVICE
	CPI 301	;SEE IF INPUT IN RANGE FROM 301
	JTS CKALFA	;TO 332, IF IT IS NOT, IGNORE THE
	CPI 333	;INPUT, IF IT IS WITHIN THE RANGE
	JFS CKALFA	;THEN HAVE AN ALPHABETICAL CHARACTER
;ALFA,	...	;TO PROCESS AS DESIRED

THE READER MAY NOTE THAT THE NUMBERS 0 THROUGH 9 ARE ALSO GROUPED TOGETHER IN THE SEQUENCE FROM 260 TO 271 AND THE PROGRAMMER CAN THUS READILY PERFORM A SIMILAR RANGE TEST TO ONLY ACCEPT NUMBERS.

THERE ARE SEVERAL OTHER "CHARACTERS" THAT ARE USED BY MANY MACHINES THAT OPERATE WITH ASCII CODE THAT WILL BE MENTIONED FOR REFERENCE. THE FUNCTIONS "CARRIAGE-RETURN" (215), "LINE-FEED" (212), "BELL" (207) AND "ROUBOUT" (377), ARE MOST OFTEN FOUND ON TELETYPE MACHINES WHICH MAKE VERY NICE I/O DEVICES FOR A COMPUTER.

WHEN AN INPUT INSTRUCTION IS EXECUTED, THE COMPUTER WILL RECEIVE EIGHT BITS OF INFORMATION SIMULTAINEOUSLY - CORRESPONDING TO THE EIGHT POSSIBLE LINES OF AN INPUT PORT WHICH ARE FED INTO THE ACCUMULATOR. IN OTHER WORDS, THE DATA IS ACCEPTED IN PARALLEL. LIKEWISE, WHEN AN OUTPUT INSTRUCTION IS EXECUTED, THE COMPUTER WILL SEND ALL EIGHT BITS OF THE ACCUMULATOR OUT TO THE APPROPRIATE OUTPUT PORT SIMULTAINEOUSLY. HOWEVER, SOME DEVICES WHICH ONE DESIRES TO OPERATE WITH THE COMPUTER MAY NOT BE "PARALLEL" DEVICES. THEY MAY INSTEAD BE "SERIALLY" OPERATED WHICH MEANS THEY DO NOT TRANSMIT INFORMATION OVER A GROUP OF WIRES, BUT RATHER SEND THE INFORMATION "ONE BIT AT A TIME" OVER A SINGLE WIRE. SUCH DEVICES MAY, HOWEVER, STILL BE CONNECTED TO AN 8008 SYSTEM SINCE ONE MAY SIMPLY "DISCARD" THE UNUSED BITS CORRESPONDING TO UNUSED LINES OF AN I/O PORT. IN SUCH CASES, THE PROGRAMMER MUST KNOW WHICH LINE OF A PORT IS THE "ACTIVE" LINE AND TAKE CARE TO ENSURE THAT THE PROGRAM MANIPULATES BITS OF INFORMATION SO THAT THEY APPEAR ON THAT LINE AT THE PROPER TIME. WHETHER A PARTICULAR DEVICE CONNECTED TO A COMPUTER IS "SERIAL" OR "PARALLEL" IN OPERATION (AS FAR AS THE COMPUTER IS CONCERNED) IS OFTEN A FUNCTION OF THE TYPE OF HARDWARE INTERFACE PROVIDED FOR THE EXTERNAL DEVICE. FOR INSTANCE, TELETYPE MACHINES ARE ESSENTIALLY "SERIAL" DEVICES SINCE THEY ACT ON INFORMATION ONE BIT AT A TIME. HOWEVER, WHEN ACTUALLY CONNECTED TO A COMPUTER ONE CAN ELECT TO HAVE A "HARDWARE" INTERFACE THAT CONVERTS INFORMATION RECEIVED FROM THE MACHINE IN SERIAL FORM AND PLACES IT IN A "PARALLEL" REGISTER BEFORE PASSING THE DATA TO THE COMPUTER, AND GOING IN THE OTHER DIRECTION, HAVE THE COMPUTER SEND DATA IN PARALLEL FORM TO THE INTERFACE WHICH WILL THEN PASS IT ON TO THE MACHINE IN BIT-SERIAL FASHION. SUCH AN INTERFACE CAN SAVE A LOT OF COMPUTER TIME BECAUSE THE EXTERNAL HARDWARE INTERFACE IS ABLE TO HANDLE THE TIME CONSUMING SERIAL TO PARALLEL AND PARALLEL TO SERIAL TASKS. HOWEVER, SUCH HARDWARE COSTS MONEY, AND IN MANY APPLICATIONS ONE MAY DESIRE TO HAVE THE COMPUTER DO THE SERIAL TO PARALLEL CONVERSION AND VICE-VERSA. THIS CAN BE ACCOMPLISHED QUITE READILY WITH A SUITABLE PROGRAM THAT ACTUALLY UTILIZES THE COMPUTER'S OWN TIMING TO DETERMINE WHEN

TO "LOOK" OR "SAMPLE" FOR THE NEXT BIT OF INFORMATION FROM THE SERIAL DEVICE OR WHEN TO SEND THE NEXT BIT OF INFORMATION TO THE SERIAL DEVICE. WHILE THE DETAILS OF CAREFULLY CONTROLLING THE TIMING FOR SUCH A PROGRAM WILL BE DISCUSSED IN THE NEXT CHAPTER, THE CONCEPT OF HAVING THE COMPUTER PERFORM PARALLEL TO SERIAL OR SERIAL TO PARALLEL CONVERSION WILL BE DEMONSTRATED WITH SEVERAL ROUTINES AT THIS POINT. THE TECHNIQUE CONSIST OF USING ACCUMULATOR "ROTATE" INSTRUCTIONS TO SHIFT THE SERIAL DATA IN OR OUT OF THE COMPUTER.

IN THE PARALLEL TO SERIAL ROUTINE SHOWN NEXT, IT WILL BE ASSUMED THAT A DEVICE THAT ACCEPTS SERIAL DATA IS CONNECTED TO THE LEAST SIGNIFICANT BIT LINE OF OUTPUT PORT "X" AND THAT THE REMAINING LINES AVAILABLE ON THE PORT ARE UNUSED. THE DEVICE WILL BE ASSUMED TO BE A UNIT THAT OPERATES WITH "ASCII" CODE AND BEFORE THE ILLUSTRATED ROUTINE IS "CALLED" THAT THE CODE FOR A CHARACTER HAS BEEN PLACED IN THE ACCUMULATOR.

```
                MNEMONIC                    COMMENTS
            ----------------            ----------------

PARSER,  LCI 010            /SET UP REGISTER "C" AS A BIT COUNTER
NEXOUT,  OUT X              /OUTPUT DATA IN ACC TO PORT X, ONLY THE
         RRC                /DATA IN LSB USED, NOW ROTATE ACC RIGHT
         DCC                /IGNORE CARRY THEN DECREMENT BIT COUNTER
         JFZ NEXBIT         /DO NEXT BIT IF CNTR NOT ZERO
         RET                /EXIT RTN WHEN ALL 8 BITS TRANSMITTED
```

IN THE FOLLOWING SERIAL TO PARALLEL ROUTINE IT IS ASSUMED THAT DATA IS ARRIVING AT THE MOST SIGNIFICANT BIT POSITION OF AN INPUT PORT AND THAT IT IS TO BE ASSEMBLED INTO AN EIGHT BIT FORMAT.

```
                MNEMONIC                    COMMENTS
            ----------------            ----------------

SERPAR,  XRA                /CLEAR ACCUMULATOR AND ALSO CLEAR
         LBA                /REGISTER "B" AT START OF ROUTINE
         LCI 010            /SET A BIT COUNTER
NEXTIN,  INP X              /BRING IN DATA FROM INPUT PORT X
         NDI 200            /SINCE ONLY MSB HAS IMPORTANT DATA, MASK
         RAL                /OFF OTHER BITS & CLR CARRY, NOW ROTATE
         ADB                /LEFT TO SAVE NEW BIT, THEN ADD IN ANY
         RAR                /PREVIOUS BITS FROM "B" AND ROTATE RIGHT
         LBA                /TO ADD ON LATEST BIT, STORE IN "B"
         DCC                /DECREMENT BIT COUNTER
         JFZ NEXTIN         /IF NOT FINISHED, GET NEXT BIT
         RET                /EXIT RTN WHEN 8 BITS RECEIVED & STORED
```

ANOTHER POPULAR "STANDARDIZED" CODE FOR OPERATING I/O DEVICES IS KNOWN AS "BAUDOT" CODE. BAUDOT CODE IS A "5 LEVEL" CODE IN THAT IT REQUIRES FIVE BITS TO SPECIFY A PARTICULAR CHARACTER. THUS, THERE ARE THEORETICALLY 32 DIFFERENT PATTERNS THAT CAN BE REPRESENTED WHEN USING BAUDOT CODE. NOW, BAUDOT CODE HAS LONG BEEN USED IN A VARIETY OF TELETYPES AND OTHER COMMUNICATION DEVICES AND THE CODE IS OF INTEREST TO MANY COMPUTER OWNERS BECAUSE OLDER MODEL TELETYPE MACHINES, PAPER TAPE PUNCHES AND PAPER TAPE READERS CAN OFTEN BE OBTAINED FROM SECOND HAND SOURCES AT QUITE REASONABLE PRICES, AND USED AS AN I/O DEVICE FOR A COMPUTER. WHILE BAUDOT CODE CAN ONLY REPRESENT 32 DIFFERENT BIT PATTERNS,

THESE MACHINES CAN PRINT ALL THE LETTERS OF THE ALPHABET, THE NUMBERS 0 THROUGH 9, AND A VARIETY OF PUNCTUATION SYMBOLS! THAT IS A LOT MORE THAN 32 DIFFERENT CHARACTERS! HOW IS IT DONE?

WELL, THE DESIGNERS OF THOSE MACHINES USED A LITTLE INGENUITY TO ENABLE THE MACHINE TO HANDLE ALMOST DOUBLE THE NUMBER OF CHARACTERS THAT COULD BE REPRESENTED BY A FIVE BIT CODE BY USING SEVERAL OF THE CODES TO "SHIFT" THE MACHINE BETWEEN TWO MODES, SO THAT IN ONE MODE IT WOULD INTERPRET THE CODES TO MEAN ONE SET OF CHARACTERS AND IN THE OTHER MODE IT WOULD INTERPRET THE CODES TO REPRESENT A DIFFERENT SET OF CHARACTERS. IN ONE MODE, TERMED THE "LETTERS" MODE, ALL THE LETTERS OF THE ALPHABET MAY BE PRINTED. IN THE "FIGURES" MODE, NUMBERS AND PUNCTUATION ARE PRINTED. THE "BAUDOT" CODE IS PRESENTED BELOW.

CHARACTERS LC	CHARACTERS UC	5 LEVEL CODE BIT POSITION	OCTAL CODES
A	-	0 0 0 1 1	003
B	?	1 1 0 0 1	031
C	:	0 1 1 1 0	016
D	$	0 1 0 0 1	011
E	3	0 0 0 0 1	001
F	!	0 1 1 0 1	015
G	&	1 1 0 1 0	032
H	#	1 0 1 0 0	024
I	8	0 0 1 1 0	006
J	'	0 1 0 1 1	013
K	(	0 1 1 1 1	017
L	)	1 0 0 1 0	022
M	.	1 1 1 0 0	034
N	,	0 1 1 0 0	014
O	9	1 1 0 0 0	030
P	0	1 0 1 1 0	026
Q	1	1 0 1 1 1	027
R	4	0 1 0 1 0	012
S	BELL	0 0 1 0 1	005
T	5	1 0 0 0 0	020
U	7	0 0 1 1 1	007
V	;	1 1 1 1 0	036
W	2	1 0 0 1 1	023
X	/	1 1 1 0 1	035
Y	6	1 0 1 0 1	025
Z	"	1 0 0 0 1	021
SPACE		0 0 1 0 0	004
CAR. RET.		0 1 0 0 0	010
LINE FEED		0 0 0 1 0	002
NULL		0 0 0 0 0	000
FIGURES		1 1 0 1 1	033
LETTERS		1 1 1 1 1	037

IN THE BAUDOT TABLE SHOWN ABOVE THE OCTAL CODES COLUMN WAS SHOWN ASSUMING THAT THE CODES WERE STORED IN THE LEAST SIGNIFICANT BIT POSITIONS OF AN 8008 REGISTER WITH THE THREE MOST SIGNIFICANT BITS SET TO 0. THE READER CAN NOW SEE THAT 26 OF THE POSSIBLE 32 CODES CAN REPRESENT TWO DIFFERENT CHARACTERS DEPENDING ON WHICH MODE THE MACHINE IS IN. THE FUNCTIONS "SPACE," "CARRIAGE-RETURN," "LINE-FEED," AND "NULL" MEAN THE SAME REGARDLESS OF WHICH MODE THE MACHINE IS IN, AND TWO CODES "FIGURES" AND "LETTERS" ARE USED TO SWITCH THE MODE OF THE MACHINE. WHILE EVERYTHING MAY SEEM FINE AT THIS POINT, IT IS IMPORTANT TO DISCUSS HANDLING

THE CODE AS PART OF AN I/O ROUTINE BECAUSE THERE IS A SUBTLE FACTOR THAT CAN BE OVER-LOOKED BY SOME BEGINNING PROGRAMMERS!

IN ACTUAL OPERATION, A BAUDOT TELETYPE OPERATES IN THE "MODE" THAT IT WAS LAST PLACED IN BY A "FIGURES" OR "LETTERS" KEY AND REMAINS IN THAT MODE UNTIL THE OPPOSITE MODE CODE IS RECEIVED. THUS, A MECHANICAL ARRANGEMENT ACTUALLY SERVES TO "REMEMBER" A "BIT" OF INFORMATION. THE FACT THAT AN EXTERNAL MECHANICAL LINKAGE IS USED TO HOLD A "BIT" OF INFORMATION MUST BE TAKEN IN ACCOUNT IF A COMPUTER PROGRAM IS TO PROCESS THE CODE WITH PRACTICAL RESULTS!

FOR INSTANCE, IF ONE HAD AN INPUT ROUTINE THAT SIMPLY LOOKED FOR A FIVE BIT PATTERN FROM A BAUDOT DEVICE ONE COULD GET THAT PATTERN IN MANY INSTANCES FROM TWO POSSIBLE CONDITIONS OF THE TELETYPE MACHINE. FOR INSTANCE WHEN THE OPERATOR TYPED AN "A" OR AN "-" MARK. IF THE PROGRAM WAS DESIGNED TO PERFORM A CERTAIN FUNCTION ON RECEIPT OF THE LETTER "A" IT WOULD ALSO PERFORM IT IF THE PUNCTUATION "-" WAS RECEIVED! TO AVOID THAT HAPPENING, ONE MIGHT INFORM THE HUMAN OPERATOR TO ALWAYS ENTER INFORMATION DURING THAT PART OF THE PROGRAM WITH MACHINE IN THE "LETTERS" MODE, BUT THAT IS NOT THE SAFEST WAY IN WHICH TO DESIGN A PROGRAM.

INSTEAD, ONE WOULD BE BETTER OFF TO ADD A BIT TO THE BAUDOT CODE WHEN IT WAS MANIPULATED IN THE COMPUTER THAT WOULD SERVE TO DIFFERENTIAT BETWEEN "LETTERS" AND "FIGURES." FOR INSTANCE, THE CODE 000011 COULD BE USED TO INDICATE THE LETTER "A" AND 100011 TO INDICATE THE PUNCTUATION "-" MARK. IN ORDER TO INSTITUTE THIS METHOD, ONE WOULD HAVE TO HAVE A PROGRAM THAT KEPT TRACK OF WHICH MODE THE TELETYPE MACHINE WAS OPERATING IN WHENEVER IT WAS RECEIVING DATA FROM THE MACHINE, BY "REMEMBERING" THE LAST "LETTERS" OR "FIGURES" CODE RECEIVED. FURTHERMORE, IN ORDER TO ENSURE THAT THE MODE WAS PROPERLY RECEIVED (SUCH AS WHEN THE PROGRAM WAS FIRST STARTED OR POWER TURNED ON THE TELETYPE MACHINE), IT WOULD BE WISE TO HAVE THE COMPUTER OUTPUT A COMMAND THAT WOULD PLACE THE MACHINE IN A KNOWN STATE SUCH AS WOULD BE ACCOMPLISHED BY OUTPUTTING A "LETTERS" OR "FIGURES" CODE AT THE START OF SUCH OPERATIONS. THEN, FOR STORAGE AND MANIPULATION IN THE COMPUTER, THE INPUT ROUTINE COULD SET A SIXTH BIT TO A "1" CONDITION WHENEVER A CODE WAS RECEIVED WHILE THE MACHINE WAS IN, SAY, THE "FIGURES" MODE, AND LEAVE THE SIXTH BIT AS A "0" WHEN CODES WERE RECEIVED IN THE "LETTERS" MODE. THE SIX BIT CODES COULD THEN BE MANIPULATED AND STORED BY THE PROGRAM IN MUCH THE SAME MANNER AS ONE MIGHT PROCESS "ASCII" CODES WITH THE ABILITY TO IMMEDIATELY RECOGNIZE THE CLOSE TO 60 DIFFERENT CHARACTERS. WHEN IT WAS DESIRED TO OUTPUT INFORMATION, THE SIXTH BIT WOULD BE USED TO INDICATE WHETHER IT WAS NECESSARY TO FIRST OUTPUT A "FIGURES" OR "LETTERS" CODE TO SET THE MACHINE IN THE PROPER MODE. (IT WOULD NOT BE NECESSARY TO OUTPUT A "FIGURES" OR "LETTERS" MODE COMMAND BEFORE EVERY CHARACTER WAS SENT BECAUSE ONE COULD USE AN ALGORITHM THAT WOULD ONLY SEND A "MODE" COMMAND WHEN THE "SIXTH BIT" WAS NOTED TO HAVE CHANGED FROM THAT PRESENT WHEN THE PREVIOUS CHARACTER WAS TRANSMITTED).

TWO SAMPLE ROUTINES FOR PERFORMING SUCH A FUNCTION, ONE FOR INPUTTING DATA FROM A BAUDOT MACHINE, AND ONE FOR OUTPUTTING DATA TO SUCH A MACHINE, WILL BE ILLUSTRATED BELOW.

```
                MNEMONIC                    COMMENTS
                --------                    --------

BAUDIN, LAI 037         /LOAD "LETTERS" CODE INTO ACCUMULATOR
        CAL OUTPUT      /CALL ROUTINE TO SEND BAUDOT CHAR
        CAL LETCOD      /INITIALIZE REG "B" TO "LETTERS"
INBAUD, CAL INPUT       /NOW ACCEPT BAUDOT CHARS FM MACHINE
```

```
                MNEMONIC                    COMMENTS
        ----------------------      ----------------------

                CPI 033         /SEE IF "FIGURES" CODE
                CTZ FIGCOD      /GO SET UP "1" AS SIXTH POSITION BIT
                CPI 037         /SEE IF "LETTERS" CODE
                CTZ LETCOD      /GO SET UP "0" AS SIXTH POSITION BIT
                ADB             /ADD IN STATUS OF SIXTH BIT POSITION
        STORBD, CAL MANIP       /USER SUBRTN TO PROCESS DATA
                JMP INBAUD      /GET NEXT CHAR IN SEQUENCE IF APPLICABLE
        FIGCOD, LBI 040         /SET SIXTH BIT IN "B" = 1
                RET             /RETURN TO MAIN ROUTINE
        LETCOD, LBI 000         /SET SIXTH BIT IN "B" = 0
                RET             /RETURN TO MAIN ROUTINE
```

THE READER SHOULD NOTE THAT THERE ARE ACTUALLY TWO ENTRY POINTS TO THE ROUTINE JUST PRESENTED. THE SUBROUTINE "BAUDIN" SHOULD BE CALLED TO INITIALIZE THE CONDITION OF THE BAUDOT MACHINE WHENEVER THE PROGRAM IS FIRST STARTED OR AT OTHER TIMES WHEN THE "MODE" OF THE MACHINE IS NOT CERTAIN. ONCE THE MACHINE AND ROUTINE HAS BEEN "INITIALIZED" THEN THE PROGRAM MAY BE CALLED AT "INBAUD" AS LONG AS SOME OTHER ROUTINE DOES NOT INTERFERE WITH THE STATUS OF REGISTER "B." THE READER WHO IS INTERESTED IN "LOGIC" MIGHT NOTE THAT REGISTER "B" IN THE ABOVE PROGRAM ACTS AS A "FLIP-FLOP" TO REMEMBER THE "MODE" IN WHICH THE TELETYPE IS OPERATING.

THE ROUTINE SHOWN NEXT ALSO HAS TWO ENTRY POINTS. THE FIRST TERMED "BAUDOT" IS USED WHEN THE FIRST CHARACTER OF A STRING OF CHARACTERS IS TO BE OUTPUTTED IN ORDER TO "INITIALIZE" THE BAUDOT MACHINE AND SET UP REGISTER "C." THE ENTRY POINT "OTBAUD" MAY THEN BE USED UNTIL THE "MODE" MEMORY REGISTER ("C") IS INTERFERED WITH BY ANY OTHER EXTERNAL ROUTINE. NOTE TOO, THAT THE ROUTINE BELOW EXPECTS THE CHARACTER TO BE OUTPUTTED TO BE RESIDING IN REGISTER "B" WHEN THE SUBROUTINE IS CALLED!

```
                MNEMONIC                    COMMENTS
        ----------------------      ----------------------

        BAUDOT, LAI 037         /LOAD "LETTERS" CODE INTO ACCUMULATOR
                CAL OUTPUT      /CALL ROUTINE TO SEND BAUDOT CHARACTER
                LCI 000         /SET INDICATOR FOR "LETTERS" IN "C"
        OTBAUD, LAB             /MOVE CHAR FM "B" TO ACCUMULATOR
                NDI 040         /SEE IF SIXTH BIT = 1, IF YES = "FIGURES"
                JTZ LTCHAR      /CHARACTER, IF NOT = "LETTERS" CHARACTER
                NDC             /IF "FIG" SEE IF LAST OUT ALSO "FIG"
                JTZ LASLET      /IF 0 HERE THEN LAST WAS A "LETTERS"
        OUTCOD, LAB             /PUT PRESENT CHARACTER IN ACCUMULATOR
                CAL OUTPUT      /SEND THE BAUDOT CHARACTER
                RET             /RETURN TO CALLING ROUTINE
        LASLET, LAI 033         /SINCE LAST WAS "LTR" PUT "FIG" CODE
        LASFIG, CAL OUTPUT      /SEND CODE
                LCB             /SAVE LATEST IN REG "C" FOR COMPARISON
                JMP OUTCOD      /SEND CURRENT CHARACTER
        LTCHAR, LAI 040         /SET MASK & SEE IF LAST WAS "LETTERS"
                NDC             /BY COMPARISON OF SIXTH BIT POSITION
                JTZ OUTCOD      /IF 0 HERE, LAST WAS ALSO "LETTERS"
                LAI 037         /IF NOT, SEND "LETTERS" CODE FIRST
                JMP LASFIG      /BY USING ABOVE RTN TO SEND "LETTERS" CODE
```

IT IS OFTEN DESIRABLE TO HAVE I/O ROUTINES THAT WILL CONVERT BETWEEN

ONE TYPE OF I/O CODE AND ANOTHER, SUCH AS BETWEEN "ASCII" AND "BAUDOT." THIS MAY BE DESIRED FOR A NUMBER OF REASONS - FOR INSTANCE BECAUSE ONE HAS ONE TYPE OF INPUT DEVICE USING ONE CODE AND A DIFFERENT OUTPUT DEVICE USING ANOTHER CODE. OR, ONE MIGHT DESIRE TO USE A PARTICULAR PROGRAM THAT WAS WRITTEN TO USE ONE KIND OF CODE, WITH A MACHINE THAT USED A DIFFERENT KIND OF CODE, WITHOUT HAVING TO MODIFY A LOT OF LOCATIONS IN THE ORIGINAL PROGRAM THAT MIGHT HAVE BEEN TESTING FOR SPECIFIC I/O CODES FROM AN EXTERNAL DEVICE. IN SUCH CASES, THE COMPUTER'S CAPABILITY TO PERFORM CONVERSION FUNCTIONS IS READILY CAPITALIZED UPON BY CONSTRUCTING A "LOOKUP" TABLE AND USING A SUITABLE PROGRAM TO CONVERT FROM ONE CODE TO ANOTHER.

FOR EXAMPLE, SUPPOSE IT WAS DESIRED TO USE A "BAUDOT" MACHINE WITH A PROGRAM THAT WAS DEVELOPED ORIGINALLY TO OPERATE WITH A MACHINE THAT USED "ASCII" CODE. ONE COULD PROCEED TO FIRST CONSTRUCT A "LOOKUP" TABLE SIMILAR IN FORMAT TO THAT SHOWN HERE:

ADDRESS	CONTENTS	COMMENTS
10 000	301	"A" (ASCII)
10 001	003	"A" (BAUDOT)
10 002	302	"B" (ASCII)
10 003	031	"B" (BAUDOT)
. .	.	. .
. .	.	. .
. .	.	. .
. .	.	. .
10 076	240	"SPACE" (ASCII)
10 077	004	"SPACE" (BAUDOT)
10 100	241	"!" (ASCII)
10 101	015	"!" (BAUDOT)
. .	.	. .
. .	.	. .
. .	.	. .
. .	.	. .
10 174	277	"?" (ASCII)
10 175	071	"?" (BAUDOT)
10 176	300	"@" (ASCII)
10 177	000	SUBSTITUTE "NULL" (BAUDOT)

IN CONSTRUCTING THE TABLE, ONE COULD ELECT TO LEAVE OUT OR "IGNORE" CHARACTERS THAT WERE NOT REPRESENTED BY BOTH CODES, OR TO SUBSTITUTE A "SUBSTITUTE" CHARACTER WHEN ONE CODE DOES NOT HAVE AN EQUIVALENT CHARACTER. EITHER METHOD REQUIRES CONSIDERATION WHEN THE SEARCH ROUTINE IS DEVELOPED. THE FORMER METHOD LEAVES THE POSSIBILITY THAT A HUMAN OPERATOR MIGHT TYPE IN A CHARACTER THAT DID NOT EXIST IN THE TABLE AND SO THE PROGRAMMER WOULD HAVE TO BE CAREFUL TO "LIMIT" THE TABLE SEARCH ROUTINE. NOTE THAT IF EVERY POSSIBLE ENTRY EXIST IN THE TABLE, THEN THE TABLE SEARCH ROUTINE WILL BE "SELF LIMITING" IN THAT A MATCH WILL ALWAYS BE FOUND. ON THE OTHER HAND, THE LATTER CHOICE OF USING A SUBSTITUTE CHARACTER REQUIRES THAT THE TABLE BE ORGANIZED SO THAT THE "PREFERRED" CHARACTER FOR CASES OF MULTIPLE SUBSTITUTION WILL BE THE ONE FOUND "FIRST" BY THE TABLE LOOKUP ROUTINE. FOR INSTANCE, THERE ARE SEVERAL CHARACTERS BESIDES THE "@" MARK, SUCH AS "]" AND "[" WHICH COULD BE INCLUDED IN THE ABOVE TABLE WHICH ARE REPRESENTED BY ASCII CODES BUT NOT BAUDOT CODES. IF ONE DECIDED TO INCLUDE THEM IN THE TABLE, BUT HAVE "NULL" CHARACTERS AS THEIR CONVERSION EQUIVALENT, ONE CAN SEE THAT A PROBLEM ARISES WHEN ONE USES THE SAME TABLE TO CONVERT FROM BAUDOT TO ASCII AS NOW THERE

ARE SEVERAL PLACES IN THE TABLE THAT HAVE THE "NULL" CODE. AS WILL BE CLEAR SHORTLY, THE ROUTINE THAT CONVERTS FROM BAUDOT TO ASCII, WILL ALWAYS REPRESENT A "NULL" CHARACTER IN BAUDOT AS A "@" SYMBOL IN ASCII BECAUSE THE BAUDOT ROUTINE "SEARCHES" THE TABLE FROM HIGHEST ADDRESS TO LOWEST AND WILL FIND THE "NULL" TO "@" ENTRY FIRST. NATURALLY, THE TABLE COULD BE RE-ORGANIZED SO THAT SOME OTHER "NULL" CONVERSION ENTRY WAS LOCATED FIRST. OR, A DIFFERENT TYPE OF LOOKUP ROUTINE THAN THE ONE TO BE PRESENTED CAN BE DEVELOPED. THESE FACTORS ARE SIMPLY BEING POINTED OUT TO INCREASE THE READER'S AWARENESS AS TO THE TYPES OF FACTORS THAT MUST BE CONSIDERED WHEN PERFORMING SUCH OPERATIONS.

A ROUTINE THAT WILL USE THE "LOOKUP" TABLE TO CONVERT "ASCII" CHARACTERS TO "BAUDOT" IS ILLUSTRATED NEXT. THIS PROGRAM, AND THE "BAUDOT" ROUTINE DISCUSSED EARLIER COULD BE USED TO OUTPUT CHARACTERS FROM A PROGRAM THAT WAS ACTUALLY DOING INTERNAL PROCESSING WITH ASCII CODES.

```
          MNEMONIC                        COMMENTS
          --------                        --------

ASBAUD,   LHI 010         /SET PAGE ADDR PNTR TO LOC OF TABLE
          LLI 000         /SET LOW ADDR PNTR TO "TOP" OF TABLE
FASCII,   CPM             /COMPARE (ASCII) CODE IN ACC TO CONTENTS
          JTZ FNDBDO      /OF TABLE, IF MATCH, DO CONVERSION
          INL             /OTHERWISE ADVANCE LOW ADDR POINTER
          INL             /TO NEXT "ASCII" CODE LOCATION IN TABLE
          JMP FASCII      /AND KEEP LOOKING FOR A MATCH
FNDBDO,   INL             /WHEN HAVE ASCII MATCH, ADV PNTR 1 LOC
          LAM             /AND FETCH BAUDOT EQUIVALENT INTO ACC
          RET             /EXIT LOOKUP ROUTINE
```

THE ABOVE ROUTINE ASSUMES THAT THE CODE (IN ASCII) FOR A CHARACTER THAT EXISTS IN THE TABLE IS IN THE ACCUMULATOR WHEN THE ROUTINE IS ENTERED. NOTE THAT THE ROUTINE DOES NOT TEST FOR THE "END" OF THE TABLE BECAUSE OF THAT ASSUMPTION. IF FOR ANY REASON IT MIGHT BE POSSIBLE FOR A CODE TO BE IN THE ACCUMULATOR THAT WAS NOT IN THE TABLE, THEN IT WOULD BE NECESSARY TO ADD AN "END OF TABLE" TEST EACH TIME THE TABLE POINTER WAS ADVANCED AND TO TAKE APPROPRIATE ACTION IF "NO MATCH" WAS FOUND IN THE TABLE.

THE NEXT ROUTINE DOES ESSENTIALLY THE REVERSE PROCESS, USING THE SAME TABLE, TO CONVERT BAUDOT CODES TO ASCII CODES. IT COULD BE USED ALONG WITH THE PREVIOUSLY DESCRIBED "BAUDIN" ROUTINE TO ACCEPT CHARACTERS FROM A BAUDOT MACHINE AND CONVERT THEM FOR USE IN A PROGRAM THAT UTILIZED ASCII CODES. AS IN THE ABOVE ROUTINE, THE PROGRAM ASSUMES THAT A VALID BAUDOT CODE IS IN THE ACCUMULATOR WHEN THE ROUTINE IS CALLED. NOTE THAT THE ROUTINE STARTS SEARCHING THE TABLE IN THE OPPOSITE DIRECTION THAN THE ROUTINE PRESENTED ABOVE.

```
          MNEMONIC                        COMMENTS
          --------                        --------

BAUDAS,   LHI 010         /SET PAGE ADDR PNTR TO LOC OF TABLE
          LLI 177         /SET LOW ADDR PNTR TO "BOTTOM" OF TABLE
FBAUDO,   CPM             /COMPARE (BAUDOT) CODE IN ACC TO CONTENTS
          JTZ FNDASC      /OF TABLE, IF MATCH, DO CONVERSION
          DCL             /OTHERWISE DECREMENT LOW ADDR POINTER
          DCL             /TO NEXT "BAUDOT" CODE LOCATION IN TABLE
          JMP FBAUDO      /AND KEEP LOOKING FOR A MATCH
```

MNEMONIC		COMMENTS
FNDASC,	DCL	/WHEN HAVE BAUDOT MATCH, DECR PNTR 1 LOC
	LAM	/AND FETCH ASCII EQUIVALENT INTO ACC
	RET	/EXIT LOOKUP ROUTINE

NATURALLY, THE TECHNIQUES ILLUSTRATED TO CONVERT BETWEEN "ASCII" AND "BAUDOT" CODES MAY BE APPLIED TO MANY OTHER TYPES OF CODES. INDEED, THE SMALL COMPUTER MAKES AN IDEAL DEVICE FOR "COUPLING" BETWEEN A VARIETY OF I/O DEVICES, PARTICULARLY IN COMMUNICATION APPLICATIONS, THUS ENABLING MACHINES OF DIFFERENT CHARACTERISTICS AND USING DIFFERENT CODES TO COMMUNICATE WITH ONE ANOTHER.

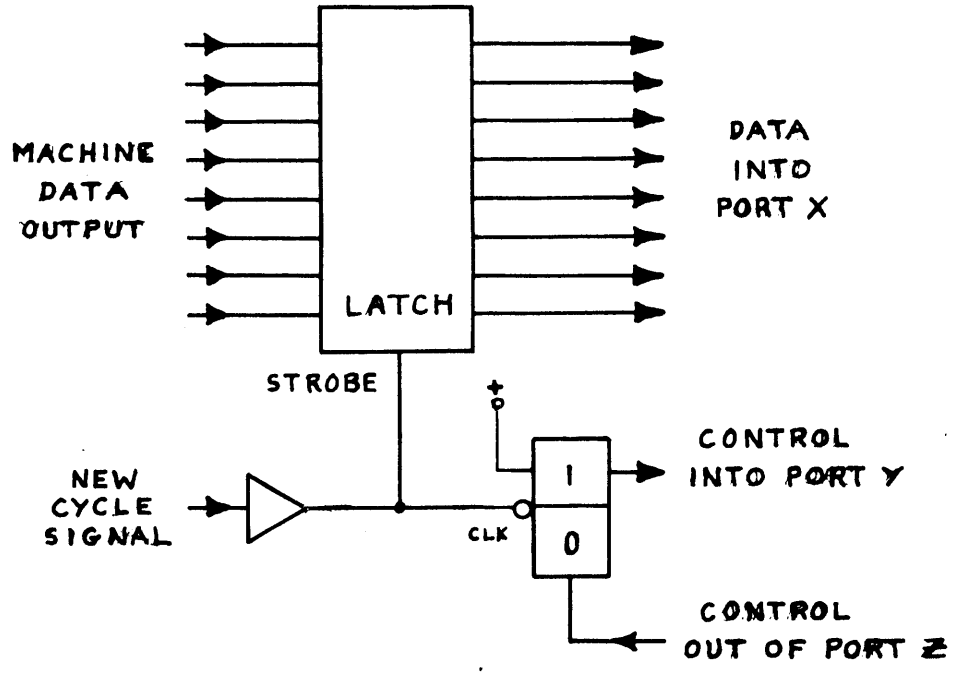
A CONCEPT THAT WILL BE DISCUSSED MORE FULLY IN THE NEXT CHAPTER WILL BE BRIEFLY MENTIONED AT THIS TIME TO POINT OUT AN IMPORTANT CONCEPT WHEN DEALING WITH I/O DEVICES CONNECTED TO THE COMPUTER. AS THE READER UNDOUBTABLY KNOWS, MANY MACHINES THAT MIGHT BE CONNECTED TO A COMPUTER ARE MUCH SLOWER IN OPERATION, IN FACT OFTEN TIMES ORDERS OF MAGNITUDE SLOWER, THAN THE BASIC OPERATING CYCLE OF A COMPUTER. FOR INSTANCE, AN 8008 SYSTEM REQUIRES BUT A MERE 32 MILLIONTHS OF A SECOND IN A TYPICAL SYSTEM TO EXECUTE AN "INPUT" INSTRUCTION. THAT IS, IN THAT SHORT AMOUNT OF TIME IT CAN "ACCESS" AN INPUT PORT AND BRING IN 8 PARALLEL BITS OF INFORMATION INTO THE ACCUMULATOR OF THE CPU.

THE EXTREME SPEED OF THE COMPUTER CAN IN FACT CAUSE PROBLEMS WHEN PERFORMING I/O OPERATIONS IF STEPS ARE NOT TAKEN TO "CONTROL" THE SITUATION. ASSUME FOR EXAMPLE, THAT A PERSON DESIRED TO CONNECT AN ELECTRONIC KEYBOARD UNIT, SIMILAR TO A TYPEWRITER, THAT WOULD PRESENT THE ASCII CODE FOR THE KEY BEING DEPRESSED IN PARALLEL ON THE LINES OF AN INPUT PORT. IF THE PERSON JUST CONNECTED THE KEYBOARD OUTPUT LINES TO THE INPUT LINES OF AN INPUT PORT, AND WANTED TO DEVELOP A PROGRAM THAT WOULD ACCEPT INFORMATION FROM THE KEYBOARD, THERE WOULD BE A NUMBER OF RATHER TOUGH PROBLEMS, AND THEY WOULD BE RELATED TO THE SPEED AT WHICH THE COMPUTER CAN OPERATE RELATIVE TO THE SPEED AT WHICH A HUMAN CAN DEPRESS THE KEYS ON A KEYBOARD.

SUPPOSE THAT THE KEYBOARD WAS DIRECTLY CONNECTED TO AN INPUT PORT AND A PROGRAMMER TRIED TO DEVELOP A ROUTINE THAT WOULD SIMPLY READ THE CODE BEING SENT BY THE KEYBOARD, STORE THE CHARACTER IN MEMORY, AND GO ON TO READ THE NEXT CHARACTER. IN THE FIRST PLACE, HOW WOULD THE PROGRAM BE ABLE TO EVEN TELL IF A KEY HAD BEEN DEPRESSED? TRUE, ONE COULD ASSUME THAT IF NO KEYS WERE DEPRESSED, THAT THE CODE BEING RECEIVED WOULD BE ALL ZEROS, AND A PROGRAM COULD CHECK FOR THAT CONDITION. BUT, EVEN IF THAT WAS DONE, THE PROGRAMMER WOULD SOON HAVE ANOTHER PROBLEM. WHEN A KEY WAS ACTUALLY DEPRESSED AND A "NON-ZERO" CONDITION RECEIVED, A SHORT PROGRAM TO PLACE THE CHARACTER IN MEMORY AND ADVANCE THE MEMORY POINTER WOULD BE ACCOMPLISHED IN THE ORDER OF A HUNDRED-MILLIONTHS OF A SECOND. THE POOR HUMAN DEPRESSING THE KEY WOULDN'T HAVE A CHANCE OF GETTING A FINGER OFF THE DEPRESSED KEY IN THAT AMOUNT OF TIME, AND IN FACT IT WOULD TAKE ON THE ORDER OF SEVERAL TENTHS OF A SECOND FOR A PERSON TO REMOVE A FINGER FROM A KEY. IN THAT AMOUNT OF TIME, THE SIMPLE INPUT ROUTINE COULD HAVE "READ" THAT SAME CHARACTER AND PACKED IT INTO MEMORY LOCATIONS A FEW HUNDRED TIMES! NOT EXACTLY THE DESIRED RESULT. WHAT NOW? WELL, ONE COULD DEVELOP THE INPUT ALGORITHM SO THAT, ONCE A "NON-ZERO" CODE WAS RECEIVED, ONE WOULD NOT ACCEPT ANOTHER CHARACTER UNTIL A "ZERO" CODE WAS OBSERVED. THAT MIGHT IMPROVE THINGS SOMEWHAT, BUT IT WOULD PRECLUDE ACTUALLY BEING ABLE TO RECEIVE A "ZERO" CODE (THAT MIGHT REPRESENT A VALID CONDITION) AND, BECAUSE OF TECHNICAL CONSIDERATIONS (SUCH AS "CONTACT BOUNCE" ON THE MECHANICAL SWITCHES OF THE

KEYBOARD) IT WOULD NOT BE A VERY RELIABLE METHOD TO UTILIZE.

INSTEAD, IT WOULD BE FAR BETTER TO PLACE AN "INTERFACE" BETWEEN THE KEYBOARD AND THE COMPUTER INPUT PORT THAT WOULD ACCOMPLISH THE FOLLOWING OBJECTIVES. WHENEVER A KEY ON THE KEYBOARD WAS DEPRESSED, THE INTERFACE WOULD "LATCH" (HOLD) THE CODE REPRESENTED BY THE KEY IN AN ELECTRONIC "BUFFER" THAT WAS CONNECTED TO THE LINES OF AN INPUT PORT. THE "BUFFER" WOULD THUS HOLD "DATA" FROM THE KEYBOARD. NEXT, WHEN THE KEY THAT HAD BEEN DEPRESSED WAS RELEASED, THE "INTERFACE" WOULD PRESENT A SIGNAL TO AN INPUT LINE OF ANOTHER INPUT PORT - TERMED A "CONTROL" PORT. FINALLY, THE INTERFACE WOULD HAVE A LINE COMING FROM AN OUTPUT PORT OF THE COMPUTER, THAT WOULD ALLOW THE COMPUTER TO SIGNAL TO THE INTERFACE THAT IT HAD TAKEN APPROPRIATE ACTION. A DIAGRAM OF AN ELECTRONIC INTERFACE WITH THE CHARACTERISTICS DESCRIBED IS SHOWN IN THE NEXT ILLUSTRATION.



WITH SUCH AN INTERFACE, ONE COULD DEVELOP A MUCH MORE RELIABLE SYSTEM USING AN INPUT PROGRAM THAT WOULD PERFORM IN THE FOLLOWING MANNER:

```
          MNEMONIC                    COMMENTS
          --------                    --------

MACHIN,   INP Y          /CHECK STATUS OF "CONTROL" FM MACHINE
          JFS MACHIN     /IF DATA NOT READY - WAIT BY LOOPING
          INP X          /DATA READY NOW SO FETCH "DATA"
          LBA            /SAVE "DATA" IN REGISTER "B"
          LAI 001        /PREPARE TO PULSE LINE ON PORT "Z"
          OUT Z          /SEND LOGIC "1" ON PORT Z CONTROL LINE
          XRA            /CLEAR ACCUMULATOR
          OUT Z          /SEND LOGIC "0" ON PORT Z CONTROL LINE
          LAB            /RESTORE "DATA" TO ACCUMULATOR
          RET            /EXIT RTN WITH "DATA" IN ACCUMULATOR
```

THE ABOVE ROUTINE ASSUMED THAT THE "CONTROL" LINE FROM THE INTERFACE

CAME INTO THE MOST SIGNIFICANT BIT OF THE ACCUMULATOR AND THAT THE CONTROL LINE GOING TO THE INTERFACE ORIGINATED FROM THE LEAST SIGNIFICANT BIT IN THE ACCUMULATOR. FURTHERMORE, WHILE THE ABOVE ROUTINE "WAITED" FOR NEW DATA TO ARRIVE FROM THE EXTERNAL DEVICE BY MONITORING THE INPUT CONTROL PORT CONTINUOUSLY, THE "JFS MACHIN" INSTRUCTION COULD HAVE BEEN REPLACED BY A DIRECTIVE TO HAVE THE COMPUTER PERFORM SOME OTHER FUNCTION(S) BEFORE TESTING INPUT PORT "Y" AGAIN INSTEAD OF WASTING TIME DOING NOTHING!

A SIMILAR TYPE OF INTERFACE, AND SIMILAR PROGRAMMING TECHNIQUES CAN BE APPLIED TO A WIDE VARIETY OF DEVICES THAT MIGHT BE CONNECTED TO THE COMPUTER. WHILE THE EXAMPLE SHOWED BUT ONE LINE BEING USED ON EACH CONTROL PORT, ONE SHOULD NOTE THAT WITH EIGHT LINES AVAILABLE ON ONE PORT, ONE CAN USE JUST A FEW "CONTROL" PORTS IN A SYSTEM TO MONITOR AND CONTROL A LARGE GROUP OF EXTERNAL INSTRUMENTS BY USING THE AVAILABLE BIT POSITIONS.

TESTING FOR ERRORS DURING I/O OPERATIONS

IT IS OFTEN DESIRABLE TO TRANSMIT DATA TO AN EXTERNAL DEVICE THAT WILL STORE THE DATA IN SOME SORT OF PERMANENT FORM, SUCH AS ON PAPER TAPE OR MAGNETIC TAPE. THEN, AT SOME LATER TIME, READ THE DATA BACK INTO THE COMPUTER. DURING SUCH A PROCESS IT IS POSSIBLE FOR ERRORS TO OCCUR. THAT IS, BITS OF INFORMATION WITHIN A "WORD" MAY BE ALTERED BECAUSE OF "NOISE" OR RANDOM ERRORS OCCURING IN THE I/O SYSTEM. WHILE SUCH ERRORS ARE LIKELY TO OCCUR AT A VERY LOW RATE IN A WELL DESIGNED, PROPERLY OPERATING I/O SYSTEM, IT IS OFTEN DESIRABLE TO UTILIZE TECHNIQUES THAT WILL AT LEAST INDICATE WHEN AN ERROR HAS OCCURRED. THERE ARE A VARIETY OF ERROR CHECKING TECHNIQUES AVAILABLE, SOME SO SOPHISTICATED THAT THEY CAN OFTEN "CORRECT" CERTAIN TYPES OF ERRORS THAT OCCUR DURING I/O OPERATIONS. TWO TECHNIQUES WILL BE DISCUSSED HERE. WHILE NEITHER ONE OF THEM HAS "ERROR CORRECTING" CAPABILITY, THEY ARE CAPABLE OF DETECTING THE MOST COMMON TYPE OF I/O ERROR WHICH IS FOR A BIT IN A WORD CHANGING STATE.

THE FIRST METHOD TO BE DISCUSSED CONCERNS THE USE OF USING "PARITY" TECHNIQUES TO DETECT TRANSMISSION ERRORS. THE TECHNIQUE CONSISTS OF EXAMINING A GROUP OF BITS FOR THE NUMBER OF BITS THAT ARE IN THE "1" CONDITION WHEN IT IS BEING READIED FOR "TRANSMISSION" AND THEN SETTING A BIT SET ASIDE FOR THE PURPOSE TO THE STATE THAT WILL MAKE THE TOTAL NUMBER OF BITS THAT ARE IN THE "1" CONDITION EITHER AN "ODD" OR "EVEN" COUNT (FOR THE ENTIRE GROUP). FOR INSTANCE, IT WAS MENTIONED EARLIER THAT THE "ASCII" CODE REQUIRED 7 BITS TO REPRESENT ALL THE POSSIBLE 128 CHARACTERS DEFINED BY THE CODE, BUT THAT MANY SYSTEMS EMPLOYED AN 8'TH BIT FOR "PARITY" PURPOSES. THUS, THE "ASCII" CODE IS IDEAL FOR USE IN TYPICAL 8008 SYSTEMS BECAUSE THERE ARE EXACTLY 8 BITS TO A COMPUTER WORD.

FURTHERMORE, THE 8008 CPU HAS AS PART OF IT'S INSTRUCTION SET, SPECIFIC INSTRUCTIONS TO FACILITATE THE USE OF PARITY TECHNIQUES. REMEMBER THE "PARITY" FLAG THAT WAS DISCUSSED IN THE CHAPTER ON THE 8008 INSTRUCTION SET AND THE VARIOUS CONDITIONAL BRANCHING INSTRUCTIONS THAT USE THE STATUS OF THE PARITY FLAG?

WHEN THE CODES FOR THE "ASCII" SUBSET WERE DESCRIBED EARLIER, IT WAS MENTIONED THAT THE EIGHTH BIT POSITION (MOST SIGNIFICANT BIT) IN THE LISTING WAS ARBITRARILY SET TO THE "1" CONDITION AS THE "ASCII" CODE DID NOT USE THAT BIT. HOWEVER, THAT BIT POSITION MAY BE USED TO SPECIFY THE DESIRED "PARITY" IN A SYSTEM WHERE PARITY CHECKING IS TO BE EMPLOYED.

FOR INSTANCE, IF ONE WANTED TO ESTABLISH AN EVEN PARITY SYSTEM, ONE WOULD PROCEED IN THE FOLLOWING MANNER.

EXAMINE THE SEVEN BITS MAKING UP THE CODE FOR THE CHARACTER TO BE TRANSMITTED (ASSUMING "ASCII" CODE FOR THIS EXAMPLE). IF THE NUMBER OF BITS IN THE CHARACTER THAT ARE A LOGIC "1" ARE "EVEN," THAT IS THERE ARE 0, 2, 4 OR 6 BITS IN THE "1" STATE, SET THE 8'TH BIT TO A "0." IF THE NUMBER OF BITS ARE "ODD," THAT IS THERE ARE 1, 3, 5 OR 7 BITS IN THE "1" STATE, SET THE 8'TH BIT TO A "1" CONDITION SO THAT THE TOTAL NUMBER OF BITS IN THE ENTIRE GROUP BECOMES AN EVEN NUMBER! SOME EXAMPLES ARE ILLUSTRATED BELOW.

	ORIGINAL 7 BIT ASCII CODE	8 BIT "EVEN" PARITY CODE
(A)	1 0 0 0 0 0 1	0 1 0 0 0 0 0 1
(B)	1 0 0 0 0 1 0	0 1 0 0 0 0 1 0
(C)	1 0 0 0 0 1 1	1 1 0 0 0 0 1 1
(D)	1 0 0 0 1 0 0	0 1 0 0 0 1 0 0
(E)	1 0 0 0 1 0 1	1 1 0 0 0 1 0 1
(0)	0 1 1 0 0 0 0	0 0 1 1 0 0 0 0
(1)	0 1 1 0 0 0 1	1 0 1 1 0 0 0 1

ONE COULD ALSO ELECT TO USE AN "ODD" PARITY SYSTEM BY ESSENTIALLY REVERSING THE SCHEME SO THAT THE 8'TH BIT IS ALWAYS SET TO MAKE THE TOTAL NUMBER OF BITS IN A GROUP THAT ARE IN THE "1" STATE BE AN "ODD" NUMBER. "ASCII" CODE USING AN 8'TH BIT TO PRODUCE AN "ODD PARITY" SYSTEM IS ILLUSTRATED BELOW FOR SEVERAL CHARACTERS.

	ORIGINAL 7 BIT ASCII CODE	8 BIT "ODD" PARITY CODE
(A)	1 0 0 0 0 0 1	1 1 0 0 0 0 0 1
(B)	1 0 0 0 0 1 0	1 1 0 0 0 0 1 0
(C)	1 0 0 0 0 1 1	0 1 0 0 0 0 1 1
(D)	1 0 0 0 1 0 0	1 1 0 0 0 1 0 0
(E)	1 0 0 0 1 0 1	0 1 0 0 0 1 0 1
(0)	0 1 1 0 0 0 0	1 0 1 1 0 0 0 0
(1)	0 1 1 0 0 0 1	0 0 1 1 0 0 0 1

ONCE ONE HAS SELECTED WHICH PARITY (ODD OR EVEN) TO USE WITH A SYSTEM ONE SIMPLY SENDS THE DATA IN THE DESIRED MODE TO THE I/O DEVICE. THEN, WHEN THE DATA IS LATER READ INTO THE COMPUTER, A CHECK IS MADE ON EACH "WORD" OF DATA RECEIVED TO DETERMINE IF THE PARITY IS CORRECT. IF IT IS NOT, THEN AN ERROR HAS OCCURRED. SAMPLE ROUTINES TO GENERATE "EVEN" PARITY WORDS FOR AN OUTPUT ROUTINE, AND FOR CHECKING FOR "EVEN" PARITY IN AN INPUT ROUTINE ARE SHOWN NEXT.

```
          MNEMONIC                    COMMENTS
          --------                    --------

SEVENP,   NDA          /ASSUME 7 BIT ASCII CODE IN ACC, 8'TH BIT
          JTP GOUT     /INIT 0, IF PARITY EVEN AS IS, SEND DATA
          XRI 200      /OTHERWISE SET MSB = 1 TO GET EVEN PARITY
GOUT,     CAL OUTPUT   /USER ROUTINE TO TRANSMIT DATA TO I/O
          RET          /EXIT EVEN PARITY GENERATOR ROUTINE
```

MNEMONIC		COMMENTS
REVENP,	NDA	/ASSUME DATA FM I/O DEVICE IN ACCUMULATOR
	RTP	/SET FLAGS, IF EVEN PARITY, ALL O.K.
	JMP PERROR	/IF NOT EVEN PARITY DO USER ERROR ROUTINE

SIMILAR ROUTINES ARE EASILY DEVELOPED FOR UTILIZING "ODD" PARITY. THE PROGRAMMER SHOULD NOTE THAT "PARITY" TECHNIQUES CAN BE USED WITH VIRTUALLY ANY CODING TECHNIQUE AS LONG AS ONE BIT IS SET ASIDE FOR THE PARITY INDICATOR. FOR INSTANCE, ONE COULD EASILY ADAPT PARITY TECHNIQUES FOR THE BAUDOT CODE DISCUSSED EARLIER PROVIDED THAT THE I/O DEVICE COULD HANDLE THE EXTRA BIT. THAT MIGHT NOT BE POSSIBLE WITH A BAUDOT TELETYPE MACHINE BUT IT MIGHT BE APPLICABLE, SAY, IF BAUDOT CODE WAS BEING WRITTEN ON A MAGNETIC TAPE UNIT WHERE EXTRA BITS COULD BE ADDED TO THE CODE AND PROCESSED BY THE I/O UNIT.

THE READER SHOULD ALSO BE AWARE OF THE FACT THAT THE USE OF PARITY CHECKING TECHNIQUES IS NOT INFALLIBLE. IT DOES DETECT ERRORS THAT RESULT IN AN ODD NUMBER OF BITS CHANGING STATE WITHIN A GROUP, BUT NOT IF AN EVEN NUMBER OF STATE CHANGES OCCUR. IT IS THUS MOST USEFUL IN A SYSTEM WHERE THE EXPECTED PROBABILITY OF MORE THAN ONE ERROR OCCURRING IN A GROUP OF EIGHT BITS IS EXTREMELY LOW. THE PROGRAMMER MIGHT ALSO WANT TO CONSIDER, WHEN USING A "PARITY" CHECKING SCHEME, THE POSSIBILITY OF TRANSMITTING EACH GROUP OF BITS TWICE. THEN, WHEN DATA IS READ BACK FROM THE I/O DEVICE, AN ALGORITHM THAT WILL SKIP THE SECOND GROUP IF THE GROUP IS RECEIVED CORRECTLY THE FIRST TIME, OR READ THE SECOND GROUP IF AN ERROR WAS DETECTED IN THE FIRST GROUP, CAN BE UTILIZED. SUCH A FORMAT, WHILE REQUIRING A LONGER TRANSMIT AND RECEIVE TIME, CAN RESULT IN HIGHLY RELIABLE I/O DATA HANDLING OPERATIONS.

ANOTHER ERROR CHECKING METHOD THAT IS OFTEN USED WHEN PASSING DATA TO AND FROM I/O DEVICES IS TERMED THE "CHECK-SUM" TECHNIQUE. THE METHOD IS QUITE SIMPLE IN APPLICATION YET REMARKABLY POWERFUL IN DETECTING ERRORS. THE TECHNIQUE CONSISTS OF SIMPLY MAINTAINING A ONE REGISTER SUM OF ALL THE DATA TRANSMITTED WITHIN A "BLOCK." THAT IS, AS EACH WORD IS SENT OUT, IT IS SUMMED WITH A REGISTER THAT CONTAINS THE SUM OF ALL PREVIOUS DATA WORDS TRANSMITTED IN THE BLOCK. (OVER-FLOWS IN THE SUMMING REGISTER ARE IGNORED). AT THE END OF A BLOCK OF DATA, THE TWO'S COMPLEMENT OF THE SUM THAT HAS BEEN COMPILED IS SENT AS THE FINAL PIECE OF DATA IN THE BLOCK.

WHEN THE BLOCK OF DATA IS READ BACK INTO THE COMPUTER A SIMILAR SUM IS FORMED AS EACH DATA WORD IS RECEIVED. THEN, WHEN THE LAST PIECE OF DATA IS RECEIVED, WHICH IS THE TWO'S COMPLEMENT OF THE "CHECK-SUM," THAT VALUE IS ADDED TO THE SUM OBTAINED FROM ALL THE PREVIOUS DATA WORDS IN THE BLOCK. THE RESULT, IF NO TRANSMISSION ERRORS HAVE OCCURRED, WILL BE ZERO - THE RESULT OF ADDING ANY NUMBER TO IT'S TWO'S COMPLEMENT. IF IT IS NOT ZERO, THEN A TRANSMISSION ERROR HAS OCCURED. THE METHOD IS SIMPLE AND QUITE RELIABLE. THE READER CAN READILY DETERMINE, THAT IF ERRORS HAVE OCCURRED, IT WILL AFFECT THE VALUE OF THE SUM AS IT IS FORMED, AND THUS LIKELY RESULT IN A NON-ZERO VALUE AS A FINAL RESULT WHEN THE CHECK-SUM AND IT'S TWO'S COMPLEMENT ARE ADDED. (NOTE: IT IS THEORETICALLY POSSIBLE FOR JUST THE RIGHT NUMBER OF ERRORS TO OCCUR WHEN READING A BLOCK OF DATA TO RESULT IN A "ZERO" CONDITION BUT IT IS QUITE SMALL - HARDLY ENOUGH TO LOSE SLEEP OVER)!

A ROUTINE FOR GENERATING A CHECK-SUM AND PLACING THE TWO'S COMPLEMENT OF THAT VALUE AS THE LAST WORD SENT IN A BLOCK OF DATA, FOLLOWED BY

MNEMONIC		COMMENTS

A ROUTINE THAT WILL READ BACK A BLOCK OF DATA USING A CHECK-SUM TECHNIQUE AND TEST TO SEE IF ANY ERRORS OCCURED IS SHOWN BELOW.

```
SCKSUM, LHI XXX     /SET PAGE ADDR WHERE BLOCK OF DATA STORED
        LLI YYY     /SET LOC ON PAGE FOR START OF DATA BLOCK
        LEI ZZZ     /SET # WORDS IN BLOCK COUNTER
        LDI 000     /SET CHECK-SUM REGISTER TO 0 AT START
NXCKSM, LAM         /FETCH DATA WORD FROM MEMORY
        ADD         /ADD PRESENT DATA TO CHECK-SUM VALUE
        LDA         /SAVE NEW CHECK-SUM VALUE
        LAM         /RESTORE ORIG DATA WORD FROM MEMORY
        CAL OUTPUT  /OUTPUT THE DATA WORD TO I/O DEVICE
        INL         /ADVANCE MEMORY POINTER
        DCE         /DECREMENT WORD COUNTER
        JFZ NXCKSM  /IF CNTR NOT 0, FETCH NEXT DATA WORD
        LAD         /PUT CHECK-SUM VALUE IN ACCUMULATOR
        XRI 377     /FORM TWO'S COMPLEMENT VALUE
        ADI 001     /IN STANDARD MANNER
        CAL OUTPUT  /SEND 2'S COMPLEMENT OF CK-SUM AS LAST
        RET         /WORD IN BLOCK AND EXIT ROUTINE

RCKSUM, LHI XXX     /SET PAGE ADDR WHERE BLOCK OF DATA GOES
        LLI YYY     /SET STARTING LOC ON PAGE FOR DATA
        LEI ZZZ     /SET # WORDS IN BLOCK COUNTER
        LDI 000     /SET CHECK-SUM REGISTER TO 0 AT START
INCKSM, CAL INPUT   /FETCH DATA FROM I/O DEVICE
        LMA         /STORE DATA WORD IN MEMORY
        ADD         /ADD NEW DATA TO CURRENT CHECK-SUM VALUE
        LDA         /SAVE NEW CHECK-SUM VALUE
        INL         /ADVANCE MEMORY POINTER
        DCE         /DECREMENT WORD COUNTER
        JFZ INCKSM  /GET NEXT DATA WORD IF CNTR NOT 0
        CAL INPUT   /NEXT WORD FROM I/O IS 2'S COMP OF CK-SUM
        ADD         /ADD IT TO CHECK-SUM FORMED BY DATA
        RTZ         /IF RESULT IS 0, O.K., EXIT ROUTINE
        JMP CKSMER  /OTHERWISE GO TO USER ERROR ROUTINE
```

THE ABOVE ROUTINES, AS THE READER WILL NOTE, ASSUME THAT DATA BLOCKS ARE ONE PAGE OR LESS IN LENGTH AND DO NOT CROSS PAGE BOUNDARIES. HOWEVER, BY THIS TIME THE READER SHOULD HAVE LITTLE DIFFICULTY WRITING A CHECK-SUM ROUTINE THAT COULD HANDLE LARGER BLOCKS.

THE NEXT CHAPTER WILL CONTAIN MORE INFORMATION OF INTEREST TO THOSE DEVELOPING PROGRAMS FOR I/O OPERATIONS THAT REQUIRE CONSIDERATION OF "REAL-TIME" PARAMETERS.

REAL-TIME PROGRAMMING AS DISCUSSED IN THIS MANUAL APPLIES TO THE DEVELOPMENT OF PROGRAMS WHOSE PROPER EXECUTION ARE DEPENDENT ON THE LENGTH OF TIME IT TAKES FOR THE COMPUTER TO PERFORM AN OPERATION OR SERIES OF INSTRUCTIONS. THE NEED FOR REAL-TIME PROGRAMMING IS INVARIABLY RELATED TO THE RECEIPT OF INFORMATION FROM DEVICES AT SPECIFIC TIMES OR THE CONTROL OF DEVICES EXTERNAL TO THE COMPUTER WHOSE PROPER OPERATION DEPEND UPON RECEIVING COMMANDS FROM THE COMPUTER AT SPECIFIC TIMES.

THE DISCUSSION OF THE SUBJECT OF REAL-TIME PROGRAMMING HAS BEEN DEFERRED TO THE LATTER PART OF THIS MANUAL AS REAL-TIME PROGRAMMING IS GENERALLY MORE DIFFICULT THAN THE DEVELOPMENT OF PROGRAMS THAT ARE NOT RESTRICTED BY EXECUTION TIMES. THE REASON IS SIMPLY THAT IN ADDITION TO THE "LOGIC" AND "TECHNIQUE" FACTORS THAT THE PROGRAMMER MUST CONSIDER WHEN DEVELOPING ANY PROGRAM, THE PROGRAMMER MUST NOW ADD IN THE FACTOR OF HOW MUCH TIME IT WILL TAKE FOR THE COMPUTER TO EXECUTE VARIOUS INSTRUCTIONS AND INSTRUCTIONAL SEQUENCES. THE PROBLEM IS REALLY ONE OF "COMPLICATION."

HOWEVER, REAL-TIME PROGRAMMING IF OFTEN VITALLY NECESSARY IN CERTAIN APPLICATIONS AND HENCE THE PROGRAMMER MUST BECOME AWARE OF SOME OF THE CRITICAL ASPECTS OF SUCH PROGRAMMING. THE READER SHOULD NOT, HOWEVER, BE OVER-WHELMED BY THE PROSPECTS OF SUCH COMPLICATIONS. FOR, ONCE ONE HAS AN UNDERSTANDING OF STANDARD MACHINE LANGUAGE PROGRAMMING PROCEDURES AND HAS GAINED A LITTLE EXPERIENCE, WHICH ONE SHOULD HAVE OBTAINED BY THE TIME ONE IS DELVING INTO THIS SECTION, ONE SHOULD FIND THE ASPECTS OF REAL-TIME PROGRAMMING SIMPLY "ONE STEP UP" AND AN EJOYABLE CHALLENGE.

AS WITH MANY OTHER ASPECTS OF PROGRAMMING, PROPER PREPARATION SUCH AS CLEARLY DEFINING THE PROBLEM TO BE HANDLED, AND PROCEEDING IN AN ORDERLY FASHION, CAN GREATLY EASE THE OVER-ALL TASK OF DEVELOPING REAL-TIME PROGRAMS.

THE LAST SEVERAL PAGES OF CHAPTER ONE PRESENTED THE TYPICAL EXECUTION TIMES FOR THE VARIOUS CLASSES OF INSTRUCTIONS AVAILABLE. THE TIMES SHOWN ARE THOSE FOR AN 8008 UNIT WHOSE MASTER CLOCK HAS BEEN ADJUSTED TO A NOMINAL FREQUENCY OF 500 KILOHERTZ. WHEN GETTING DOWN TO PRACTICAL APPLICATIONS, ONE MUST REALIZE THAT ANY SYSTEM WILL HAVE SOME FINITE DEVIATION FROM THE NOMINAL FREQUENCY. FOR INSTANCE, IF AN 8008 SYSTEM HAS A CRYSTAL CONTROLLED MASTER CLOCK, THE POSSIBLE VARIATION FROM THE NOMINAL FREQUENCY MIGHT BE IN THE ORDER OF 0.05 TO 0.1 PERCENT. SOME 8008 SYSTEMS MIGHT HAVE RESISTOR-CAPACITOR CONTROLLED MASTER CLOCKS AND THE POSSIBLE VARIATION FROM THE NOMINAL COULD BE CONSIDERABLY WIDER - UP TO 4 OR 5 PERCENT. IN ANY EVENT, WHEN CONTEMPLATING THE DEVELOPMENT OF REAL-TIME PROGRAMS, ONE MUST ALWAYS TAKE INTO ACCOUNT THE POSSIBLE VARIATION FROM NOMINAL OF THE MASTER CLOCK FREQUENCY, AND IN FACT SHOULD PLAN PROGRAMS TO OPERATE UNDER "WORST CASE" VARIATION CONDITIONS. THUS, IF ONE WAS THINKING OF USING AN 8008 SYSTEM TO CONTROL A PROCESS THAT REQUIRED TIMING ACCURACIES OF 0.01 PERCENT, ONE COULD IMMEDIATELY STOP CONSIDERING USING A COMPUTER THAT HAD A MASTER CLOCK ACCURATE TO ONLY 0.05 PERCENT! A SECOND CONSIDERATION ABOUT WHETHER TO USE A COMPUTER TO CONTROL TIME-DEPENDENT EVENTS, INVOLVES HOW CLOSE TOGETHER EVENTS THAT ARE TO BE CONTROLLED NEED TO OCCUR. IT CAN BE OBSERVED BY EXAMINING THE INFORMATION AT THE END OF CHAPTER ONE, THAT ALMOST ALL THE INSTRUCTIONS REQUIRE A MINIMUM OF 20 MICROSECONDS TO BE EXECUTED. THUS, ONE CANNOT PLAN ON USING THE COMPUTER TO CONTROL EVENTS THAT ARE LESS THAN THAT FAR APART IN TIME. IN FACT, BECAUSE I/O INSTRUCTIONS THEMSELVES TAKE 24 AND 32 MICROSECONDS, AND BECAUSE THOSE INSTRUCTIONS WOULD INVARIABLY BE REQUIRED TO DEAL WITH EXTERNAL DEVICES, ALONG WITH THE FACT THAT ONE WILL

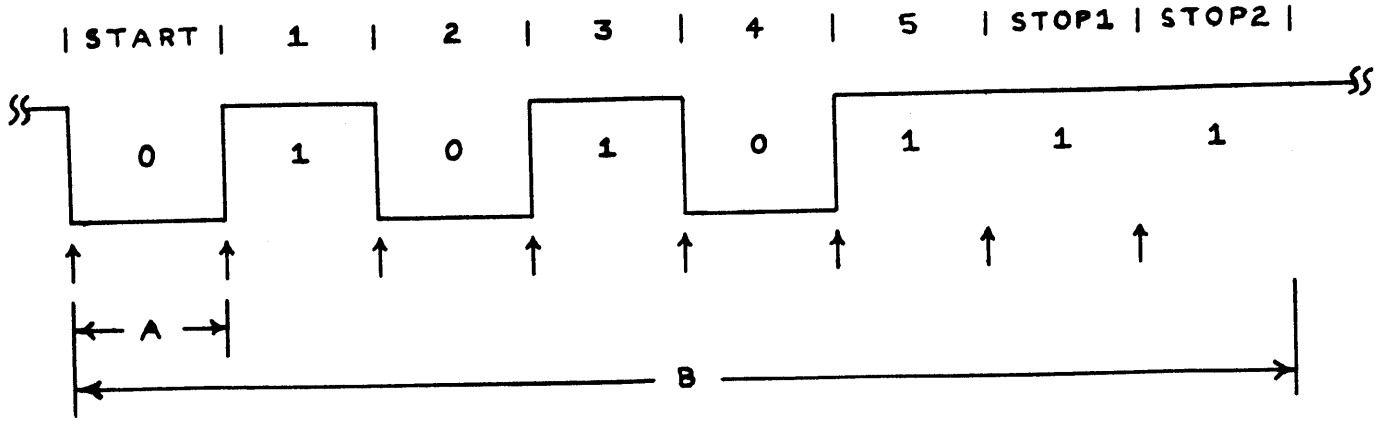
ALMOST CERTAINLY WANT TO DO SOME OTHER INSTRUCTIONS BETWEEN I/O COMMANDS, IT IS A PRETTY GOOD RULE OF THUMB TO DISQUALIFY THE USE OF AN 8008 SYSTEM AS A REAL-TIME CONTROLLER IF ANY TWO EVENTS DEPENDENT UPON TIMING FROM THE COMPUTER WILL OCCUR WITHIN 100 MICROSECONDS. A SECOND RULE OF THUMB TO IMMEDIATELY REJECT THE USE OF SUCH A SYSTEM AS A TIME DEPENDENT CONTROLLER, ONE THAT IS PRETTY MUCH DERIVED FROM EXPERIENCE, IS IF THE APPLICATION WILL REQUIRE MUCH MORE THAN 1000 I/O OPERATIONS PER SECOND. UNLESS, SUCH OPERATIONS ARE STRICTLY REPETITIVE AND THE PREVIOUS RULE CAN BE MET. THIS SECOND RULE OF THUMB IS DERIVED FROM PRACTICAL EXPERIENCE WITH "PROGRAMMING OVERHEAD" WHICH RESULTS WHEN A VARIETY OF TIME-DEPENDENT EVENTS MUST BE "JUGGLED" IN A REAL-TIME PROGRAM.

THE PROSPECTIVE REAL-TIME PROGRAMMER SHOULD BECOME FAMILIAR WITH THE LENGTHS OF TIME REQUIRED TO EXECUTE THE VARIOUS CLASSES OF INSTRUCTIONS. ONE OF THE FIRST NEW HABITS TO LEARN WHEN PREPARING REAL-TIME PROGRAMS IS TO WRITE DOWN THE EXECUTION TIME REQUIRED FOR EACH INSTRUCTION ALONGSIDE THE MNEMONIC AS THE PROGRAM IS WRITTEN. IT THEN BECOMES AN EASY MATTER TO FIGURE OUT "TOTALS" FOR VARIOUS PORTIONS OF THE ROUTINE(S). ADDITIONALLY, IT IS OFTEN HELPFUL TO WRITE DOWN THE "TOTAL" EXECUTION TIMES ALONG "PATHS" AND "LOOPS" ON A FLOW CHART OF THE PROGRAM. REAL-TIME PROGRAMMING OFTEN REQUIRES A FAIR AMOUNT OF "JUGGLING" BETWEEN CHOICES OF INSTRUCTIONS USED AND ALTERNATE SEQUENCES OF COMMANDS IN ORDER TO OBTAIN DESIRED PROGRAM EXECUTION TIMES. HAVING CRITICAL TIMING INFORMATION ON HAND IN THE FORMS SUGGESTED CAN PROVIDE THE PROGRAMMER WITH A QUICK VIEW OF HOW THE PROGRAM DEVELOPMENT EFFORT IS PROCEEDING.

IN ANY PROGRAMMING APPLICATION, FLOW CHARTING IS AN EXTREMELY VALUABLE AID TO ENABLING ONE OBTAIN AN "OVER-ALL" VIEW OF A PROGRAM'S OPERATION. IN REAL-TIME PROGRAMMING ANOTHER TOOL OF EQUAL IMPORTANCE SHOULD BE BROUGHT INTO USE. THAT TOOL IS A "TIMING DIAGRAM." A "TIMING DIAGRAM" ILLUSTRATES THE RELATIONSHIP IN TIME BETWEEN THE OCCURENCE OF SPECIFIC EVENTS OF INTEREST TO THE PROGRAMMER.

A TIMING DIAGRAM IS SHOWN ON THE TOP OF THE NEXT PAGE. THE DIAGRAM ILLUSTRATES THE DESIRED STATUS OF A SIGNAL LINE AS A FUNCTION OF TIME FOR AN ELECTRONIC SIGNAL THAT IS TO PROVIDE INFORMATION TO A "BAUDOT" TELETYPE MACHINE. THE DIAGRAM SHOWS THE SIGNAL CONDITIONS REQUIRED TO DIRECT THE MACHINE TO PRINT THE LETTER "Y" OR THE FIGURE "6" DEPENDING ON WHICH MODE THE TELETYPE IS OPERATING IN ("LETTERS" OR "FIGURES"). THIS DIAGRAM WILL BE USED TO DEVELOP A SAMPLE PROGRAM FOR OPERATING A TELETYPE PRINTER MECHANISM AS AN INTRODUCTION TO THE CONSIDERATIONS REQUIRED WHEN DEALING WITH REAL-TIME PROGRAMMING.

IN ORDER TO CLARIFY THE DIAGRAM A BRIEF EXPLANATION OF THE OPERATION OF A BAUDOT TELETYPE MACHINE WILL BE PRESENTED. A TELETYPE MACHINE IS AN "ASYNCHRONOUS" DEVICE IN THAT IT REQUIRES "START" AND "STOP" INFORMATION. ONCE THE MECHANISM IN THE TELETYPE HAS BEEN STARTED IN MOTION BY A "START" SIGNAL, THE MACHINE "EXAMINES" THE STATUS OF A SIGNAL LINE DURING SPECIFIC TIME PERIODS IN ORDER TO RECEIVE A "CODE" THAT WILL ENABLE IT TO PRINT A SPECIFIC CHARACTER. AT THE END OF THE PERIOD OF TIME OCCUPIED BY THE "CODE SIGNALS" THE MACHINE EXPECTS A "STOP" SIGNAL SO THAT VARIOUS MECHANICAL OPERATIONS MAY BE COMPLETED AND THE INTERNAL MECHANISMS SET UP TO BEGIN ANOTHER "CYCLE" OF OPERATION. WHEN DEALING WITH TELETYPE MACHINES A "CYCLE" IS OFTEN TERMED AS REQUIRING A CERTAIN NUMBER OF "UNITS OF TIME." THE DIAGRAM ILLUSTRATES A "CYCLE" FOR CERTAIN KINDS OF BAUDOT TELETYPE MACHINES. (THOSE THAT REQUIRE A "STOP" LENGTH OF TWO UNITS)! THE CYCLE IS SHOWN DIVIDED INTO 8 EQUAL UNITS OF TIME. THE FIRST UNIT OF TIME IS RESERVED FOR A "START" PULSE. BY DEFINITION, THE START PULSE MUST BE A LOGIC "0" AS SHOWN IN THE DIAGRAM.



TIMING DIAGRAM FOR SENDING BAUDOT CHARACTER "Y" OR "6" TO PRINTER

THE NEXT 5 UNITS OF TIME ARE USED TO TRANSMIT THE "BAUDOT" CODE FOR WHATEVER CHARACTER IS TO BE PRINTED BY THE MACHINE. THE LAST 2 UNITS OF TIME MUST BE A LOGIC "1" TO PLACE THE MACHINE IN THE "STOP" MODE AND ALLOW IT TO COMPLETE THE CYCLE. THE DIAGRAM ABOVE SHOWS A CYCLE IN UNITS OF TIME. TO PUT THE DIAGRAM INTO PRACTICAL USE, ONE MUST DEFINE THE UNIT OF TIME. FOR INSTANCE, SUPPOSE ONE HAD A TELETYPE MACHINE THAT USED THE CYCLE FORMAT ILLUSTRATED THAT WAS DESIGNED TO OPERATE CORRECTLY WHEN EACH UNIT OF TIME (THE LENGTH OF TIME NOTED BY THE ARROWS MARKED "A" ON THE ABOVE DIAGRAM) WAS 20 MILLISECONDS (NOMINALLY). AN ENTIRE CYCLE WOULD THUS REQUIRE 160 MILLISECONDS (FOR THE TIME SPAN MARKED "B" ON THE ABOVE DIAGRAM).

IF IT WAS DESIRED TO HAVE THE COMPUTER SEND A SIGNAL ON AN OUTPUT LINE THAT CLOSELY APPROXIMATED THE DESIRED SIGNAL PATTERN, ONE WOULD HAVE TO DEVELOP A PROGRAM THAT WOULD CHANGE THE "STATE" OF THE LINE ON AN OUTPUT PORT THAT WAS SUPPLYING THE SIGNAL TO THE MACHINE AT THE TIMES INDICATED BY THE SHORT UPWARD POINTING ARROWS SHOWN UNDERNEATH THE DIAGRAM. THE RESULTING PROGRAM WOULD BE A "REAL-TIME" PROGRAM!

REAL-TIME PROGRAMMING FOR THIS TYPE OF APPLICATION IS RELATIVELY STRAIGHT-FORWARD. FIRST OF ALL, THERE IS ONLY ONE SIGNAL LINE TO BE CONCERNED WITH (IN MANY REAL-TIME APPLICATIONS THERE MAY BE A MULTITUDE OF LINES TO CONTROL)! SECONDLY, THE AMOUNT OF TIME BETWEEN "EVENTS" IS QUITE LARGE SO THERE WILL NOT BE ANY REQUIREMENT FOR FANCY PROGRAMMING STREAMLINED FOR SPEED OF OPERATION. IN FACT, ALL ONE REALLY HAS TO DO IS MAKE SOME SIMPLE MATHEMATICAL CALCULATIONS AND DEVELOP SOME "TIMING LOOPS" THAT WILL MAKE THE PROGRAM "WAIT" FOR THE DESIRED LENGTH OF TIME BETWEEN SENDING "BITS" OF INFORMATION TO THE OUTPUT PORT THAT WILL CARRY THE SIGNAL TO THE TELETYPE UNIT. THE PROGRAM BECOMES SIMPLY A LITTLE FANCIER VERSION OF THE "PARALLEL TO SERIAL" OUTPUT PROGRAM DISCUSSED IN THE PREVIOUS CHAPTER.

A SUITABLE PROGRAM IS PRESENTED BELOW. A DISCUSSION WILL BE PRESENTED AFTER THE PROGRAM. NOTE NOW THAT THE EXECUTION TIMES HAVE BEEN PROVIDED ALONGSIDE TIME-DEPENDENT PORTIONS OF THE PROGRAM.

```
                MNEMONIC                    COMMENTS
                --------                    --------

BDOUT,  LCI 006         /SET BIT CNTR = # BITS + 1
        NDA             /SET CARRY BIT = "0"
        RAL             /BRING "0" FM CARRY INTO LSB OF ACC
```

```
24          MORBDO, OUT X        /SEND "START" OR "CODE" BITS TO MACHINE
20                  RAR          /POSITION NEXT BIT OF CODE
44 + 19,848         CAL BDELAY   /GIVE MACHINE ONE UNIT OF TIME
20                  DCC          /SEE IF FINISHED START & CODE BITS
44 / 36             JFZ MORBDO   /IF NOT, SEND NEXT BIT
32                  LAI 001      /PREPARE TO SEND STOP BITS
24                  OUT X        /SEND STOP BIT #1
44 + 19,848         CAL BDELAY   /GIVE MACHINE ONE UNIT OF TIME
44 + 20             CAL DUMMY    /PROVIDE LITTLE MORE TIME
44 + 20             CAL DUMMY    /PROVIDE LITTLE MORE TIME
24                  OUT X        /SEND STOP BIT #2
44 + 19,848         CAL BDELAY   /GIVE MACHINE ONE UNIT OF TIME
44 + 20             CAL DUMMY    /PROVIDE LITTLE MORE TIME
44 + 20             CAL DUMMY    /PROVIDE LITTLE MORE TIME
                    RET          /EXIT OUTPUT A CHARACTER RTN

20          DUMMY,  RET          /SHORT RTN TO EAT UP TIME

32          BDELAY, LDI 215      /SET TIMER LOOP COUNTER
24                  OUT Z        /OUTPUT TO UNUSED PORT TO TRIM TIME
24                  OUT Z        /OUTPUT TO UNUSED PORT TO TRIM TIME
44 + 20             CAL DUMMY    /USE A LITTLE TIME BEFORE STARTING LOOP
44 + 20     MDELAY, CAL DUMMY    /FOR A TIME CONSUMING LOOP
20                  DCD          /SEE IF TIME EXPIRED (CNTR = 0)?
12 / 20             RTZ          /EXIT BACK TO CALLING RTN WHEN FINISHED
44                  JMP MDELAY   /OTHERWISE CONTINUE USING UP TIME
```

THE ABOVE ROUTINE ASSUMED THAT THE DATA TO THE TELETYPE MACHINE ORIGINATED FROM THE LEAST SIGNIFICANT BIT IN THE ACCUMULATOR.

THE READER SHOULD NOTE THAT FOR CASES WHERE THERE ARE TWO POSSIBLE EXECUTION TIMES FOR AN INSTRUCTION, SUCH AS A CONDITIONAL INSTRUCTION, THAT THE TIME REQUIRED FOR THE CONDITION "MOST OFTEN" TO OCCUR IN THE PROGRAM WAS SHOWN FIRST, FOLLOWED BY THE TIME REQUIRED WHEN THE OTHER CONDITION OCCURED.

THE PROGRAM WAS INITIALLY DEVELOPED BY WRITING THE "MAIN" PORTION WITH THE TIME REQUIRED FOR THE "BDELAY" SUBROUTINE CONSIDERED AS AN "UNKNOWN" FACTOR. WHEN THE BASIC FORMAT OF THE PROGRAM HAD BEEN DETERMINED THE EXECUTION TIME OF THE "LOOP" STARTING AT THE LABEL "MORBDO" WHICH INCLUDED THE FIVE INSTRUCTIONS:

```
MORBDO, OUT X
        RAR
        CAL BDELAY
        DCC
        JFZ MORBDO
```

WAS CALCULATED - LEAVING OUT THE AS YET UNDETERMINED TIME OF "BDELAY." THE TIME REQUIRED BY THE FIVE INSTRUCTIONS WHEN "LOOPING" WAS FOUND TO BE 152 MICROSECONDS. SINCE IT WAS KNOWN THAT A TOTAL OF 20,000 MICROSECONDS (20 MILLISECONDS) WAS DESIRED BETWEEN OUTPUTTING EACH BIT IN THE "CODE" IT WAS THEN EASY TO CALCULATE THAT 20,000 - 152 = 19,848 MICROSECONDS DELAY WAS REQUIRED IN "BDELAY."

THE SUBROUTINE "BDELAY" IS A TYPICAL EXAMPLE OF A TIMING DELAY LOOP. THE MAIN PORTION OF THE DELAY LOOP STARTS AT "MDELAY" AND INCLUDES THE FOUR INSTRUCTIONS:

```
MDELAY, CAL DUMMY
       DCD
       RTZ
       JMP MDELAY
```

THE THEORY BEHIND THE "BDELAY" SUBROUTINE WAS TO EXECUTE THE "MDELAY" LOOP THE REQUIRED NUMBER OF TIMES TO GET CLOSE TO A DELAY OF 19,848 MICROSECONDS AND THEN CLOSE ANY GAP BY THE "SET UP" INSTRUCTION FOR THE "LOOP" AND PERHAPS A FEW "FILLER" INSTRUCTIONS.

THE TIME REQUIRED TO COMPLETE THE FOUR INSTRUCTIONS IN THE "MDELAY" LOOP WHEN THE "RTZ" CONDITION IS NOT MET IS 140 MICROSECONDS. FINDING OUT HOW MANY TIMES IT IS NECESSARY TO EXECUTE THE LOOP TO GET CLOSE TO A DELAY OF 19,848 MICROSECONDS IS A SIMPLE MATTER OF DIVIDING. DOING SO YIELDED A FIGURE OF ALMOST 142 (DECIMAL). TAKING INTO ACCOUNT THE FACT THAT IT WAS NOT DESIRABLE TO GO OVER THE ALLOTED TIME, AND THE FACT THAT SETTING UP THE LOOP WOULD TAKE SOME TIME, THE FIGURE OF 141 DECIMAL WAS CHOSEN - WHICH IS 215 OCTAL. ONE OTHER FACTOR HAD TO BE CONSIDERED. WHEN THE COUNTER IN THE LOOP REACHED ZERO, THE "RTZ" INSTRUCTION WOULD BE EXECUTED AND THE "JMP MDELAY" COMMAND WOULD NOT. THUS, THE FULL LOOP WOULD ONLY BE EXECUTED 140 (DECIMAL) TIMES - THE LAST TIME THROUGH THE "MDELAY" ROUTINE WOULD ONLY TAKE 104 MICROSECONDS. THUS, AT THIS POINT IT WAS POSSIBLE TO CALCULATE THE TOTAL DELAY CAUSED BY EXECUTING THE "MDELAY" LOOP THE SELECTED NUMBER OF TIMES: 140 X 140 = 19,600 PLUS 104 FOR A TOTAL OF 19,704 MICROSECONDS. THEN IT WAS AN EASY MATTER TO DETERMINE HOW MUCH TIME TO USE TO "SET UP" THE "MDELAY" ROUTINE. THE DESIRED TOTAL DELAY OF 19,848 MINUS THE 19,704 MICROSECONDS CONSUMED BY EXECUTING THE "MDELAY" ROUTINE 141 (DECIMAL) TIMES LEFT 144 MICROSECONDS TO BE CONSUMED. THE "LDI 215" AT THE START OF "BDELAY" ONLY REQUIRED 32 MICROSECONDS SO 112 MORE MICROSECONDS WERE CONSUMED BY ADDING THE "FILLER" INSTRUCTIONS "CAL DUMMY" AND TWO "OUT X" COMMANDS. THE TOTAL "BDELAY" SUBROUTINE THEN EQUALLED EXACTLY THE DESIRED DELAY TIME OF 19,848 MICROSECONDS!

AFTER SENDING THE START AND 5 CODE BITS IT WAS NECESSARY TO SEND A "TWO UNIT" STOP PULSE. SINCE THE STOP PULSE BY DEFINITION WAS TO BE A LOGIC "1," IT WAS NECESSARY TO SET UP THE STOP BIT AS A "1" IN THE ACCUMULATOR. THE READER CAN CALCULATE THAT THE ACTUAL DELAY BETWEEN THE SENDING OF THE LAST CODE BIT AND THE FIRST "STOP" UNIT IN THE ROUTINE COMES OUT TO BE 20,024 MICROSECONDS. REMEMBER, IN MAKING THE CALCULATION THAT THE "JFZ MORBDO" INSTRUCTION WILL ONLY REQUIRE 36 MICROSECONDS ON THE FINAL EXECUTION OF THE "LOOP" THEREBY REDUCING THE LOOP EXECUTION TIME TO 19,992 MICROSECONDS AND THE "LAI 001" WILL ADD 32 MICROSECONDS TO THAT VALUE BEFORE THE NEXT "OUT X" INSTRUCTION CAN BE EXECUTED. HOWEVER, FOR THE APPLICATION, THE VALUE OF 20,024 IS PLENTY CLOSE ENOUGH TO 20,000 (OFF BY ABOUT 0.1 %) TO OPERATE A TELETYPE WHICH CAN TYPICALLY OPERATE RELIABLY WITH THE TIMING OFF BY 10 TO 20 PERCENT!

THE DELAY BETWEEN THE FIRST STOP UNIT AND THE SECOND, AS WELL AS THE FINAL DELAY TO COMPLETE THE SECOND STOP UNIT, WAS MADE TO COME OUT NICELY TO 20,000 MICROSECONDS BY THE INSERTION OF THE "CAL DUMMY" COMMANDS FOLLOWING THE "CAL BDELAY" INSTRUCTIONS.

THE ABOVE ROUTINE, AS THE READER CAN UNDOUBTABLY SEE, COULD BE MODIFIED TO SERVE TO OPERATE A VARIETY OF TELETYPE MACHINES OPERATING AT DIFFERENT SPEEDS BY CHANGING THE "TIMING LOOPS." THE PROGRAM COULD ALSO BE MODIFIED FOR ASCII CODED MACHINES, OR OTHER TYPES OF CODES BY CHANGING THE "BIT COUNTER" AND POSSIBLY ALTERING THE LENGTH OF THE "STOP" PULSE DEPENDING ON THE TYPE OF MACHINE BEING DRIVEN. FURTHERMORE, THE TECHNIQUES DEMONSTRATED CAN BE APPLIED TO MANY OTHER TYPES OF PROBLEMS.

A SIMILAR ROUTINE COULD BE DEVELOPED TO RECEIVE DATA FROM THE SAME KIND OF BAUDOT MACHINE. HOWEVER, WHEN RECEIVING DATA FROM SUCH A UNIT THERE ARE A FEW NEW CONCEPTS TO CONSIDER.

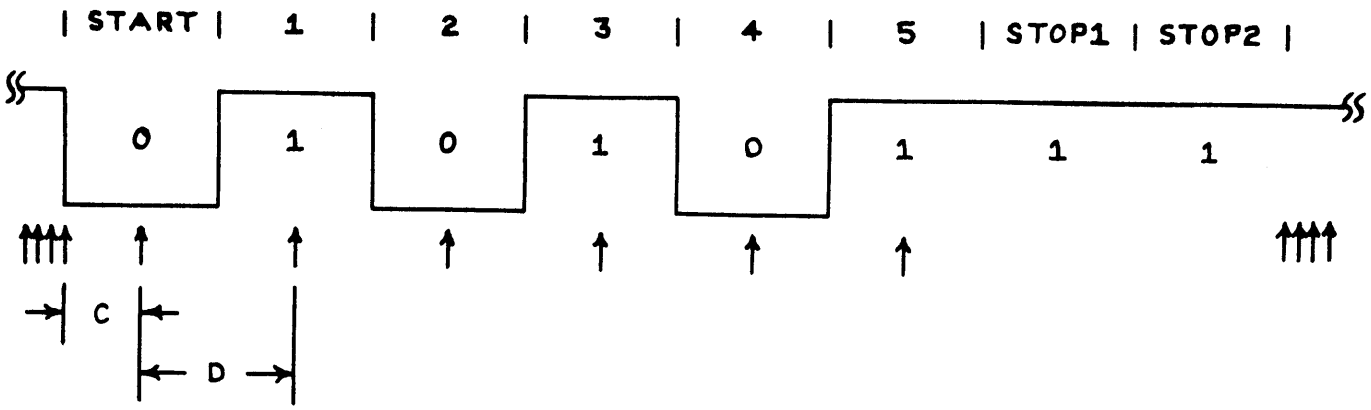
WHEN THE COMPUTER WAS SENDING INFORMATION TO THE TELETYPE PRINTER IT HAD AN ADVANTAGE IT WILL NOT HAVE WHEN IT IS USED TO RECEIVE INFORMATION FROM THE MACHINE. NAMELY, WHEN TRANSMITTING, THE COMPUTER HAD "CONTROL" OF WHEN THE EXTERNAL MACHINE WOULD BE OPERATED. IN THE REVERSE MODE, THE COMPUTER WILL HAVE NO "KNOWLEDGE" OF WHEN THE EXTERNAL DEVICE WILL BEGIN TO OPERATE AND SEND DATA TO THE COMPUTER!

ADDITIONALLY, ONCE A "CHARACTER" STARTS ARRIVING ON A LINE OF AN INPUT PORT, THE "TOLERANCE" SITUATION REVERSES. WHAT IS MEANT BY THIS IS THAT WHEN THE COMPUTER SENT DATA TO THE PRINTER MECHANISM, IT WAS POSSIBLE FOR THE COMPUTER TO BE MUCH MORE ACCURATE IN PROVIDING PROPER TIMING TO THE MACHINE, THAN THE MACHINE REQUIRED TO OPERATE SUCCESSFULLY. THUS, IF THE TIME PERIOD FOR A "UNIT" OF TIME WAS OFF A FEW TENTHS OF A PERCENT WHEN GENERATED BY THE COMPUTER, IT WOULD NOT AFFECT THE OPERATION OF THE MACHINE. HOWEVER, WHEN THE COMPUTER IS RECEIVING DATA FROM THE MACHINE, THE START OF EACH UNIT OF TIME MAY BE OFF BY AS MUCH AS 10 PERCENT OR SO, BECAUSE OF THE LOOSE TOLERANCE OF THE ELECTRO-MECHANICAL MACHINERY INVOLVED. IF THE COMPUTER PROGRAM DOES NOT MAKE PROPER ALLOWANCES FOR SUCH POSSIBLE VARIATIONS, THEN "INCORRECT" DATA MAY BE RECEIVED.

FORTUNATELY, THE PROBLEMS RELATED TO THESE CONCEPTS ARE NOT TOO DIFFICULT TO OVERCOME. THE FIRST PROBLEM, DETERMINING WHEN THE EXTERNAL MACHINE IS STARTING TO SEND, CAN BE SOLVED BY PERIODICALLY CHECKING THE INPUT LINE FOR THE PRESENCE OF A "0" CONDITION INDICATING A "START" BIT. (NOTE: WHILE THERE IS ANOTHER MANNER IN WHICH ONE COULD DETECT THE BEGINNING OF AN EXTERNAL OPERATION IN PROPERLY EQUIPPED 8008 SYSTEMS, THROUGH THE USE OF A HARDWARE GENERATED "INTERRUPT" SCHEME, SUCH A METHOD IS MORE PROPERLY CONCERNED WITH HARDWARE CONSIDERATIONS WHICH ARE NOT WITHIN THE INTENDED SUBJECT MATTER OF THIS MANUAL. IF SUCH A DETECTION SCHEME WERE USED, THE REMAINDER OF THIS DISCUSSION ON HANDLING THE RECEIPT OF THE INCOMING DATA WOULD STILL APPLY). NATURALLY, HOW OFTEN ONE CHECKED FOR THE PRESENCE OF A "START" BIT WOULD HAVE AN AFFECT ON THE OVER-ALL ABILITY OF A REAL-TIME PROGRAM TO RECEIVE THE DATA. FOR INSTANCE, ASSUMING A START BIT IS PRESENT FOR 20 MILLISECONDS AS IN THE CASE FOR THE TYPE OF MACHINE BEING DISCUSSED, IT WOULD BE FOOLISH TO TEST FOR THE PRESENCE OF SUCH A "START" BIT AT PERIODS THAT WERE 21 MILLISECONDS APART! IN FACT, BECAUSE OF OTHER CONSIDERATIONS, IT WOULD NOT BE WISE TO CHECK FOR A "START" BIT MUCH LESS OFTEN THAN EVERY FEW MILLISECONDS.

THE SECOND PROBLEM OF DEALING WITH THE LOOSE TOLERANCE OF THE MACHINERY CAN BE EFFECTIVELY DEALT WITH BY ADJUSTING THE RECEIVE ROUTINE SO THAT IT "SAMPLES" THE INCOMING SIGNAL AT THE THEORETICAL MIDDLE OF A "UNIT" OF TIME RATHER THAN AT THE BEGINNING OR END OF A TIME PERIOD. OF COURSE THE ABILITY TO DO THIS ALSO DEPENDS ON HOW CLOSELY ONE IS ABLE TO DETECT THE ACTUAL "START" OF A CHARACTER FROM THE MACHINE.

A TIMING DIAGRAM SHOWING A "BAUDOT" CHARACTER BEING SENT BY A MACHINE IS ILLUSTRATED AT THE TOP OF THE NEXT PAGE. SHORT UPWARD POINTING ARROWS ALONG THE BOTTOM OF THE DIAGRAM ILLUSTRATE THE TIMES AT WHICH A "REAL-TIME" PROGRAM WOULD NEED TO "SAMPLE" THE INCOMING LINE IN ORDER TO CORRECTLY RECEIVE THE DATA. NOTE THAT PRIOR TO THE TIME A "START" SIGNAL IS DETECTED, THE COMPUTER SHOULD SAMPLE THE LINE OFTEN IN ORDER TO MINIMIZE THE PERIOD OF TIME IN WHICH A START SIGNAL MAY BE PRESENT BUT UNDETECTED. NEXT, IT IS DESIRABLE TO ADJUST THE "SAMPLE" PERIOD SO THAT IT COINCIDES WITH THE THEORETICAL MIDDLE OF A UNIT OF TIME, RATHER



TIMING DIAGRAM FOR RECEIVING BAUDOT CHARACTER "Y" OR "6"

THAN SAMPLE AT INTEGERS OF UNITS OF TIME AFTER THE START SIGNAL WAS DETECTED. THIS METHOD COMPENSATES FOR THE "TOLERANCE" PROBLEM MENTIONED PREVIOUSLY.

FINALLY, AFTER THE 5'TH CODE BIT HAS BEEN RECEIVED, ONE CAN OBSERVE THAT IT WILL NOT BE NECESSARY TO START TESTING FOR A NEW "START" PULSE FOR ABOUT 2 AND 1/2 TIME UNITS AS IT IS KNOWN THAT THE MACHINE WILL BE USING THAT TIME TO COMPLETE IT'S OPERATION. THUS, THE COMPUTER WOULD BE ABLE TO PERFORM SOME OTHER FUNCTIONS FOR ABOUT 50 MILLISECONDS BEFORE GOING BACK TO THE "SAMPLE" MODE TO LOOK FOR A NEW START BIT - THAT IS ENOUGH TIME TO PERFORM A FEW THOUSAND INSTRUCTIONS ON AN 8008 SYSTEM!

A SAMPLE ROUTINE FOR RECEIVING INFORMATION FROM A DEVICE IN ACCORDANCE WITH THE ABOVE DIAGRAM, ASSUMING THAT THE TIME SPAN MARKED "C" IN THE ABOVE DIAGRAM WAS 10 MILLISECONDS, AND THAT MARKED "D" WAS 20 MILLISECONDS IS ILLUSTRATED NEXT. THE READER MAY NOTE THAT IT IS ESSENTIALLY AN EXPANDED VERSION OF A "SERIAL TO PARALLEL" ROUTINE WITH INSTRUCTIONS TO CONTROL THE TIMING ADDED.

```
                        MNEMONIC                    COMMENTS
                    ----------------            ----------------

            BDIN,   LBI 000         /CLEAR INCOMING FORMING & STORAGE REGISTER
                    LCI 005         /SET BIT COUNTER
32          STRTIN, INP X           /LOOK FOR "START" BIT
32                  NDI 200         /MASK OFF IRRELEVANT DATA
44 / 36             JTS STRTIN      /IF NO START BIT, FORM "SAMPLING LOOP"
44 + 9796           CAL HDELAY      /IF FIND LOGIC "0" ASSUME START, DELAY
32                  INP X           /TO MIDDLE OF START UNIT & VERIFY RECEIPT
32                  NDI 200         /OF A START BIT BY MAKING APPROPRIATE TEST
36 / 44             JTS STRTIN      /IF NOT "0" HERE ASSUME FALSE START
44 + 20             CAL DUMMY       /STRETCH THE DELAY A LITTLE
44                  JMP MORBDI      /STRETCH THE DELAY A LITTLE MORE
44+19748    MORBDI, CAL IDELAY      /MAIN DELAY LOOP = ALMOST 1 FULL TIME UNIT
32                  INP X           /GET NEXT BIT
32                  NDI 200         /TRIM TO JUST DESIRED DATA BIT
20                  RAL             /SAVE INCOMING BIT IN CARRY FLAG
20                  LAB             /GET ANY PREVIOUS BITS
20                  RAR             /ROTATE NEW BIT FROM CARRY INTO REGISTER
20                  LBA             /SAVE IN REGISTER "B"
20                  DCC             /DECREMENT BITS COUNTER
44 / 36             JFZ MORBDI      /DELAY & FETCH NEXT INCOMING BIT
```

```
                        MNEMONIC                COMMENTS
                        --------                --------

20                      RRC             /HAVE ALL 5 BAUDOT BITS - RIGHT JUSTIFY
20                      RRC             /IN ACCUMULATOR BY ROTATES
20                      RRC             /BEFORE PREPARING TO EXIT RTN
44 + 9796               CAL HDELAY      /OPTIONAL DELAY TO MAKE SURE INTO "STOP"
44 + 20                 CAL DUMMY       /PART OF OPTIONAL DELAY
44 + 20                 CAL DUMMY       /PART OF OPTIONAL DELAY
20                      RET             /UNITS AREA BEFORE EXITING ROUTINE

32              IDELAY, LDI 215         /SET TIME LOOP COUNTER
12                      RTS             /TRIM TIME - CONDX NEVER MET
44 + 20         RDELAY, CAL DUMMY       /TIME CONSUMING LOOP
20                      DCD             /DECREMENT COUNTER
12 / 20                 RTZ             /EXIT TO CALLING RTN WHEN CNTR = 0
44                      JMP RDELAY      /OTHERWISE CONTINUE USING UP TIME

32              HDELAY, LDI 106         /SET TIME LOOP COUNTER
44                      JMP RDELAY      /GO USE UP ABOUT 1/2 A TIME UNIT

20              DUMMY,  RET             /SHORT RTN TO USE UP TIME
```

WHILE THE ABOVE ROUTINE IS SIMILAR IN MANY RESPECTS TO THE ONE DESCRIBED EARLIER FOR TRANSMITTING DATA FROM THE COMPUTER, SEVERAL DIFFERENT FEATURES WILL BE HIGH-LIGHTED. FIRST, THE READER CAN NOTE THAT THE PROGRAM EXPECTS DATA TO BE ARRIVING AT THE MOST SIGNIFICANT BIT POSITION OF THE ACCUMULATOR (AS IN THE SERIAL TO PARALLEL ROUTINE IN THE PREVIOUS CHAPTER).

NEXT, THE READER SHOULD NOTE THAT THE THREE INSTRUCTIONS STARTING AT THE LABEL "STRTIN" FORM A "LOOP" TO TEST FOR A "START" BIT ARRIVING FROM THE INPUT PORT. THE READER CAN SEE THAT THE LOOP REQUIRES 108 MICROSECONDS TO EXECUTE AND THUS IT IS POSSIBLE FOR A START UNIT TO HAVE BEEN PRESENT FOR ALMOST THAT LENGTH OF TIME BEFORE IT IS DETECTED. FOR INSTANCE, IF THE START PULSE ACTUALLY STARTED JUST A MICROSECOND AFTER THE "INP X" INSTRUCTION AT "STRTIN" WAS EXECUTED, THAT PULSE WOULD NOT BE DETECTED UNTIL THE "INP X" INSTRUCTION WAS EXECUTED ON THE NEXT ROUND. HOWEVER, IT IS ALSO POSSIBLE FOR THE PROGRAM TO DETECT THE START BIT AT JUST ABOUT THE INSTANT IT ACTUALLY HAPPENS - THUS, THERE CAN BE A VARIATION IN DETECTING THE BEGINNING OF THE "START" TIME UNIT OF ABOUT 108 MICROSECONDS. NOW, THE ACTUAL DETECTION OF THE START PULSE IS USED AS A REFERENCE FOR "DELAYING" TO THE MIDDLE OF THE TIME UNIT IN ORDER TO "SAMPLE" THE REMAINING BITS IN THE DESIRED REGION. ON THE AVERAGE, ONE COULD ASSUME THAT THE START PULSE WAS DETECTED IN ABOUT THE MIDDLE OF THE POSSIBLE RANGE OF VARIATION, WHICH WOULD BE ABOUT 54 MICROSECONDS AFTER THE PULSE ACTUALLY STARTED. THIS INFORMATION IS USED TO ESTABLISH APPROXIMATELY HOW LONG THE "HDELAY" LOOP SHOULD BE IN ORDER TO GET CLOSE TO THE THEORETICAL MIDDLE OF A TIME UNIT. THUS, IF ONE ASSUMES THAT ON AN AVERAGE, THE START PULSE IS DETECTED 54 MICROSECONDS AFTER IT BEGAN, AND ONE ADDS 144 MICROSECONDS FOR THE EXECUTION OF THE INSTRUCTIONS FROM "STRTIN" TO THE "CAL HDELAY," ONE CAN DETERMINE THAT "HDELAY" NEEDS TO CONSUME 9802 MICROSECONDS. THE VALUE 9796 ACTUALLY DEVELOPED WAS A "CLOSE ENOUGH" COMPROMISE FOR THE SITUATION.

ANOTHER AREA OF INTEREST NEAR THE END OF THE MAIN ROUTINE IS MARKED BY THE COMMENTS AS AN "OPTIONAL DELAY TO MAKE SURE INTO "STOP" UNITS AREA BEFORE EXITING ROUTINE." AS POINTED OUT EARLIER, AFTER THE FIVE DATA BITS HAVE BEEN SAMPLED THE COMPUTER HAS QUITE A BIT OF TIME - UP TO ABOUT 58 MILLISECONDS IN WHICH TO PERFORM SOME OTHER FUNCTIONS BE-

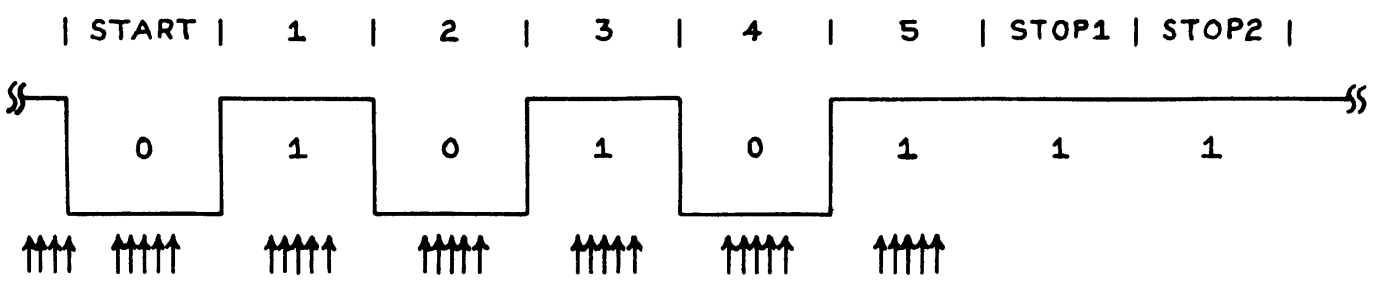
CAUSE THE MODEL MACHINE WOULD BE UNABLE TO SEND A NEW "START" PULSE UNTIL IT HAD COMPLETED IT'S CYCLE DENOTED BY THE TWO STOP UNITS IN THE DIAGRAM. HOWEVER, IN SOME INSTANCES, THE COMPUTER MAY NOT REQUIRE ANY WHERE NEAR THAT LENGTH OF TIME TO PROCESS THE CHARACTER JUST RECEIVED. IN SUCH CASES, THE PROGRAMMER WOULD WANT TO MAKE SURE THE PROGRAM DID NOT START "LOOKING" FOR A NEW START BIT BEFORE THE LAST "DATA" BIT HAD BEEN COMPLETED. THE "OPTIONAL" HALF UNIT DELAY ENSURES IN SUCH A CASE THAT THE MACHINE WOULD BE IN IT'S "STOP UNITS" PHASE, WHICH BY PREVIOUS DEFINITION WOULD BE A LOGIC "1" CONDITION, BEFORE IT BEGAN "LOOKING" FOR A NEW LOGIC "Ø" CONDITION SIGNIFYING A NEW START PULSE.

FINALLY, THE READER MIGHT TAKE NOTE OF AN INTERESTING "TRICK" TO GET A RATHER SHORT ADDITIONAL DELAY BY THE USE OF THE "RTS" INSTRUCTION AS THE SECOND COMMAND IN THE "IDELAY" SUBROUTINE. A CONDITIONAL RETURN INSTRUCTION WHEN THE CONDITION IS NOT MET IS THE ONLY TYPE OF COMMAND THAT WILL USE BUT 12 MICROSECONDS OF TIME. THE "RTS" INSTRUCTION INSERTED AT THAT POINT WILL NEVER HAVE THE TRUE CONDITION MET AS THE READER MAY VERIFY BY CLOSE EXAMINATION OF THE POSSIBLE CONDITION OF THE "SIGN" FLAG WHENEVER THAT INSTRUCTION IS EXECUTED. IT IS A GOOD TECHNIQUE TO REMEMBER IF A 12 MICROSECOND DELAY IS REQUIRED BUT THE PROGRAMMER MUST MAKE CERTAIN THAT THE CONDITION WILL NEVER BE SATISFIED WHEN USED FOR THAT PURPOSE! (REMEMBER, VIRTUALLY ALL OTHER TYPES OF INSTRUCTIONS TAKE UP AT LEAST 20 MICROSECONDS OF EXECUTION TIME IN A NOMINALLY ADJUSTED 8008 SYSTEM).

AS ANOTHER EXAMPLE OF THE DETAILS OF REAL-TIME PROGRAMMING, THE ABOVE EXAMPLE WILL BE EXPANDED TO DEMONSTRATE HOW THE PROGRAM COULD BE IMPROVED TO INCREASE THE RELIABILITY OF RECEIVING CORRECT DATA FROM THE EXTERNAL MACHINE. AS MANY READERS MAY KNOW, THE INCOMING DATA FROM AN ELECTRO-MECHANICAL MACHINE SUCH AS A TELETYPE MAY BE "NOISY." THAT IS, A SIGNAL THAT IS SUPPOSED TO BE, FOR INSTANCE, IN THE LOGIC "1" STATE FOR AN ENTIRE UNIT OF TIME MAY OCCASIONALLY GO TO THE "Ø" CONDITION FOR SMALL FRACTIONS OF A UNIT OF TIME, OR VICE-VERSA. IN THE ABOVE PROGRAM THE COMPUTER "SAMPLES" FOR THE STATE OF THE INCOMING SIGNAL JUST ONCE IN EACH UNIT OF TIME. IF BY CHANCE IT SHOULD SAMPLE THE SIGNAL AT THE MOMENT THAT "NOISE" WAS PRESENT, INCORRECT DATA COULD BE RECEIVED. IN A "CRITICAL" APPLICATION, IT MIGHT BE DESIRABLE TO REDUCE THE CHANCE OF SUCH AN ERROR OCCURING. THIS COULD BE DONE BY "SAMPLING" THE INCOMING SIGNAL SEVERAL TIMES DURING EACH UNIT OF TIME AND COMPUTING AN AVERAGE OF THE "VALUE" RECEIVED TO DETERMINE WHETHER THE SIGNAL WAS TRULY IN A "1" OR "Ø" STATE. FOR INSTANCE, ONE COULD ELECT TO "SAMPLE" THE SIGNAL FIVE TIMES NEAR THE "MIDDLE" OF EACH UNIT OF TIME AND THEN MAKE A DECISION AS TO WHETHER THE SIGNAL WAS A "1" OR A "Ø" BY DETERMINING WHICH STATE WAS DETECTED 3 OR MORE OUT OF THE 5 SAMPLED TIMES. SUCH A "SAMPLING" METHOD WOULD GREATLY REDUCE THE CHANCES OF "NOISE" CAUSING AN INCORRECT SIGNAL LEVEL TO BE RECEIVED.

THE TIME DIAGRAM AT THE TOP OF THE NEXT PAGE ILLUSTRATES A SIGNAL WITH THE UPWARD ARROWS ALONG THE BOTTOM OF THE DIAGRAM REPRESENTING THE MULTIPLE SAMPLING POINTS IN EACH UNIT OF TIME. DEVELOPING A PROGRAM TO GIVE THE IMPROVED PERFORMANCE IS NOT DIFFICULT BUT IT DOES REQUIRE A FEW MORE TIME RELATED CONSIDERATIONS WHEN DEVELOPING THE "SOFTWARE." THESE ILLUSTRATIONS WILL BE POINTED OUT IN THE DISCUSSION THAT FOLLOWS.

TO BEGIN DEVELOPMENT OF THE MULTIPLE-SAMPLING PROGRAM A MAJOR SUBROUTINE WAS DEVELOPED THAT WOULD PERFORM THE TASK OF "SAMPLING" FIVE TIMES IN SUCCESSION, KEEPING TRACK OF WHETHER A "1" OR "Ø" WAS RECEIVED, AND FINALLY DETERMINING WHICH STATE WAS RECEIVED MOST OFTEN. THE SUBROUTINE WITH EXECUTION TIMES FOR EACH INSTRUCTION IS PRESENTED AFTER THE DIAGRAM ON THE NEXT PAGE. THE READER MIGHT PAY SPECIAL ATTENTION TO THE MANNER IN WHICH THE "PREDOMINANT" SIGNAL STATE WAS DETERMINED.



TIMING DIAGRAM FOR MULTIPLE SAMPLING OF INCOMING SIGNAL

```
                        MNEMONIC                    COMMENTS
                    ----------------            ----------------

32          SAMPLE, LDI 005         /SET COUNTER FOR NUMBER OF SAMPLES
32                  LEI 377         /SET UP REG "E" FOR STORING SIGNAL STATE
32          BITEST, INP X           /SAMPLE CURRENT SIGNAL ON INPUT LINE
32                  NDI 200         /MASK OFF UNUSED INPUT LINES
44 / 36             CTS PLUSE       /INCREMENT "E" IF SIGNAL A LOGIC "1"
32                  NDI 200         /RESTORE FLAGS TO REFLECT ACC CONTENTS
36 / 44             CFS MINUSE      /DECREMENT "E" IF SIGNAL A LOGIC "0"
20                  DCD             /DECREMENT SAMPLING COUNTER
44 / 36             JFZ BITEST      /SAMPLE AGAIN IF COUNTER NOT = 0
20                  LAE             /WHEN HAVE 5 SAMPLES PLACE "E" INTO ACC
32                  NDI 200         /MASK OFF ALL BUT MOST SIGNIFICANT BIT
20                  RET             /EXIT WITH PREDOM SIG STATE IN MSB OF ACC

20          PLUSE,  INE             /INCREMENT REGISTER "E"
20                  RET             /EXIT

20          MINUSE, DCE             /DECREMENT REGISTER "E"
20                  RET             /EXIT
```

INFORMATION REGARDING THE AMOUNT OF TIME REQUIRED TO EXECUTE PORTIONS OF THE "MULTIPLE SAMPLING" ROUTINE JUST PRESENTED IS REQUIRED BEFORE THE OVER-ALL ROUTINE CAN BE DEVELOPED FOR REASONS THAT WILL SOON BE APPARENT.

THE READER CAN CONFIRM THAT THE TIME BETWEEN EACH OF THE FIVE SAMPLES WILL BE 280 MICROSECONDS FOR A TYPICAL 8008 SYSTEM REGARDLESS OF WHAT SIGNAL STATE WAS RECEIVED. IT IS IMPORTANT TO NOTICE HOW THE SAMPLING ROUTINE WAS "BALANCED" BY THE APPROPRIATE CHOICE OF INSTRUCTIONS SO THAT THE RECEIPT OF EITHER SIGNAL STATE RESULTS IN THE SAME TOTAL TIME TO EXECUTE THE "SAMPLING LOOP." IF THIS REQUIREMENT WERE NOT MET THE PROGRAMMER WOULD HAVE QUITE A "HEAD-ACHE" TRYING TO DEVELOPE AN ACCURATE ROUTINE BASED ON ALL THE POSSIBLE COMBINATIONS OF "1" AND "0" SIGNAL STATES THAT COULD BE RECEIVED!

THE READER SHOULD ALSO TAKE NOTE THAT THE "SET UP" TIME, THAT IS THE TIME TO EXECUTE THE INSTRUCTIONS FROM THE LABEL "SAMPLE" TO "BITEST" PLUS THE TIME TO ACTUALLY "CALL" THE SUBROUTINE WOULD REQUIRE 108 MICROSECONDS. THAT IS, IT WILL TAKE 108 MICROSECONDS FROM THE TIME THE PROGRAM STARTS TO "CALL" THE SUBROUTINE UNTIL THE FIRST "INP X" INSTRUCTION IS ENCOUNTERED.

ADDITIONALLY, THE READER SHOULD NOTE THAT IT WILL REQUIRE 344 MICROSECONDS FROM THE TIME THE 5'TH SAMPLE IS TAKEN UNTIL THE SUBROUTINE IS

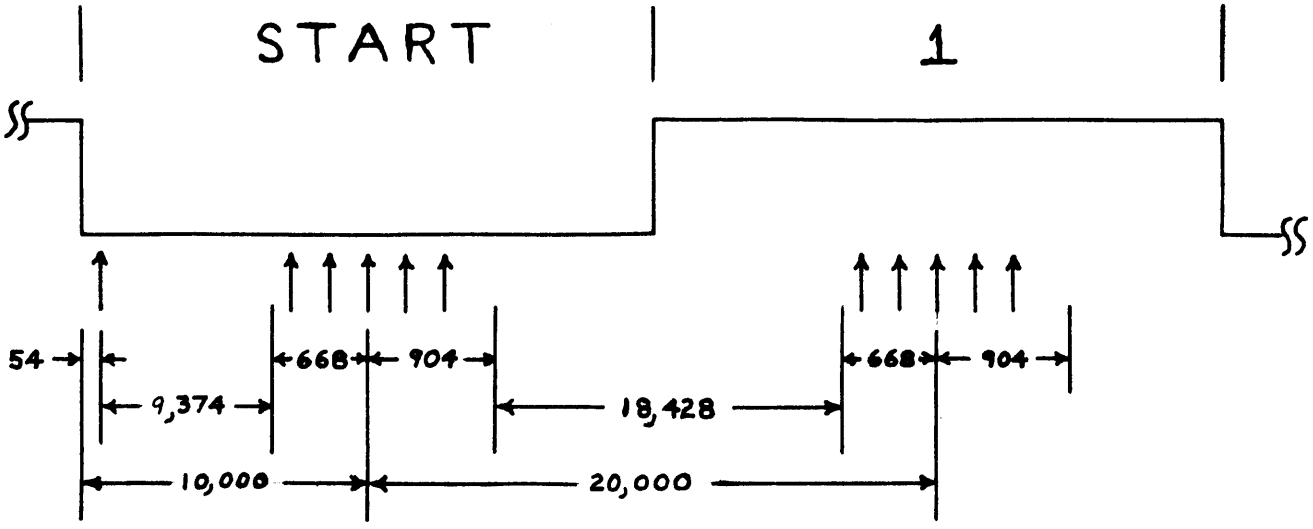
ACTUALLY EXITED!

IT IS IMPORTANT TO KNOW THESE RELATIONSHIPS SO THAT THE ENTIRE SUBROUTINE CAN BE PROPERLY LOCATED WITHIN A TIME FRAME. FOR INSTANCE, SINCE IT WOULD BE DESIRABLE TO HAVE THE 3'RD "SAMPLE" TAKE PLACE AT THE THEORETICAL "MIDDLE" OF A "UNIT OF TIME" IT WILL BE NECESSARY TO START "CALLING" THE "SAMPLE" SUBROUTINE WHEN THERE ARE ABOUT 668 MICROSECONDS REMAINING BEFORE THE THEORETICAL MIDDLE OF THE "UNIT OF TIME." THIS IS BECAUSE IT WILL REQUIRE 108 MICROSECONDS TO "CALL" AND "SET UP" THE SAMPLING SUBROUTINE, PLUS 280 MICROSECONDS BETWEEN THE 1'ST AND 2'ND SAMPLE AND ANOTHER 280 MICROSECONDS BETWEEN THE 2'ND AND 3'RD SAMPLE.

SIMILARLY IT IS IMPRTANT TO KNOW THAT THERE WILL BE 904 MICROSECONDS FROM THE TIME THE 3'RD SAMPLE IS TAKEN UNTIL THE ROUTINE IS EXITED. AS, 280 MICROSECONDS WILL BE TAKEN BETWEEN SAMPLE NUMBER 3 AND 4, ANOTHER 280 MICROSECONDS BETWEEN SAMPLE 4 AND 5, AND AN ADDITIONAL 344 MICROSECONDS FROM SAMPLE NUMBER 5 TO THE TIME THE ROUTINE IS EXITED.

WITH THIS INFORMATION NOW AVAILABLE ONE CAN CALCULATE HOW MUCH TIME SHOULD BE USED FROM THE TIME A START BIT IS RECEIVED UNTIL IT IS TIME TO "CALL" THE "SAMPLE" SUBROUTINE SO THAT THE 3'RD SAMPLE POINT WILL BE IN THE MIDDLE OF A "UNIT OF TIME." AND, AFTER THAT, HOW MUCH DELAY TO PROVIDE FROM THE TIME THE "SAMPLE" SUBROUTINE IS EXITED IN ONE UNIT OF TIME UNTIL IT IS TO BE CALLED AGAIN TO SAMPLE THE SIGNAL IN THE MIDDLE RANGE OF THE NEXT UNIT OF TIME.

IN A SITUATION SUCH AS THE ONE BEING DISCUSSED, IT IS OFTEN HELPFUL TO PRODUCE AN "EXPANDED TIMING DIAGRAM" TO ILLUSTRATE SMALLER PORTIONS OF "CRITICAL" TIME RELATIONSHIPS. AN EXPANDED DIAGRAM SHOWING THE INFORMATION JUST DERIVED AS IT APPLIES TO THE "START" BIT AND THE 1'ST "DATA" BIT OF THE EXAMPLE INCOMING SIGNAL IS SHOWN BELOW.



EXPANDED TIMING DIAGRAM

WITH THE TIMING REQUIREMENTS OF THE "SAMPLE" SUBROUTINE KNOWN, THE APPROPRIATE DELAYS TO PLACE THE "SAMPLING" SUBROUTINE SUCH THAT THE 3'RD SAMPLE IS AT THE MIDDLE OF A "UNIT OF TIME" CAN BE ASCERTAINED AS SHOWN ON THE ABOVE EXPANDED DIAGRAM. IT IS THEN A RELATIVELY EASY MATTER TO MODIFY THE PROGRAM PREVIOUSLY DEVELOPED FOR THE CASE WHEN ONLY A SINGLE SAMPLE WAS TAKEN PER TIME UNIT SO THAT IT "CALLS" THE "SAMPLE" SUBROUTINE. AN EXAMPLE OF SUCH A ROUTINE IS PRESENTED NEXT.

```
                    MNEMONIC                    COMMENTS
                    --------                    --------

          BDIN,     LBI 000       /CLEAR INCOMING FORMING & STORAGE REGISTER
                    LCI 005       /SET BIT COUNTER
32        STRTIN,   INP X         /LOOK FOR "START" BIT
32                  NDI 200       /MASK OFF IRRELEVANT DATA
44 / 36             JTS STRTIN    /IF NO START BIT, FORM "SAMPLING LOOP"
44 + 9184           CAL HDELAY    /IF FIND LOGIC "0" ASSUME START, DELAY
44 + 1528           CAL SAMPLE    /AND THEN DO MULTIPLE SAMPLE ON START BIT
36 / 44             JTS STRTIN    /IF RESULT NOT "0" ASSUME FALSE START
44 + 20             CAL DUMMY     /ADD COMPENSATING DELAY BEFORE ENTERING
20                  NDA           /MAIN "DATA" SAMPLING ROUTINE
20                  NDA           /WITH THESE THREE INSTRUCTIONS
44+18240  MORBDI,   CAL IDELAY    /EXECUTE MAIN DELAY LOOP
44 + 1528           CAL SAMPLE    /MULTIPLE SAMPLE ROUTINE ON "DATA" BITS
20                  RAL           /SAVE RESULTING STATE IN CARRY FLAG
20                  LAB           /GET ANY PREVIOUS BITS
20                  RAR           /ROTATE NEW BIT FROM CARRY INTO ACC
20                  LBA           /SAVE FORMATION IN REGISTER "B"
20                  DCC           /DECREMENT BITS COUNTER
44 / 36             JFZ MORBDI    /DELAY & THEN FETCH NEXT "DATA" BIT
20                  RRC           /HAVE ALL 5 "DATA" BITS - RIGHT JUSTIFY
20                  RRC           /IN ACCUMULATOR BY ROTATES
20                  RRC           /BEFORE PREPARING TO EXIT
44 + 9184           CAL HDELAY    /OPTIONAL DELAY TO REACH "STOP" AREA
20                  RET           /EXIT BAUDOT INPUT ROUTINE

32        IDELAY,   LDI 202       /SET TIME LOOP COUNTER
20                  NDA           /TRIM TIME DELAY
20                  NDA           /TRIM TIME DELAY
44 + 20   RDELAY,   CAL DUMMY     /TIME CONSUMING LOOP
20                  DCD           /DECREMENT COUNTER
12 / 20             RTZ           /EXIT TO CALLING RTN WHEN CNTR = 0
44                  JMP RDELAY    /OTHERWISE CONTINUE USING UP TIME

32        HDELAY,   LDI 101       /SET TIME LOOP COUNTER
20                  NDA           /TRIM TIME DELAY
20                  NDA           /TRIM TIME DELAY
44                  JMP RDELAY    /GO USE UP MORE TIME

20        DUMMY,    RET           /SHORT RTN TO USE UP TIME
```

THE INFORMATION PRESENTED TO THIS POINT IN THE CHAPTER HAS BEEN CONCERNED WITH ILLUSTRATING TECHNIQUES TO COORDINATE THE EXECUTION OF A PROGRAM WITH THE TIMING REQUIREMENT OF AN EXTERNAL DEVICE, THROUGH THE METHOD OF PROVIDING TIME DELAYS, TO EFFECTIVELY "SLOW DOWN" THE EXECUTION OF A PROGRAM. HOWEVER, ANOTHER ASPECT OF REAL-TIME PROGRAMMING INVOLVES ESSENTIALLY THE OPPOSITE OBJECTIVE. THAT IS TO OBTAIN MAXIMUM SPEED OF OPERATION FROM A COMPUTER PROGRAM SO THAT IT MAY HANDLE EVENTS THAT MIGHT BE OCCURING QUITE RAPIDLY. THE BALANCE OF THIS CHAPTER WILL PRESENT SEVERAL BASIC GUIDE LINES FOR "STREAMLINING" THE OPERATION OF A PROGRAM TO OBTAIN MAXIMUM SPEED OF EXECUTION.

PERHAPS THE FIRST POINT TO PRESENT IS THAT THERE IS A COROLLARY BETWEEN OBTAINING MAXIMUM OPERATING SPEED AND THE AMOUNT OF MEMORY REQUIRED BY THE PROGRAM THAT MAY AT FIRST SEEM A LITTLE STRANGE. THAT IS, AS ONE ATTEMPTS TO PROGRAM AN 8008 SYSTEM TO EXECUTE A PROGRAM THAT WILL PERFORM A FUNCTION IN A MINIMUM AMOUNT OF TIME, ONE GENERALLY WILL INCREASE THE AMOUNT OF MEMORY NEEDED TO STORE THE OPERATING PROGRAM. THE

REASON FOR THIS RELATIONSHIP IS THAT STREAMLINING A PROGRAM GENERALLY REQUIRES THE ELIMINATION OR REDUCTION IN THE USE OF "LOOPS" AND SUBROUTINES, WHICH, THE READER MAY RECALL, WERE EARLIER STRESSED FOR THEIR ABILITY TO SAVE MEMORY STORAGE SPACE!

TO ILLUSTRATE HOW THE ELIMINATION OF "LOOPS" CAN DRAMATICALLY REDUCE THE TIME REQUIRED TO EXECUTE A SPECIFIC FUNCTION, CONSIDER THE EXAMPLE PRESENTED NEXT. IN THIS CASE, A PROGRAMMER NEEDS TO LOAD THREE CONSECUTIVE WORDS IN MEMORY WITH THE CONTENTS OF THE ACCUMULATOR IN AS LITTLE TIME AS POSSIBLE. A ROUTINE USING A "LOOP" MIGHT BE AS SHOWN HERE:

```
32            LBI 003
28    AGAIN,  LMA
20            INL
20            DCB
44/36         JFZ AGAIN
```

THE READER MAY EASILY CALCULATE THAT THE TOTAL TIME REQUIRED TO EXECUTE THE ABOVE LOOP WOULD BE 360 MICROSECONDS. A ROUTINE THAT DID NOT USE A LOOP COULD BE EXECUTED IN ABOUT 1/3 THE TIME IN THIS PARTICULAR CASE AS ILLUSTRATED NEXT:

```
28            LMA
20            INL
28            LMA
20            INL
28            LMA
```

THE "STRAIGHT" ROUTINE ONLY REQUIRES 124 MICROSECONDS TO DO THE SAME JOB. WHILE THE COROLLARY MENTIONED ABOVE MIGHT NOT SEEM EVIDENT WHEN SUCH A SHORT LOOP IS INVOLVED, CONSIDER THE SAME CASE IF 20 LOCATIONS IN MEMORY WERE TO BE LOADED WITH THE DATA IN THE ACCUMULATOR. ONE CAN CALCULATE THAT THE LOOP METHOD WOULD ONLY REQUIRE 8 (DECIMAL) LOCATIONS IN MEMORY FOR THE OPERATING PORTION OF THE PROGRAM AND WOULD EXECUTE THE PROGRAM IN 2264 MICROSECONDS. ON THE OTHER HAND, THE "STRAIGHT" ROUTINE METHOD WOULD REQUIRE SOME 39 LOCATIONS IN MEMORY FOR STORAGE OF THE OPERATING PROGRAM, BUT THAT "STRAIGHT" ROUTINE WOULD BE EXECUTED IN A MERE 940 MICROSECONDS.

THE ELIMINATION OF SUBROUTINES CAN ALSO GREATLY SPEED UP THE OPERATION OF A CRITICAL PORTION OF A PROGRAM AS SHOWN BY THE FOLLOWING EXAMPLE. THE FOLLOWING "SUBROUTINE" METHOD MIGHT BE USED AS PART OF A PROGRAM THAT WAS TO RAPIDLY OUTPUT THE CONTENTS OF THE ACCUMULATOR AS A SERIES OF OCTAL DIGITS. I.E., THE OUTPUT DEVICE WOULD ONLY RECEIVE THE THREE LEAST SIGNIFICANT BITS IN THE ACCUMULATOR.

```
24            OUT X
44 + 80       CAL ROTAND
24            OUT X
44 + 80       CAL ROTAND
24            OUT X
16            HLT
```

WHERE THE SUBROUTINE "ROTAND" APPEARS AS:

```
20    ROTAND, RAR
20            RAR
20            RAR
20            RET
```

ONE CAN CALCULATE THAT EXECUTING THE ABOVE "SUBROUTINED" PROGRAM WOULD

REQUIRE 336 MICROSECONDS. THE "STRAIGHT" PROGRAM METHOD SHOWN BELOW ONLY REQUIRES 208 MICROSECONDS TO DO THE SAME FUNCTION.

```
24          OUT X
20          RAR
20          RAR
20          RAR
24          OUT X
20          RAR
20          RAR
20          RAR
24          OUT X
16          HLT
```

WHILE THE ABOVE EXAMPLE DOES NOT SUPPORT THE "MEMORY USAGE CORALLARY" ONE CAN SEE THAT IF THE SUBROUTINE WERE SOMEWHAT LONGER - SAY IT CONTAINED EIGHT OR NINE INSTRUCTIONS, THAT THE CORALLARY WOULD BE TRUE.

ANOTHER RULE OF THUMB TO APPLY TOWARDS DEVELOPING PROGRAMS TO OPERATE IN A MINIMUM AMOUNT OF TIME IS TO DO AS MUCH WORK AS POSSIBLE WITH CPU REGISTERS INSTEAD OF WITH MEMORY. FOR INSTANCE, SUPPOSE ONE HAD AN INSTRUMENT INTERFACED TO A 8008 SYSTEM THAT PERIODICALLY NEEDED TO SEND A SHORT "BURST" OF DATA TO THE COMPUTER FOR STORAGE. FOR TECHNICAL CONSIDERATIONS ASSUME THAT IT WAS DESIRED TO RECEIVE THE "BURST" AS RAPIDLY AS POSSIBLE, AFTER WHICH THE COMPUTER WOULD HAVE SOME "IDLE" TIME TO PROCESS THE DATA. ONE CAN READILY SEE BY THE FOLLOWING EXAMPLE THAT IT WILL TAKE MUCH LESS TIME TO STORE, SAY FOUR "CHARACTERS" IN CPU REGISTERS, THAN TO STORE THE SAME AMOUNT DIRECTLY IN MEMORY. A ROUTINE TO STORE THE CHARACTERS DIRECTLY IN MEMORY WOULD REQUIRE:

```
32          INP X
28          LMA
20          INL
32          INP X
28          LMA
20          INL
32          INP X
28          LMA
20          INL
32          INP X
28          LMA
```

OR A TOTAL OF 300 MICROSECONDS. STORING THE DATA IN CPU REGISTERS WOULD ONLY REQUIRE 216 MICROSECONDS USING THE FOLLOWING ROUTINE.

```
32          INP X
20          LBA
32          INP X
20          LCA
32          INP X
20          LDA
32          INP X
20          LEA
```

THE FACTOR THAT MIGHT BE PARTICULARLY VALUABLE IN A "TIME-TIGHT" APPLICATION IS THAT EACH CHARACTER IN THE SECOND ROUTINE COULD BE ACCEPTED AT 52 MICROSECOND INTERVALS WHILE THE FIRST ROUTINE COULD NOT ACCEPT THE CHARACTERS AT A RATE FASTER THAN EVERY 80 MICROSECONDS. NATURALLY, THE ABOVE EXAMPLE IS STRICTLY LIMITED TO THE CASE WHERE VERY SHORT "BURSTS" ARE BEING HANDLED AS THERE ARE A LIMITED NUMBER OF CPU REGISTERS AVAILABLE IN WHICH TO STORE DATA. HOWEVER, THE PRINCIPLE CAN BE VALUABLE.

THE CONCEPT OF UTILIZING CPU REGISTERS AS MUCH AS POSSIBLE CAN BE EXTENDED TO A VARIETY OF APPLICATIONS BESIDES THE ONE ILLUSTRATED ABOVE. FOR INSTANCE, IT IS OFTEN ADVANTAGEOUS TO SET UP CPU REGISTERS IN ADVANCE OF A "CRITICAL" TIME PERIOD IN ORDER TO STREAMLINE A PROGRAM DURING SELECTED OPERATING PERIODS. FOR INSTANCE, SUPPOSE ONE NEEDED TO INPUT DATA AT A FAST RATE AND ALSO PERFORM SOME MANIPULATION OF THE DATA. SUCH AS, PERFORM A TWO'S COMPLEMENT OPERATION ON THE DATA AND THEN DEPOSIT THE DATA IN MEMORY. ONE WAY TO DEVELOP THE ROUTINE WOULD BE AS FOLLOWS:

```
32     RECEIV, INP X
32             NDI 377
32             ADI 001
28             LMA
20             INL
44/36          JFZ RECEIV
```

THE ABOVE ROUTINE COULD HAVE THE TIME FACTOR DECREASED BY ABOUT 12 PERCENT IF, PRIOR TO ENTERING THE "LOOP" (A NECESSARY EVIL IN THIS EXAMPLE BECAUSE A "LARGE" BLOCK OF DATA IS HYPOTHETICALLY BEING PROCESSED) ONE FIRST SET CPU REGISTER "B" TO CONTAIN "377" AND CPU REGISTER "C" TO HOLD "001," AND USED THE ROUTINE SHOWN NEXT.

```
32     RECEIV, INP X
20             NDB
20             ADC
28             LMA
20             INL
44/36          JFZ RECEIV
```

A FEW CLOSING COMMENTS ON THE SUBJECT OF "STREAMLINING" REAL-TIME PROGRAMS WOULD INCLUDE THE MENTION THAT IF "SUBROUTINES" ARE NECESSARY, TO USE THOSE VALUABLE "RESTART" COMMANDS WHICH ONLY REQUIRE 20 MICROSECONDS FOR AN EFFECTIVE "CALL" INSTEAD OF 44 MICROSECONDS. ADDITIONALLY, THE PROGRAMMER SHOULD PAY STRICT ATTENTION TO OVER-ALL PROGRAM ORGANIZATION IN ORDER TO REDUCE TIME CONSUMING "OVERHEAD" OPERATIONS, OR AT LEAST TO DEFER SUCH OPERATIONS FOR EXECUTION DURING NON-CRITICAL TIME PERIODS.

FINALLY, REAL-TIME PROGRAMMING IS AN AREA WHERE THE CREATIVE PROGRAMMER CAN HAVE A LOT OF FUN. EXPERIMENT, LOOK FOR NEW METHODS TO SOLVE A PARTICULAR PROBLEM - YOU MAY FIND A BETTER, FASTER WAY! SUCH AS:

HAVE THE FIRST INSTRUCTION OF THE ABOVE ROUTINE LOCATED AT THE ADDRESS OF RESTART LOCATION "X," MODIFY THE ROUTINE AS ILLUSTRATED, AND CUT ANOTHER 7 PERCENT OFF THE REQUIRED EXECUTION TIME OF THE ROUTINE!

```
32             INP "X"
20             NDB
20             ADC
28             LMA
20             INL
12/20          RTZ
20             RST "X"
```

'PROM' PROGRAMMING CONSIDERATIONS

FOR READERS WHO MAY NOT BE FAMILIAR WITH THE ABBREVIATION, A "PROM" IS A "PROGRAMMABLE READ-ONLY MEMORY" ELEMENT. A PROGRAMMABLE READ-ONLY MEMORY ELEMENT IS AN ELECTRONIC DEVICE THAT CAN BE "PROGRAMMED" WITH A PROGRAM USING A SPECIAL INSTRUMENT SO THAT IT CONTAINS A "PERMANENT" PROGRAM. SOME "PROM" ELEMENTS CAN BE "ERASED" AND RE-PROGRAMMED BY USING SPECIAL INSTRUMENTS WHICH ARE GENERALLY TOO EXPENSIVE FOR THE AVERAGE USER TO HAVE READILY AVAILABLE. WHEN THE "PROGRAMS" IN SUCH ELEMENTS NEED TO BE CHANGED IT IS GENERALLY NECESSARY TO SEND THE DEVICE BACK TO THE MANUFACTURER OR REPRESENTATIVE FOR PROCESSING.

THE KEY FEATURE THAT A "READ-ONLY MEMORY" ELEMENT HAS OVER A "RAM" (READ AND WRITE MEMORY) DEVICE IS THAT ONCE A PROGRAM HAS BEEN PLACED IN A "ROM" IT IS NON-VOLATILE, OR PERMANENT. A SEMI-CONDUCTOR "RAM" DEVICE WILL LOSE IT'S CONTENTS IF POWER IS REMOVED FROM THE DEVICE. A "ROM" WILL RETAIN THE INFORMATION PLACED IN IT IF POWER IS REMOVED. THUS, THE "ROM" IS AN IDEAL MEMORY DEVICE IN WHICH TO STORE PROGRAMS THAT ARE PERMANENT IN NATURE OR THAT HAVE FREQUENT USE IN A SYSTEM WHERE POWER MAY FREQUENTLY BE REMOVED. IT ELIMINATES THE PROCESS OF HAVING TO "LOAD" PROGRAMS BACK INTO MEMORY WHEN A COMPUTER SYSTEM IS INITIALLY "POWERED-UP" FOR A PERIOD OF OPERATION.

THE KEY DISADVANTAGE OF THE "ROM" IS THAT THE COMPUTER CANNOT ALTER THE CONTENTS OF THOSE MEMORY LOCATIONS ASSIGNED TO A "ROM" DEVICE. THUS ONE MUST TAKE SPECIAL PRECAUTIONS WHEN DESIGNING PROGRAMS THAT ARE TO RESIDE IN A "ROM" DEVICE.

FOR INSTANCE, ONE CANNOT USE MEMORY ADDRESSES IN A ROM TO STORE TEMPORARY POINTERS AND COUNTERS FOR A PROGRAM THAT NEEDS TO ALTER SUCH POINTERS AND COUNTERS DURING THE PROGRAM'S OPERATION - AND SIMILARLY ONE CANNOT USE ANY SUCH LOCATIONS FOR ANY KIND OF TEMPORARY STORAGE OF DATA OR OTHER "TEMPORARY" INFORMATION, BECAUSE, AS JUST MENTIONED, THE COMPUTER WILL NOT BE ABLE TO "WRITE" THE INFORMATION INTO THE ROM!

THUS, IF A PROGRAM IS TO BE STORED IN A ROM, AND IT IS NECESSARY TO USE POINTERS AND COUNTERS IN A PROGRAM (AS WILL CERTAINLY BE THE CASE IN MANY APPLICATIONS) ONE SHOULD ARRANGE THE PROGRAM TO USE CPU REGISTERS FOR THOSE PURPOSES, OR TO USE ADDRESSES IN MEMORY THAT WILL CONTAIN RAM ELEMENTS.

A ROM ELEMENT CAN BE CONSIDERED AS A "HARDWARE" MEMORY ELEMENT AND AS SUCH, ONE OF THE FIRST MATTERS ONE SHOULD CONSIDER WHEN PLANNING ON INSTALLING ROMS IN A COMPUTER SYSTEM, IS WHERE TO ASSIGN THE ROM ELEMENTS IN MEMORY. A GOOD RULE OF THUMB IS TO PLACE SUCH ELEMENTS AT THE UPPER EXTREME ADDRESSES AVAILABLE IN THE SYSTEM. FOR INSTANCE, IF ONE HAS AN 8008 SYSTEM CAPABLE OF ADDRESSING UP TO 4 K OF MEMORY, (PAGES 00 THROUGH 17) IT WOULD BE ADVISABLE IN MOST CASES TO DEVELOP PROGRAMS FOR ROM(S) THAT ARE ON PAGE 17, OR IF MORE PAGES ARE REQUIRED FOR ROMS, TO WORK DOWNWARD FROM THAT ADDRESS. (MOST ROM AND PROM DEVICES CAN CONTAIN 256 EIGHT BIT WORDS - OR ONE "PAGE" IN A TYPICAL 8008 SYSTEM.) THIS ALLOWS ALL ADDRESSES BELOW THE ROM ELEMENT(S) TO BE AVAILABLE AS ONE CONTINUOUS BLOCK OF "READ AND WRITE" MEMORY WHICH IS GENERALLY A MORE CONVENIENT ARRANGEMENT THAN, SAY, STICKING A ROM ELEMENT ON PAGE 10 IN SUCH A SYSTEM, THUS DIVIDING THE AVAILABLE ADDRESSES FOR RAM MEMORY INTO TWO SEPARATE AREAS.

ALTERNATIVELY, ONE MIGHT WANT TO CONSIDER PLACING ROM ELEMENTS AT THE LOWEST AVAILABLE ADDRESSES FOR THE SYSTEM, AND LEAVING THE UPPER ADDRESSES AVAILABLE AS ONE CONTINUOUS BLOCK FOR RAM ELEMENTS. HOWEVER,

UNLESS A SYSTEM IS BEING DESIGNED TO SERVE AS A SPECIAL FUNCTION DEVICE, IT IS GENERALLY WISE TO NOT USE A ROM ON PAGE 00 IN AN 8008 SYSTEM AS IT WILL OCCUPY ALL THE POSSIBLE "RESTART" (RST) INSTRUCTION LOCATIONS! THE EXCEPTION TO THIS WOULD BE IF ONE DELIBERATELY WANTED TO HAVE "POWER-UP" ROUTINES THAT USED THE INTERRUPT FACILITY OF THE 8008 SYSTEM IN CONJUNCTION WITH A ROM TO AUTOMATICALLY GO TO A "RESTART" LOCATION. THE "RST" CLASS OF INSTRUCTIONS, WHICH USE THE SPECIAL LOCATIONS ON PAGE 00, ARE PARTICULARLY USEFUL COMMANDS WITH GENERAL PURPOSE APPLICATIONS, AS DISCUSSED ELSEWHERE IN THIS MANUAL, AND ONE SHOULD CONSIDER THEIR GENERAL PURPOSE CAPABILITIES CAREFULLY BEFORE DECIDING TO RESTRICT THEM TO A ROM APPLICATION.

THE TYPES OF PROGRAMS THAT ARE GENERALLY MOST SUITABLE FOR PLACEMENT ON ROMS INCLUDE ROUTINES TO ASSIST GETTING A SYSTEM "ON-LINE" IMMEDIATELY FOLLOWING POWER TURN-ON, SUCH AS I/O ROUTINES AND "PROGRAM LOADERS." FREQUENTLY UTILIZED PROGRAMS THAT ONE MAY NOT WANT TO HAVE TO BE BOTHERED LOADING EACH TIME A SYSTEM IS STARTED, OR PROGRAMS FOR DEDICATED APPLICATIONS.

FOR INSTANCE, A USER WITH A TELETYPE SYSTEM MIGHT WANT TO PUT A STANDARD ROUTINE TO INPUT AND OUTPUT INFORMATION TO THE DEVICE (WHICH COULD BE CALLED BY GENERAL ROUTINES) AND POSSIBLY A "LOADER PROGRAM" THAT WOULD ENABLE THE USER TO QUICKLY LOAD PROGRAMS INTO RAM MEMORY VIA A PAPER TAPE READER. IN SUCH AN APPLICATION, ONE MIGHT ALSO HAVE SPACE ON A PROM TO INCLUDE A SIMPLE PROGRAM THAT WOULD ENABLE ONE TO EXAMINE AND MODIFY MEMORY LOCATIONS USING THE TELETYPE DEVICE. THUS, WHENEVER POWER WAS APPLIED TO THE COMPUTER SYSTEM, ONE WOULD INSTANTLY BE IN A POSITION TO "LOAD" LARGER PROGRAMS INTO RAM MEMORY, OR TO IMMEDIATELY USE THE TELETYPE TO PLACE INFORMATION INTO RAM MEMORY. WITHOUT A ROM, THE USER WOULD HAVE TO USE MANUAL CONTROL METHODS TO "LOAD" A "LOADER" PROGRAM OR OTHER ROUTINES INTO MEMORY. THE SAVINGS IN TIME ONE CAN ACHIEVE BY USING A ROM TO STORE "START-UP" PROGRAMS OVER HAVING TO USE PURELY MANUAL PROCEDURES CAN BE WELL WORTH THE COST OF A ROM OR PROM DEVICE.

HOWEVER, A USER WHO DESIRED TO DEVELOP SUCH A PACKAGE FOR STORAGE ON A ROM DEVICE WOULD HAVE TO BE PARTICULARLY CAREFUL WHEN DEVELOPING THE TELETYPE I/O ROUTINE IF SUCH A ROUTINE REQUIRED "REAL-TIME PROGRAMMING" CONSIDERATIONS, SUCH AS A "TIMING LOOP." FOR INSTANCE, THE READER WHO HAS READ THE PREVIOUS CHAPTER WILL REALIZE THAT IF THE COMPUTER PROGRAM ITSELF WILL CONTROL THE ACTUAL OPERATION OF A DEVICE SUCH AS A TELETYPE MACHINE, AND "TIMING LOOPS" ARE ESTABLISHED TO CONTROL THE PRECISE TIME AT WHICH EVENTS WILL OCCUR, THAT THE ACTUAL TIMING REQUIRED TO PROPERLY OPERATE A DEVICE WILL BE A FUNCTION OF THE DEVICE BEING CONTROLLED AS WELL AS THE TIMING IN THE COMPUTER ITSELF, AND THAT THE ACCURACY AT WHICH SUCH TIMING MUST BE MAINTAINED IS A FUNCTION OF THE ACCURACY OF THE TIMING IN THE COMPUTER SYSTEM AND THE DEVICE ITSELF. THIS ACCURACY MAY VARY BETWEEN DIFFERENT UNITS. IF A FIXED "TIMING LOOP" WAS PROGRAMMED INTO A "PROM" AND AT SOME LATER DATE THE EXTERNAL DEVICE WAS REPLACED WITH A DIFFERENT ONE, OR THE TIMING OF THE COMPUTER WAS ADJUSTED, THE ORIGINAL "TIMING LOOP" MIGHT BE MADE INVALID. THUS, IN SUCH AN APPLICATION, IT MIGHT BE WISE TO PLACE THE ACTUAL "DATA" VALUE THAT IS TO CONTROL THE "TIMING LOOP" IN A "RAM" LOCATION AND HAVE THE PROGRAM IN THE PROM ACCESS THAT VALUE, WHICH WOULD BE MANUALLY INSERTED BY THE OPERATOR, RATHER THAN HAVING THE VALUE BE "FIXED" IN THE PROM. THE FOLLOWING TWO SUBROUTINES WILL HELP CLARIFY THE POINT.

PROM PROGRAM WITH A "FIXED" TIMING LOOP VALUE

```
TIME,   LDI 100      /SET TIMING LOOP COUNTER
```

```
TIMER,  CAL DUMMY    /DELAY SUBROUTINE
        DCD          /DECREMENT TIMING LOOP COUNTER
        RTZ          /EXIT SUBROUTINE WHEN TIME DELAY DONE
        JMP TIMER    /OTHERWISE CONTINUE TIMING LOOP
```

PROM PROGRAM WITH CAPABILITY TO ALTER TIMING LOOP VALUE

```
TIME,   LHI XXX      /SET POINTER TO "RAM" LOCATION WHERE
        LLI YYY      /TIMING LOOP COUNTER VALUE STORED
        LDM          /SET TIMING LOOP COUNTER VALUE
TIMER,  ...          /SAME AS ABOVE ROUTINE
```

THE SECOND ROUTINE ILLUSTRATED ABOVE ASSUMES THAT THE CPU MEMORY POINTER REGISTERS WILL BE SET UP TO POINT TO A LOCATION IN RAM MEMORY WHERE THE ACTUAL "LOOP COUNTER" VALUE WILL HAVE BEEN PLACED BY THE OPERATOR. WHILE THE METHOD NECESSITATES THE OPERATOR HAVING TO SET THE PROPER VALUE INTO RAM MEMORY BEFORE USING THE PROGRAM STORED ON THE ROM, IT AVOIDS THE PROBLEM OF HAVING A "USELESS" PROGRAM IN THE PROM IF A TIMING VALUE MUST BE ALTERED AT SOME FUTURE DATE. IT SHOULD BE APPARENT THAT THIS KIND OF SCHEME CAN BE APPLIED TO ANY SIMILAR SITUATION WHERE A "VALUE" USED BY A PROGRAM MIGHT CONCEIVABLY NEED TO BE ALTERED.

IF, FOR SOME REASON, ONE DID NOT WANT TO HAVE TO DEDICATE A LOCATION IN RAM MEMORY FOR A "VARIABLE" VALUE IN SUCH A ROUTINE - THERE IS STILL ANOTHER TRICK THAT CAN "SAVE" THE DAY IN SUCH A SITUATION. THE OPERATOR COULD MANUALLY LOAD THE "D" REGISTER IN THE CPU PRIOR TO USING THE ABOVE TYPE OF SUBROUTINE (OR HAVE AN EXTERNAL ROUTINE IN RAM MEMORY PERFORM THE SAME FUNCTION BEFORE USING THE ROUTINE), IN WHICH CASE ONE COULD ELIMINATE THE PORTION OF THE ABOVE ROUTINE LABELED "TIME" AND SIMPLY USE THAT PORTION LABELED "TIMER."

A GOOD RULE OF THUMB TO APPLY WHEN CONSIDERING THE USE OF ROM IN A SYSTEM IS TO TAILOR THE PROGRAM FOR COMPACTNESS. AFTER ALL, THE MORE ROUTINES OR SUBROUTINES ONE CAN STORE ON A PROM, THE MORE USEFUL THE DEVICE WILL BE. MAKE EVERY EFFORT TO SAVE MEMORY SPACE BY JUDICIOUS USE OF SUBROUTINING, WITH MULTIPLE ENTRY POINTS IF APPLICABLE, AND BY USE OF PROGRAM LOOPS. AN EARLIER CHAPTER STRESSED THE CONCEPT AND PROVIDED GUIDELINES AND FORMULAS FOR CALCULATING WHEN SUCH TECHNIQUES ARE APPLICABLE. ONE SHOULD FIGURE ON SPENDING SOME EXTRA TIME WHEN DEVELOPING PROGRAMS TO BE STORED ON ROMS IN ORDER TO LOOK AT WAYS TO SAVE MEMORY SPACE. TRY TO USE EVERY AVAILABLE LOCATION ON A PROM - AFTER ALL, ANY UNUSED LOCATIONS WILL BE "PERMANENTLY" WASTED. IF ONE FINDS ONE HAS SOME ROOM LEFT IN A PROM AFTER ONE HAS PLACED THE PROGRAMS REQUIRED TO BE ON THE DEVICE FOR A PARTICULAR APPLICATION, CONSIDER THE POSSIBILITY OF "TUCKING IN" A FEW SMALL ROUTINES THAT WOULD HAVE GENERAL USEFULNESS. SUCH ROUTINES AS "SWITCH," "ADV," AND "CNTDWN" WHICH WERE PRESENTED AND USED FREQUENTLY IN EXAMPLES THROUGH-OUT THIS MANUAL ARE TYPICAL KINDS OF GENERALLY USEFUL SUBROUTINES THAT ONE MIGHT CONSIDER HAVING ON A ROM RATHER THAN "WASTING" ANY LOCATIONS. THESE TYPES OF ROUTINES WOULD THEN ALWAYS BE AVAILABLE IN THE SYSTEM FOR USE BY PROGRAMS RESIDING IN RAM.

ABOVE ALL, HOWEVER, ONCE ONE HAS DEVELOPED ROUTINES FOR A PROM, ONE SHOULD THOROUGHLY TEST AND CHECK THE PROGRAM(S) TO MAKE SURE THEY ARE ABSOLUTELY OPERATING AS INTENDED. AFTER ALL, IT IS A BIT COSTLY TO MAKE A "PROGRAM PATCH" ON A READ-ONLY MEMORY ELEMENT!

CREATIVE PROGRAMMING CONCEPTS

ONCE ONE HAS BECOME FAMILIAR WITH THE FUNDAMENTAL ASPECTS OF MACHINE LANGUAGE PROGRAMMING. ONCE ONE IS FAMILIAR WITH THE MNEMONICS THAT REPRESENT THE MACHINE LANGUAGE COMMANDS AND CAN MENTALLY THINK OF THE FUNCTIONS THAT THOSE MNEMONICS REPRESENT. ONCE ONE HAS LEARNED HOW TO FORMALIZE AND PLAN OUT A PROGRAM, UNDERSTANDS FLOW CHARTING, AND MEMORY ALLOCATION OR MAPPING. ONCE ONE HAS HAD SOME PRACTICE AT DEVELOPING ALGORITHMS AND COMBINING SMALLER ALGORITHMS INTO FULL SIZED PROGRAMS BY SUBROUTINING. ONCE ONE IS FAMILIAR WITH SETTING UP POINTERS, COUNTERS, FORMING PROGRAM LOOPS, UTILIZING BIT "MASKS." ONCE ONE HAS A "FEEL" FOR ORGANIZING DATA FOR TABLES, AND UNDERSTANDS HOW DATA CAN BE SORTED. ONCE ONE UNDERSTANDS HOW MATHEMATICAL INFORMATION MAY BE PROCESSED BY THE COMPUTER. AND, ONCE ONE KNOWS HOW TO GET DATA INTO AND OUT OF THE CPU FROM AND TO SOME EXTERNAL DEVICES. I.E., ONCE ONE HAS SPENT A LITTLE TIME STUDYING THE ASPECTS OF MACHINE LANGUAGE PROGRAMMING A COMPUTER - AS ONE WILL HAVE DONE BY READING (AND HOPEFULLY LEARNING!) THE INFORMATION PRESENTED IN THE PRECEEDING SECTIONS OF THIS MANUAL. THEN, ONE SHOULD BE IN A POSITION TO UNDERSTAND AND APPRECIATE THE TRUE POTENTIAL OF A DIGITAL COMPUTER WHEN IT'S POWER IS UNLEASHED UNDER THE AUSPICES OF A CREATIVE PROGRAMMER. THEN, IS WHEN ONE CAN REALLY START HAVING FUN CREATING AND DEVELOPING COMPLETELY ORIGINAL PROGRAMS TO PERFORM MYRIADS OF PERSONALLY DESIRED FUNCTIONS. THIS IS THE POINT AT WHICH ONE MAY TAKE A "BROAD VIEW" OF THE IMMENSE CAPABILITY OF THE MACHINE BY STANDING BACK AND PONDERING SOME "SCENES" MUCH THE WAY AN ARTIST WOULD PONDER A BLANK CANVAS BEFORE STARTING TO PAINT A "CONCEPT" OR "IMAGE" THAT EXISTED PURELY IN THE ARTIST'S MIND. THE DISCUSSION THAT FOLLOWS MERELY PRESENTS SOME WAYS IN WHICH TO VIEW THE CAPABILITY OF A DIGITAL COMPUTER. SOME POINTS OF VIEW THAT MAY HELP PROGRAMMER'S APPROACH PROGRAMMING TASKS WITH CREATIVITY. NO GREAT "MAGIC" IS CLAIMED FOR THE IDEAS PRESENTED, NO GUARANTEE IS MADE THAT THE POINTS OF VIEW WILL INSPIRE EVERYONE TO GREATER PROGRAMMING CREATIVITY OR ABILITY. BUT, IT IS KNOWN THAT THE VIEWS PRESENTED HAVE HELPED AT LEAST ONE PROGRAMMER TO CREATE COUNTLESS PROGRAMS, SOME OF WHICH OTHERS HAD CLAIMED "COULDN'T BE DONE ON A SMALL MACHINE," AND SOLVE NUMEROUS PROGRAMMING PROBLEMS, WHILE HAVING A LOT OF FUN - AND QUITE OFTEN SAVING A LOT OF TIME! THUS, THE IDEAS WILL BE PRESENTED IN THE HOPES THAT PERHAPS A FEW OTHERS WILL BENEFIT A LITTLE, OR A LOT.

IT MUST BE ADMITTED THAT TO SOME READERS THE CONCEPTS DISCUSSED IN THIS SECTION MIGHT SEEM "TRIVIAL" AT FIRST GLANCE. PERHAPS THE REASON SOME PEOPLE INITIALLY SEE THE CONCEPTS AS TRIVIAL IS BECAUSE THEY ARE PROFOUNDLY BROAD AND TO SOME LUCKY PEOPLE, PERHAPS, INSTINCTIVELY OBVIOUS. HOWEVER, MOST READERS WILL PROBABLY FIND THE CONCEPTS "GROW" AS ONE DOES MORE AND MORE PROGRAMMING, UNTIL ONE DAY, THE READER "DISCOVERS" A PROFOUNDLY "SIMPLE" WAY TO HANDLE A PROGRAMMING PROBLEM BASED ON A VARIATION OF ONE SORT OR ANOTHER OF THE CONCEPTS PRESENTED IN THIS SECTION.

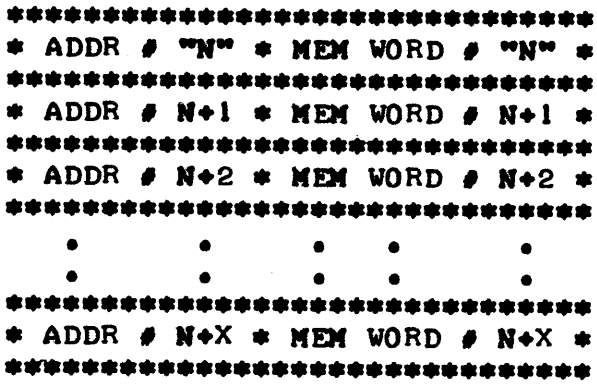
FOR WHAT THEY ARE WORTH, THE CONCEPTS TO BE PRESENTED WILL BE DISCUSSED IN THREE PARTS.

THE ONE DIMENSIONAL VIEW

THE UNDERLYING PRINCIPAL IN THIS ENTIRE DISCUSSION ON CREATIVE PROGRAMMING IS TO LEAVE OUT THE DETAILS OF THE OPERATION OF THE CPU AND IT'S ASSOCIATED REGISTERS. IT IS KNOWN THAT THE CPU AND THE ASSOCIATED

REGISTERS CAN DO A WHOLE HOST OF SPECIFIC OPERATIONS - MATHEMATICAL, BOOLEAN LOGIC, EXECUTE CONDITIONAL BRANCHES AND WHATEVER. THESE FUNCTIONS WILL BE TAKEN FOR GRANTED IN THE FOLLOWING DISCUSSION. WHAT IS IMPORTANT IN THE PRESENT SITUATION IS TO REALIZE THAT THE POWER OF THE COMPUTER IS IN IT'S MEMORY. THE CPU OBTAINS IT'S INSTRUCTIONS FROM MEMORY, AND THE CPU IS ABLE TO MANIPULATE INFORMATION IN MEMORY. THE CPU IS ABLE TO ACCESS A PARTICULAR WORD IN MEMORY, IN THE CASE OF AN 8008 SYSTEM, BY POINTING TO THE "ADDRESS" USING THE "H & L" REGISTERS. FOR EACH SPECIFIC "ADDRESS" THERE IS A "SPECIFIC WORD IN MEMORY" THAT CONTAINS EIGHT BINARY BITS.

ONE WAY TO VIEW THE ORGANIZATION OF MEMORY IS TO THINK OF MEMORY AS BEING ONE LONG LINE OF WORDS - STACKED ONE AFTER THE OTHER. IN FACT, THIS IS THE WAY VIRTUALLY ANY MACHINE LANGUAGE PROGRAMMER FIRST STARTS THINKING OF MEMORY BECAUSE OF THE SIMPLE WAY IN WHICH EACH MEMORY ADDRESS CORRESPONDS TO A WORD IN MEMORY - AND MEMORY ADDRESSES ARE SIMPLY A SERIES OF CONSECUTIVE NUMBERS.



THUS ONE CAN CONSIDER MEMORY AS SIMPLY BEING ONE LONG STRING OF LOCATIONS THAT MAY BE FILLED WITH WHATEVER INFORMATION IS DESIRED IN A SERIAL SEQUENCE. IF ONE WERE TO FILL EACH MEMORY WORD WITH A "CODE" THAT SYMBOLIZED A LETTER OR DIGIT, OR PUNCTUATION SYMBOL, ONE COULD PROCEED TO FILL A "STRING" OF MEMORY LOCATIONS WITH ENGLISH (OR FRENCH, OR GERMAN, OR WHATEVER) WORDS, AND GO ON TO FORM SENTENCES, AND BY USING OTHER CODES, TO SEPARATE SENTENCES INTO PARAGRAPHS.

N	O	W	SPACE	I	S
ADDR N	ADDR N+1	ADDR N+2	ADDR N+3	ADDR N+4	ADDR N+5

OR, ONE COULD PLACE MATHEMATICAL VALUES IN MEMORY LOCATIONS, SEPARATE THOSE VALUES BY "OPERATOR" SYMBOLS, AND PROCESS "COLUMNS" OF MATHEMATICAL DATA. (ASSUMING IN THIS STRICT CASE THAT THE VALUES WERE SMALL ENOUGH TO BE STORED IN ONE MEMORY WORD.)

```
ADDR  N  :   +100
ADDR N+1 : MINUS
ADDR N+2 :   - 50
ADDR N+3 : EQUAL
```

OR, THE CONTENTS OF MEMORY WORDS MAY BE USED TO SYMBOLIZE JUST ABOUT ANY ABSTRACT ITEM THAT THE PROGRAMMER MIGHT DESIRE. THE PROGRAMMER NEED

SIMPLY FORM A CODE THAT THE PROGRAMMER DESIRES TO HAVE SYMBOLIZE SOMETHING.

```
ADDR  N   :  SYMBOL FOR "APPLES"
ADDR N+1  :  SYMBOL FOR "PEARS"
ADDR N+2  :  SYMBOL FOR "BANANAS"
ADDR N+3  :  SYMBOL FOR "CHERRIES"
ADDR N+3  :  SYMBOL FOR "LEMONS"
ADDR N+4  :  SYMBOL FOR "BELLS"
```

THE READER SHOULD REALIZE HERE, THAT THE CONCEPT BEING PRESENTED IS CONCENTRATING ON HOW MEMORY IS UTILIZED FOR HANDLING "DATA" OR INFORMATION. IT IS TAKEN FOR GRANTED THAT A PORTION OF MEMORY WILL BE USED FOR THE ACTUAL OPERATING PROGRAM THAT "CONTROLS" THE MANIPULATION OF THE MEMORY THAT IS BEING USED FOR THE "DATA." THUS, IN THE ABOVE EXAMPLES ONE MUST REALIZE THAT AN "OPERATING PROGRAM" WILL PLACE THE CODES FOR LETTERS OR DIGITS, PUNCTUATION MARKS, SPACES, AND SO FORTH, AND PERFORM WHATEVER PROCESSING IS DESIRED. AN OPERATING PROGRAM WILL TAKE THE VALUES GIVEN IN THE MATHEMATICAL EXAMPLE AND "INTERPRET" THE SYMBOLS AND PERFORM THE DESIRED FUNCTIONS. AND, AN OPERATING PROGRAM IN THE THIRD EXAMPLE WOULD RECOGNIZE A PARTICULAR CODE TO MEAN "APPLES" AND PRINT OR DISPLAY THE ENTIRE WORD (OR PICTURE!) WHEN IT INTERPRETED THAT CODE. THE PRIMARY POINT BEING MADE IS THAT THE DATA IS ORGANIZED AS A LONG "LINE" OF INFORMATION. THAT LINE OF INFORMATION CAN BE ARBITRARILY SPLIT UP INTO MANY PARTS AND PIECES OF THE LINE BE CONSIDERED AS FORMING ONE PARTICULAR SECTION, AS IN THE CASE WHEN ONE "ENGLISH WORD" IS FORMED FROM A SERIES OF "LETTERS." THE LONG LINE IS SIMPLY FORMED, AND LOCATIONS ALONG THE LINE ARE MARKED, BY A "MEMORY ADDRESS."

HOWEVER, AND THIS THE CREATIVE PROGRAMMER SHOULD TAKE PARTICULAR NOTE OF, THE FACT THAT LOCATIONS ARE MARKED ALONG THE LINE BY "MEMORY ADDRESSES" CAN BE TRANSFORMED BY THE PROGRAMMER SO THAT MEMORY ADDRESSES ESSENTIALLY STAND FOR ANY ARBITRARILY ASSIGNED "MARKER." IN OTHER WORDS, TO THE PROGRAMMER, MEMORY ADDRESS NUMBER "N" CAN CORRESPOND TO TIME "T," OR DISTANCE "D," OR POINT "Z." THUS, ONE CAN STORE, SAY, THE VALUE OF THE AMPLITUDE OF A SIGNAL AT TIME "T" IN ONE LOCATION, THE VALUE AT TIME T + T' IN THE NEXT LOCATION, THE VALUE AT TIME T + 2T' IN THE NEXT LOCATION. FURTHERMORE, IT SHOULD BE APPARENT THAT T' CAN BE "SCALED" AS DESIRED BY APPROPRIATE PROGRAMMING SO THAT T' REPRESENTS ONE MICROSECOND, OR MILLISECOND, OR SECOND, OR A YEAR!

FURTHERMORE, ONE CAN ACTUALLY GO BEYOND THE POINT OF CONSIDERING THE LOCATIONS TO BE A LONG STRAIGHT LINE, BY CONSIDERING THE POSSIBILITY OF MANIPULATING THE LINE OF LOCATIONS AS A PIECE OF STRING. ONE CAN FIGURATIVELY "CUT" THE PIECE OF "STRING" AT ANY DESIRED LOCATION AND FORM THE "STRING" INTO A "RING" OR "CIRCLE." THIS IS EASILY ACCOMPLISHED BY SIMPLY HAVING THE "MEMORY ADDRESS POINTER" GO BACK TO LOCATION "N" WHEN IT REACHES LOCATION "N + X." CONSIDER THE POSSIBILITY OF DOING SUCH AN OPERATION WITH THREE SECTIONS OF THE LINE AND USING THE TECHNIQUE TO SIMULATE A "ONE ARMED BANDIT" MACHINE:

ADDR N	APPLE	ADDR N+X+1	PEAR	ADDR N+2X+1	BANANA
ADDR N+1	PEAR	ADDR N+X+2	BANANA	ADDR N+2X+2	LEMON
ADDR N+2	CHERRY	ADDR B+X+3	LEMON	ADDR N+2X+3	APPLE
ADDR N+3	BANANA	ADDR N+X+4	BELL	ADDR N+2X+4	BELL
ADDR N+4	LEMON	ADDR N+X+5	CHERRY	ADDR N+2X+5	PEAR
ADDR N+X	BELL	ADDR N+X+6	APPLE	ADDR N+2X+6	CHERRY

ONE COULD DEVELOP ALGORITHMS TO "SPIN" THE MEMORY POINTER AROUND EACH "RING" AND RANDOMLY COME TO A STOP AT A LOCATION WITHIN EACH RING. THE RESULTS OF THE EVENTS IN ALL THREE "RINGS" COULD THEN BE PROCESSED TO DETERMINE WHETHER ONE "HIT A JACKPOT" OR MISSED. THE DETAILS OF SUCH A PROGRAM WILL BE LEFT TO THE CREATIVE PROGRAMMER, BUT THE CONCEPT OF HOW ONE COULD APPROACH SUCH A SIMULATION PROJECT IS HOPEFULLY CLEAR.

FINALLY, TO TAKE THE "ONE DIMENSION" VIEW A LITTLE FURTHER, ONE CAN GO DOWN TO THE "BIT" LEVEL. SINCE A MEMORY WORD IN AN 8008 SYSTEM ACTUALLY CONSIST OF 8 INDIVIDUAL "BITS," ONE COULD CONSIDER MEMORY TO BE A LONG LINE OF "1'S" AND "0'S." EACH MEMORY LOCATION CONTAINS EIGHT BITS AND BY USING CONSECUTIVE MEMORY LOCATIONS ONE CAN BUILD UP LONG "STRINGS" OF BITS. AGAIN, THE "STRING" CAN BE "BROKEN" AT ANY DESIRED POINT AND MANIPULATED AS DESIRED. THIS TECHNIQUE CAN BE USED, SAY, TO SIMULATE A HUGE "SHIFT REGISTER" (USING ROTATE INSTRUCTIONS) OR TO REPRESENT AN EVENT OCCURING, OR NOT OCCURING AT POINTS IN TIME, OR AT DISTANCES ALONG A LINE. IN THIS VIEW, A BIT IS "ADDRESSED" AS BEING AT A SPECIFIC "POSITION" WITHIN A SPECIFIC "MEMORY ADDRESS LOCATION." WHILE THE PROGRAMMING "OVERHEAD" TO MANIPULATE SUCH "DATA" WILL GENERALLY BE MORE COMPLICATED THAN THE CASE WHERE ENTIRE MEMORY "WORDS" ARE USED TO REPRESENT A "SYMBOL" OR PIECE OF DATA, ONE CAN SEE THAT THE BASIC CONCEPT OF CONSIDERING ALL BITS IN MEMORY AS BEING FORMED OF ONE CONTINUOUS "LINE" OF ONES AND ZEROS IS A VALID, AND OFTEN USEFUL IMAGE.

THE TWO DIMENSIONAL VIEW

THE CONCEPT OF VIEWING MEMORY AS A TWO DIMENSIONAL PLANE WILL BE STARTED BY CONSIDERING AN IMAGE AT THE BIT LEVEL.

ADDR N * ADDR N+X+1 * ADDR N+2X+1

ADDR N	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
	1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
	1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1
	1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1
	1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1
	1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1
	1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1
	1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1
	1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1
	1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1
	1 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 1
	1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1
	1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1
	1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1
	1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1
	1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1
	1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1
	1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1
	1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1
	1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1
	1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
ADDR N+X	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

ADDR N+X * ADDR N+2X * ADDR N+3X

THE ABOVE DIAGRAM ILLUSTRATES AN IMAGE CREATED BY THE STATUS OF THE

BITS IN A "PLANE" OF MEMORY. THE "PLANE WAS ESTABLISHED BY ESSENTIALLY TAKING "LINES" OF MEMORY ADDRESSES (AS PRESENTED IN THE "ONE DIMENSIONAL VIEW") AND PLACING THEM ALONGSIDE ONE ANOTHER TO FORM A SURFACE OR "PLANE." THIS CONVENTION WOULD BE ESTABLISHED BY THE MANNER IN WHICH THE PROGRAMMER MANIPULATED THE MEMORY POINTER IN THE CPU. IN THE ABOVE ILLUSTRATION THE "PLANE" IS ESTABLISHED AT THE MOST FUNDAMENTAL (AND COMPLEX) LEVEL AND BITS WITHIN EACH WORD ARE MANIPULATED. AS MAY BE OBSERVED IN THE ABOVE DIAGRAM, ONE CAN VIEW AND MANIPULATE BITS IN MEMORY SO AS TO FORM "PICTURES" OR "DIAGRAMS." THE ABOVE REPRESENTS A RECTANGLE, A DIAMOND, AND A CROSS AS AN IMAGE MADE UP OF APPROPRIATE ONES AND ZEROS IN SELECTED BIT POSITIONS. ONE COULD THUS MANIPULATE PORTIONS OF MEMORY TO REPRESENT "PICTURES." (OR CHARTS, GRAPHS, PLOTS!) THE DEGREE OF DETAIL WHICH ONE CAN OBTAIN BY SUCH MANIPULATIONS IS A FUNCTION OF HOW MANY "BITS" ARE USED TO REPRESENT A GIVEN "AREA" OF A REAL (OR PROPOSED "REAL") OBJECT. THE ABOVE EXAMPLE PRESENTS ALL KINDS OF POSSIBILITIES FOR THE CREATIVE PROGRAMMER. ONE CAN USE SUCH TECHNIQUES TO FORM "MODELS," CREATE PATTERNS, AND SO FORTH.

IN FACT, GOING THE OTHER WAY SO TO SPEAK, THAT IS FROM HAVING THE COMPUTER GENERATE PATTERNS OR OBJECTS, ONE CAN ALSO TAKE THE TWO DIMENSIONAL CONCEPT AND APPLY IT TOWARDS HAVING THE COMPUTER RECOGNIZE OBJECTS BY "PROJECTING" THEIR SHAPE OR FORM AS A SIMILAR IMAGE OF ONES AND ZEROS IN MEMORY.

MUCH RESEARCH IS CURRENTLY BEING CONDUCTED TOWARDS DEVELOPING ALGORITHMS THAT CAN RECOGNIZE "OBJECTS." ONE APPROACH THAT IS BEING STUDIED IS AN INTERESTING APPLICATION OF THE TWO DIMENSIONAL CONCEPT. A "PICTURE" OF AN "OBJECT" IS "MAPPED" INTO MEMORY WITH "1'S" BEING USED TO REPRESENT THE AREA OCCUPIED BY THE "OBJECT" AND "0'S" FOR AREAS "OUTSIDE." THEN, THE COMPUTER IS "TRAINED" TO IDENTIFY OBJECTS BY USING ALGORITHMS BASED ON A "NEIGHBORING BITS" SCHEME. IN THIS MANNER, THE COMPUTER DETERMINES HOW MANY "0'S" SURROUND A "1" AND PERFORMS CALCULATIONS TO FIND THE "OUTLINE" AND SHAPE OF THE OBJECT. THESE FINDINGS ARE THEN COUPLED WITH COMPLEX ALGORITHMS TO ATTEMPT TO IDENTIFY THE OBJECT FROM A "CLASS" OF POSSIBILITIES.

SUCH PROGRAMS ARE OF COURSE QUITE COMPLEX AND THE DETAILS OF SUCH MANIPULATIONS ARE SOMEWHAT ESOTERIC. BUT, THE IDEA IS INTRIGUEING AND CAN PROVIDE FERTILIZATION FOR THE CREATIVE PROGRAMMER'S IMAGINATION.

TAKING THE TWO DIMENSIONAL VIEW TO THE MEMORY WORD LEVEL IS PERHAPS A BIT LESS COMPLICATED (IT IS! IT IS!) THAN CONSIDERING IT AT THE BIT LEVEL. IN THIS CASE, ONE NEEDS ONLY ENVISION A "PLANE" OF MEMORY WORDS WHICH CAN CONTAIN CODES FOR LETTERS, NUMBERS, SYMBOLS OR ACTUAL MATHEMATICAL VALUES. THE READER HAS ALREADY SEEN EXAMPLES OF PROGRAMS THAT COULD BE CONSIDERED AS TWO DIMENSIONAL IN ORGANIZATION. ONE FOR INSTANCE, WAS DESCRIBED IN CHAPTER FOUR IN THE PRESENTATION OF THE NAMES SORTING PROGRAM. THERE, LINES OF NAMES WERE FORMED "ONE BENEATH THE OTHER" IN ORDER TO MAKE THE SORT ROUTINE EASIER TO PROGRAM. ONE MIGHT REVIEW THE DIAGRAM SHOWING THE SAMPLE NAMES STORED IN MEMORY AS THEY RELATE TO THE MEMORY ADDRESSES, WHICH WAS PRESENTED NEAR THE END OF CHAPTER FOUR.

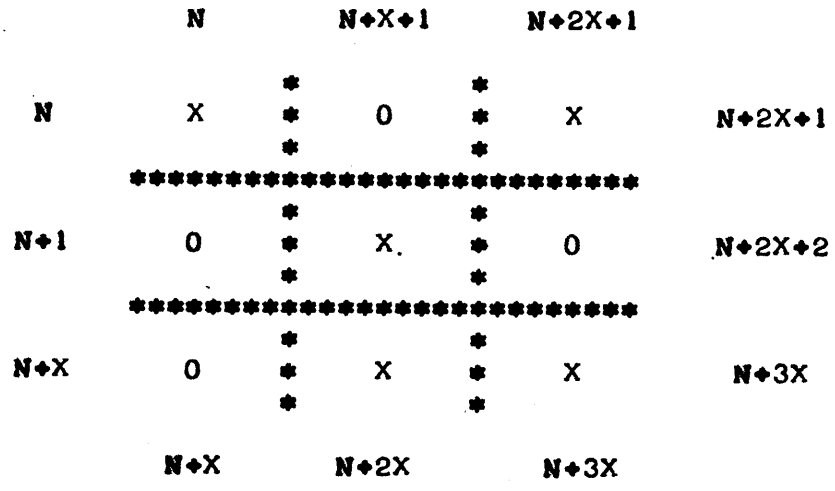
THE PROGRAMMER IS AGAIN REMINDED THAT AS IN THE ONE DIMENSIONAL VIEW, THE MEMORY ADDRESSES THAT FORM THE "X" AND "Y" BOUNDARIES OF A TWO DIMENSIONAL MEMORY PLANE CAN ACTUALLY BE THOUGHT OF AS ARBITRARY UNITS - SUCH AS TIME, FREQUENCY, OR DISTANCE, AND THE PROGRAMMER ALSO HAS THE FREEDOM TO "SCALE" BOTH THE "X" AND "Y" BOUNDARIES BY APPROPRIATE SOFTWARE. THE NEXT ILLUSTRATION SHOWS HOW AN "ALTITUDE MAP" OF A GEOGRAPHICAL AREA MIGHT BE STORED IN A "PLANE" OF MEMORY.

	N	N+X	N+2X	N+3X	N+4X	N+5X	N+6X	
N	060	065	070	075	074	070	064	500 YDS
N+1	061	076	084	083	080	076	070	400 YDS
N+2	062	078	088	098	096	091	082	300 YDS
N+3	062	078	090	102	101	089	072	200 YDS
N+4	055	070	075	053	047	063	039	100 YDS
N+(X-1)	040	035	020	010	011	009	008	0 YDS

0 YDS 100 YDS 200 YDS 300 YDS 400 YDS 500 YDS 600 YDS

IN THE ABOVE ILLUSTRATION EACH MEMORY LOCATION CONTAINS A VALUE THAT REPRESENTS THE ELEVATION OF A PIECE OF LAND. THE TOP AND LEFT SIDE OF THE ILLUSTRATION SHOWS THE ACTUAL MEMORY ADDRESSES IN THE COMPUTER WHILE THE BOTTOM AND RIGHT SIDE ILLUSTRATE THAT EACH "ADDRESS" ACTUALLY STANDS FOR "100 YARDS DISTANCE." IT SHOULD BE APPARENT THAT THE ELEVATION FACTORS COULD BE, INSTEAD, INCHES OF RAINWATER, OR A TEMPERATURE PROFILE FOR THE AREA, OR, AS PREVIOUSLY MENTIONED, THAT THE "YARDS" CAN BE ALMOST ANYTHING ELSE THE PROGRAMMER MIGHT DESIRE TO DEFINE.

AS A FINAL EXAMPLE OF THE TWO DIMENSIONAL CONCEPT, THE READER WILL BE LEFT WITH THE FOLLOWING DIAGRAM - WHICH HOPEFULLY WILL ENCOURAGE ONE TO CONSIDER THE POSSIBILITIES FOR MUCH MORE COMPLEX "BOARD GAMES!"



FINALLY, THE READER WILL BE REMINDED, THAT IN A MANNER SIMILAR TO FORMING A "RING" AS DISCUSSED IN THE ONE DIMENSIONAL VIEW, ONE CAN ALSO CONSIDER FORMING A "CYLINDER" OUT OF A "PLANE" WITH INTERESTING RAMIFICATIONS!

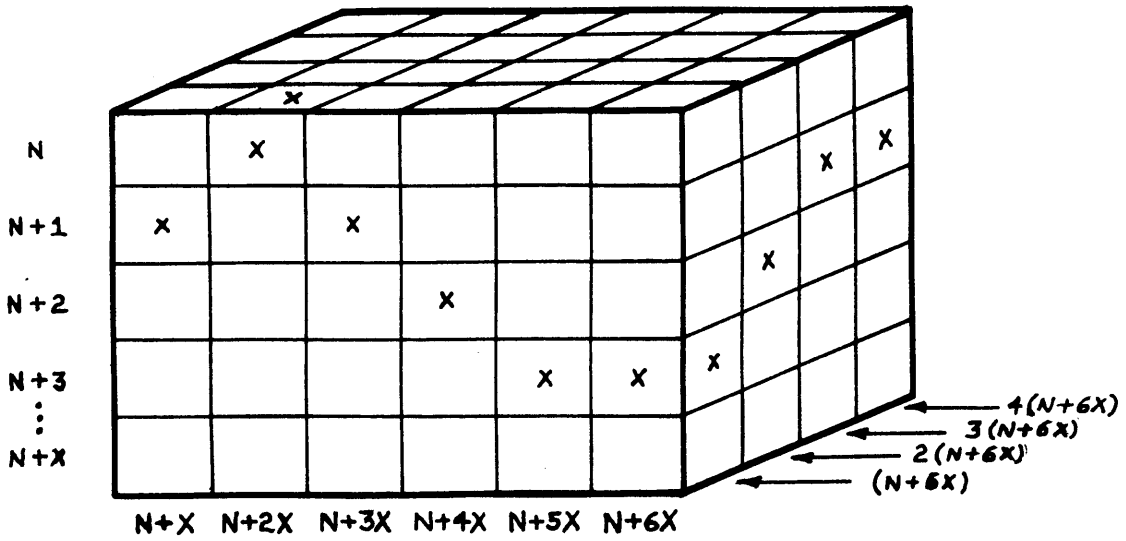
THE THREE DIMENSIONAL VIEW

IT SHOULD NOW BE APPARENT THAT IF ONE CAN SET UP MEMORY LOCATIONS BY APPROPRIATE ADDRESSING TO REPRESENT "LINES" AND "PLANES," ONE CAN EXTEND THE PRINCIPLE OUT TO THE "THIRD DIMENSION" TO FORM "CUBES" OF MEMORY. THERE ARE MANY INTERESTING POSSIBILITIES WHEN MEMORY IS VIEWED IN

THIS MANNER. ONE CAN PLOT THREE DIMENSIONAL GRAPHS OR VECTORS. ONE CAN APPROACH MANY TYPES OF "MODELING" AND MANIPULATE SUCH MODELS SO AS TO OBTAIN DIFFERENT "CROSS-SECTIONAL" VIEWS.

AS IN THE CASE OF THE ONE AND TWO DIMENSIONAL IMAGES, THE PROGRAMMER CAN SUBSTITUTE (EFFECTIVELY) MEMORY ADDRESSES FOR SCALE FACTORS, NOW ALONG THREE AXIS. AND, AS IN THE PREVIOUS EXAMPLES, ONE CAN TAKE SUCH MANIPULATIONS DOWN TO THE BIT LEVEL IF DESIRED.

THE DIAGRAM BELOW PRESENTS AN IMAGE OF MEMORY WHEN VIEWED AS A THREE DIMENSIONAL WORKING AREA.



IT IS HOPED, THAT BY THIS TIME, THE READER HAS RECEIVED SUFFICIENT INFORMATION ON THE PRACTICAL ASPECTS OF MACHINE LANGUAGE PROGRAMMING FROM THE PRECEEDING CHAPTERS, AND THAT THIS CONCLUDING CHAPTER HAS PROVIDED SOME STIMULATING CONCEPTS, SO THAT THE READER MAY GO ON TO DEVELOP PROGRAMS THAT WILL BE OF PARTICULAR VALUE TO THE INDIVIDUAL. IT IS ALSO HOPED THAT THOSE WHO HAVE BEEN INTRODUCED TO THE SUBJECT BY THIS MANUAL, WILL FIND MACHINE LANGUAGE PROGRAMMING AN EXCITING, ENJOYABLE, AND IN AS MANY WAYS AS POSSIBLE, A REWARDING ENDEAVOR!