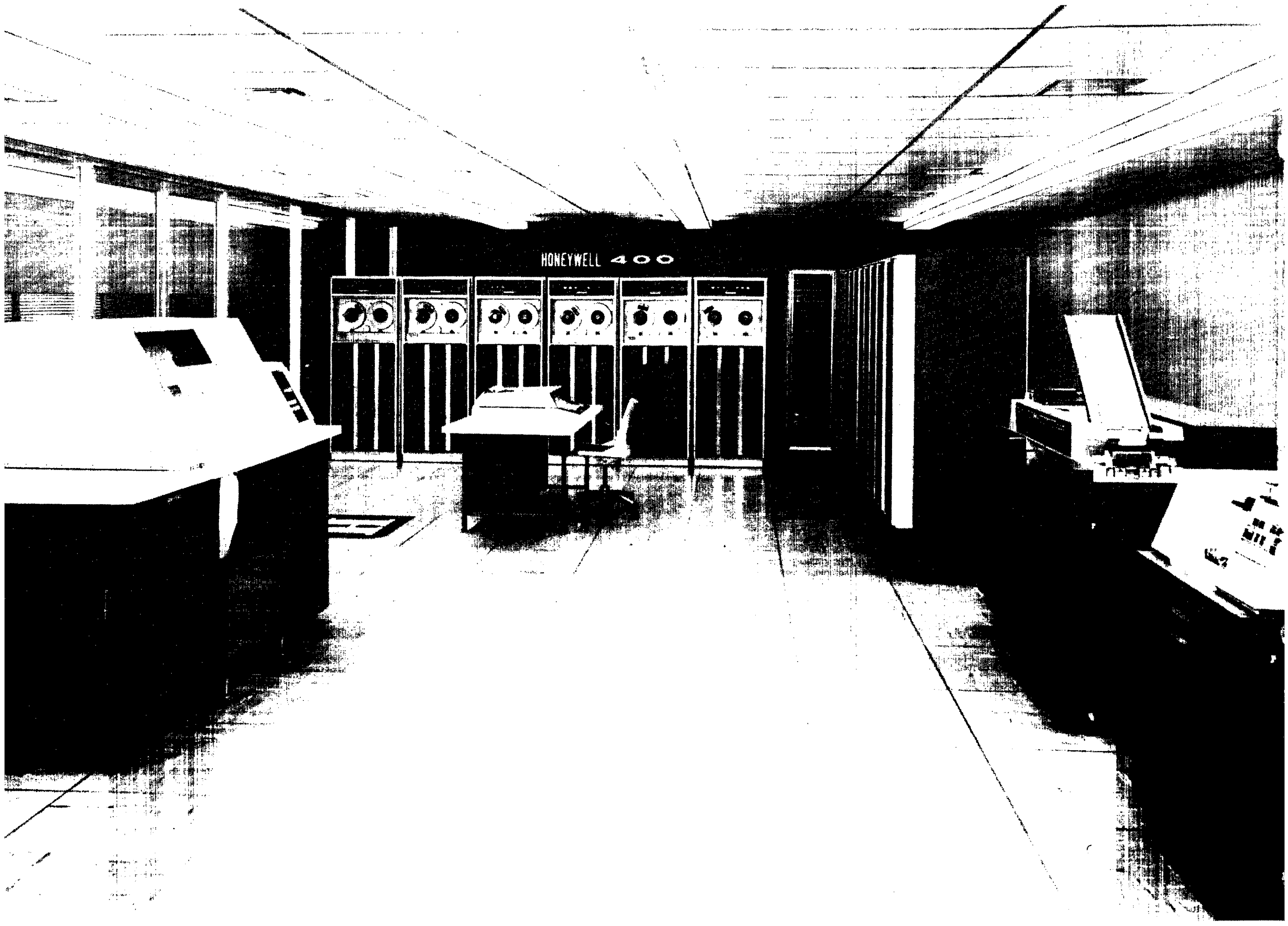
MANUAL OF ASSEMBLY LANGUAGE

EFFICIENT ASSEMBLY SYSTEM

II

HONEYWELL 400

Transistorized Data Processing System



Litho in U.S.A.
NP-1676

DSI-133
21062

EASY II

EFFICIENT ASSEMBLY SYSTEM

MANUAL OF ASSEMBLY LANGUAGE

PRICE $4.00

Questions and comments regarding this manual should be addressed to:

Honeywell Electronic Data Processing
Merchandising Division
60 Walnut Street
Wellesley Hills 81, Massachusetts

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

FOREWORD

This manual describes the programming language for EASY II (Efficient Assembly System). It also contains background information on the entire EASY II system and on the Honeywell 400 Transistorized Data Processing System. It is intended for those familiar with data processing techniques.

INTRODUCTION

EASY II is a programming and operating system designed to assist the users of larger Honeywell 400 systems (at least 2,048 words of memory) in all phases of their data processing efforts. It greatly simplifies programming and speeds other operations required for efficient operation.

The major elements of EASY II are:

1. An EASY language which identifies instructions with easily remembered operation codes and identifies memory locations with symbolic tags of the programmer's choosing;
2. An Assembly Program which accepts programs written in the EASY language, translates them into machine language, and prints a listing of their coding with diagnostic information;
3. An updating and selection program, LAMP-PSP, which maintains a master program tape containing previously assembled programs and containing a library of frequently-used routines; it also selects programs from this tape for production or test runs;
4. A Monitor program which loads and runs programs;
5. Sort and Collate programs;
6. Routines for generating coding which handles card input, tape input and output, and the preparation of punched and printed reports;
7. A tape-handling routine, THOR, which enables the operator to position tapes and perform general tape maintenance; and
8. A library of routines for handling other common data processing problems.

As shown in Figure 1, the application of these programs produces a complete system for preparing a program, testing it, and running it. The core of this system is the use of the Assembly, LAMP-PSP, and Monitor programs.

The EASY II programmer writes his instructions in EASY language and has these instructions punched in EASY cards. The Assembly program then converts the data on these cards into the machine language of the Honeywell 400, recording the converted coding on tape and also punching a card deck, if desired. Assembly also prints a listing of the original and converted coding, complete with diagnostic information including the identification of certain types of programming errors.

The next step is to add the newly assembled programs to the master program tape, the

SECTION I

THE HONEYWELL 400

The basic unit of information in the Honeywell 400 System is a fixed-length word which consists of 48 information bits and two parity bits used by the automatic checking circuitry. Each high-speed memory location is capable of storing one such word. The check bits of memory words are not directly available to the programmer nor are their values subject to program control; therefore subsequent discussion of memory words will refer to the 48 information bits, unless otherwise noted.

Data Words

A computer program generally manipulates data in one or more different forms: binary, octal, decimal, alphanumeric, or a combination of these. The Honeywell 400 is capable of handling all of these types of information. It may interpret the 48 bits of a word in groups of three for the purpose of octal operation, in groups of four for decimal operation, in groups of six for alphanumeric operation, or as individual units of information for pure binary operation. Figure 2 illustrates the structures of these different words.

Octal words in the Honeywell 400 consist of 16 digits. Only the digits 0 through 7 are legal octal characters. The Honeywell 400 can accept input or code its output in these characters.

A decimal word contains either 11 decimal digits with a sign, or 12 decimal digits without a sign. The decimal arithmetic instructions interpret all operands as a sign and 11 decimal digits. The sign is represented by four bits: a negative sign is represented by four binary zeros, any other bit configuration in the sign position (high-order four bits) is interpreted as a positive sign.

An alphanumeric word in the Honeywell 400 consists of eight six-bit groups. Each six-bit configuration may represent any one of 26 alphabetic characters, 10 decimal digits, or 20 special symbols (such as punctuation marks).

The 48 binary digits of a word may also represent an unsigned binary number of 48 bits. The binary arithmetic instructions, binary add and binary subtract, treat their operands as 48-bit unsigned numbers.

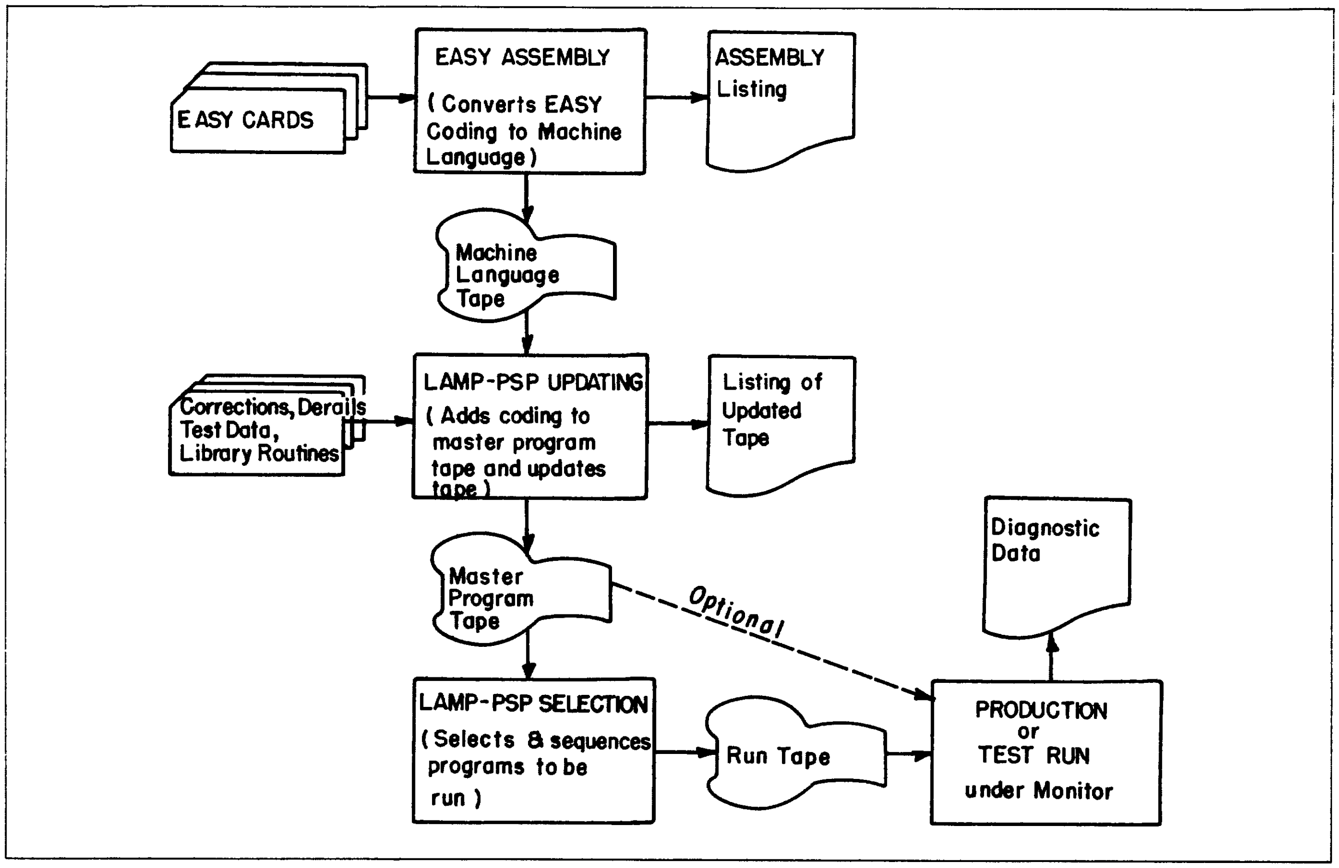


Figure 1. Simplified Representation of EASY II System

tape which stores the EASY library file and the machine-language coding for all programs to be processed. This step is accomplished by LAMP-PSP (Library Additions and Maintenance Program - Program Selection Process). This program can also correct and update the tape as directed by an input card deck. The output of this program is an updated master program tape and a printer listing of the programs on that tape.

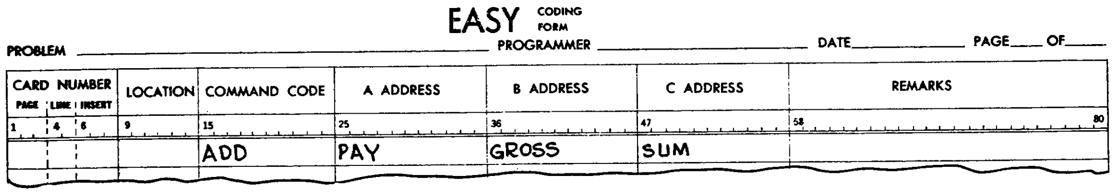
Then, the LAMP-PSP program may be used again to select programs from the master program tape and to record them on a run tape. This process creates a tape with programs arranged in the order in which they will be processed. Thus, it minimizes the time required to load each program into memory.

Finally, to allow testing and running of programs without interruption, EASY Monitor provides facilities for loading programs automatically and for printing diagnostic data on an on-line printer. This data is dumped dynamically without altering the original program and without manual intervention. Thus, the use of computer time is again minimized by eliminating brief, repetitive runs.

EASY II Language

The EASY II language consists of symbolic machine instructions, library calls, and descriptors for peripheral functions.

"ADD PAY (to) GROSS (and store in) SUM" - this is an example of an EASY symbolic instruction. The programmer writes such an instruction on an EASY coding form as follows:



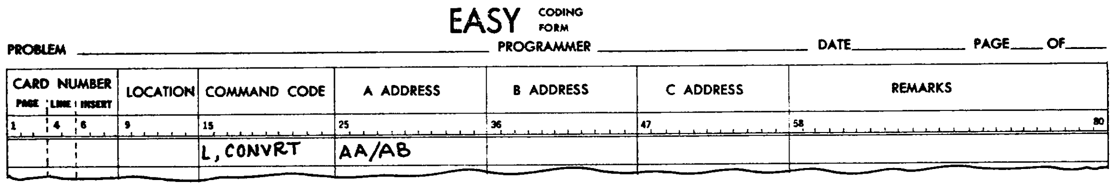
EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE LINE INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	9	15	25	36	47	58 80
		ADD	PAY	GROSS	SUM	

Two major advantages of EASY instructions are immediately obvious: their three-address format allows manipulating two operands and storing the result with a single instruction - the most natural way of handling data; their use of mnemonic operation codes (e.g., ADD) and of tags (e.g., PAY, GROSS, SUM) simplifies coding and eliminates the need for the programmer to recall the actual memory addresses of the operands.

Library call instructions allow the programmer to insert previously checked-out routines in a program. Such instructions are similar to symbolic instructions and are equally easy to use. For instance, to call in a conversion routine, the programmer need only write:



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE LINE INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	9	15	25	36	47	58 80
		L, CONVRT	AA/AB			

The Assembly program will automatically call in this routine and specialize it according to the parameters represented by AA and AB. It will then insert the routine at the point in the program where the call instruction was given. Thus, once coded in a generalized form, any routine may be kept on the master program tape for easy insertion into a program.

Descriptors are used to generate routines which handle tape input and output operations, card input operations, and operations which print or punch a report. A descriptor is a line of coding consisting of parameters which describe a file on tape, the format of card data, or the format of a report to be prepared. The Assembly program interprets these parameters and generates a routine to perform all facets of the specified operation. The programmer then uses this routine by inserting linkage instructions in his program.

The process is similar to that used with library routines but differs from it in one important respect: rather than generate a number of small routines to handle each input or output function, Assembly generates one all-encompassing routine and stores it separately from the main program. This technique saves memory space by storing coding certain to be used repetitively and then linking the main program to it.

The EASY language has other important advantages, such as permitting the use of constants, literals, and highly flexible forms of addressing. In addition, special EASY control instructions facilitate segmenting programs, allocating and reserving memory, and setting up registers.

Assembly Program

The Assembly Program translates programs from the EASY language into the machine language of the Honeywell 400; it inserts coding and generates routines as specified in library calls and descriptors. The output from Assembly is threefold:

1. A file of machine-language programs on magnetic tape. This file represents the machine-language equivalent of the EASY instructions with additional coding as specified in the library calls and descriptors;
2. A binary card deck representing the assembled coding may also be punched; and
3. A complete printed listing of each assembled program, including the line number, the symbolic EASY instructions, the machine-language coding (in octal notation), the memory location assigned to each word (in octal and decimal), and any errors detected during Assembly. These errors include illegal operation codes, illegal addressing, duplication of tags, etc.

Assembly uses the principle of "batch processing"; that is, it assembles several programs without interruption, detecting and indicating errors in the programs being assembled but continuing to assemble the programs. This is another time-saving feature of EASY.

EASY programs can be assembled on a Honeywell 400, 800 or 1800. The result of all these assembly processes is a program in the Honeywell 400 machine language, though this program can also be run on a Honeywell 800 or 1800 using an automatic simulator routine.

LAMP-PSP

LAMP-PSP is a dual-purpose program which maintains a master program tape and selects programs from this tape to create a run tape.

The updating and/or selection process is accomplished as directed by a card deck containing special control instructions together with coding to be added to the tape, corrections, derails, and test data. Again, this program "batch processes", updating or selecting several programs or library routines in a single continuous run.

The output of the updating process is a revised master program tape and a listing of this tape. The new tape includes all library routines and programs on the old tape except those deleted by control instructions in the input deck. It is arranged in alphabetical order in two major files, the library file containing all library routines and the program file containing systems programs (e.g., Assembly, LAMP-PSP, Minotor, etc.) and object programs (i.e., operational programs to be processed). The printed listing shows the order of programs on the tape and records changes made during the updating process.

During a production or test run, locating programs in the alphabetically ordered master tape can be quite time-consuming. Therefore it is usually desirable (though not necessary) to select programs from this tape and create an integrated run tape with programs arranged in the order in which they will be executed. This is easily accomplished by preparing an input card deck which specifies programs in the desired order and then using this card deck to control the selection process.

The selection process produces a run tape with programs arranged in the order of execution and a printer listing of this tape. The listing is simply a table of contents for the run tape, showing program and segment names in the order of their appearance on the tape.

Monitor

EASY Monitor is a system of programs and routines which supervise the loading and operation of checkout, production and systems runs. Included in the functions automatically performed by Monitor are: controlling test data distribution and providing memory and tape dumps; controlling the Orthocorrection routine which regenerates lost tape data; loading and starting programs; and keeping the operator informed of the progress of the run by means of typeouts. Monitor makes provision for automatic operation of runs under its own control and for manual operation under the operator's control. Use of a run tape (rather than the master program tape) enables Monitor to load and execute each program without manual intervention.

Sort and Collate Programs

The EASY II Sort and Collate programs provide an unusually fast and versatile method of ordering a file and of combining several pre-ordered files. The programs may be used separately or together as the particular situation demands.

The Sort accepts one file of tape data arranged in random fashion and produces a file arranged in ascending alphanumeric sequence. The input file may be recorded on several reels; the output file is normally written on one reel with items sequenced on the basis of keys identified by the programmer. High sorting speeds are achieved with the "polyphase" technique developed by Honeywell.

The Collate program combines two, three, four or five pre-ordered files and writes a single ordered file arranged in ascending alphanumeric sequence. Each input file may consist of several tape reels; the output file may be written on any number of tape reels.

Both programs make provision for "own coding". That is, they allow the programmer to process data being sorted or collated and to modify the Sort or Collate program itself.

Tape Input/Output, Report Editor, and Card Input Routines

The three input/output routines eliminate the chores involved in controlling peripheral operations and free the programmer to concentrate on solving the main problem at hand. They are:

Tape Input/Output Routines - these routines read and write EASY tape files and perform all associated control operations; they are also able to handle non-standard tape files. The routines open and close tape files, get or write items in a file, and force tape swaps.

Report Editor - this routine performs the editing, output, and control operations required when printing a report in a specified format, writing a tape to be printed off-line, or punching a card deck.

Card Input Editor - this routine reads punched cards, edits the card data, and stores the data in memory.

These routines can be used together in any combination. All are generated on the basis of descriptor cards and used by linkage calls included in the main program.

THOR

THOR (the Tape Handling Option Routine) is a general tape-handling and correction routine controlled by parameters which an operator types in from the console. Under the direction of these parameters, THOR positions, copies, corrects, and edits tape. It also locates information on tape, compares the contents of two tapes for discrepancies, and performs general tape maintenance. THOR manipulates tapes using two basic control methods: first, it moves tapes forward or backward a specified number of records; secondly, it searches forward until it finds a specified record. This second method is known as the file option; it represents an unusually versatile technique for locating information on tape.

EASY Library

A number of thoroughly tested routines are supplied by Honeywell for storage on the library section of the master program tape. Included are several scientific routines such as floating-point add and subtract, matrix and multiple regression analysis, etc., and a number of frequently used business routines. Detailed specifications on each of these routines are available under separate cover. In addition, by means of LAMP-PSP each installation can add to its own library to meet specific requirements. Any routine on the master program tape can easily be inserted into a program with a call instruction. Assembly specializes each routine according to parameters in the call instruction and produces coding which efficiently handles the operation described by the parameters.

A memory word may also contain suitable combinations of octal or decimal digits, alphanumeric characters, and binary digits totalling 48 bits.

Instruction Words

The 48 bits of a Honeywell 400 instruction word are interpreted as four groups of 12 bits each. Counting from the left, bits 1 through 12 represent the command code group; and bits 13 through 24, 25 through 36, and 37 through 48 are designated as the A address group, B address group, and C address group, respectively. The address portions of instruction words are normally used to designate the locations of operands and results, but in certain instructions they may contain special information, such as the number of words to be moved, the number of decimal or binary digits to be shifted, or the number of words to be written.

The machine instructions are grouped in eight categories: arithmetic, logical, shift, decision, transfer and sequence change, edit, peripheral, and index and check. In general, Honeywell 400 instructions are uniquely identified by the high-order six bits (1-6) of the command code group; these six bits are called the operation code. (The exceptions to this rule are explained below.) The remaining six bits (7-12) of the command code group represent the index registers associated with each address group; bits 7 and 8 are associated with the A address group, bits 9 and 10 with the B address group, bits 11 and 12 with the C address group.

In certain instructions, bits 9 and 10 (i.e., the index register group associated with the B address) are used as part of the operation code; the B address cannot be indexed in these instructions.

The command codes for individual instructions, together with their mnemonic operation codes in EASY language are discussed by major instruction type in Section IV.

The general format of a Honeywell 400 machine instruction word is shown below.

1 Command Code			12	13 A Address 24	25 B Address 36	37 C Address 48
Operation Code	Ai	Bi	Ci			
XXXXXX	XX	XX	XX	XXXXXXXXXXXX	XXXXXXXXXXXX	XXXXXXXXXXXX

The two bits in Ai designate the index register used with the A address group. The value of Ai may be:

1. 00 - the A address is not indexed;
2. 01 - the A address is indexed by index register 1;

3. 10 - the A address is indexed by index register 2; or

4. 11 - the A address is indexed by index register 3.

The two bits which represent Bi and the two bits which represent Ci are interpreted similarly, save in those instructions where the Bi field is part of the operation code.

Instruction	Command Code (12 Bits)			Address A (12 Bits)			Address B (12 Bits)			Address C (12 Bits)							Groups
Alphanumeric	R 1	O 2	B 3	I 4	N 5	S 6	O 7	N 8									Characters
Decimal (Signed or Unsigned)	± 1	1 2	2 3	3 4	4 5	5 6	6 7	7 8	8 9	9 10	0 11	1 12					Digits
Octal	0 1	1 2	2 3	3 4	4 5	5 6	6 7	7 8	7 9	6 10	5 11	4 12	3 13	2 14	1 15	0 16	Digits
Binary	1 (48 Binary Digits) 48																Bits

Figure 2. Honeywell 400 Word Formats

Reserved Memory Locations

Memory locations in the Honeywell 400 high-speed memory are directly addressable by internally stored instructions. While the use of almost all memory locations is under control of the program, some locations are also reserved for special functions. These special memory locations may be divided into four categories: special-purpose locations, unprogrammed subsequence locations, working subsequence locations, and input/output area locations. (The addresses of these locations are given in Appendix A.)

The special-purpose locations are those which are used by the central processor to store checking and control information, and also information relating to previously executed instructions. One of these locations, for example, contains the three index registers and the sequence register, while another - called the low-order product word - contains the low-order signed 11 decimal digits of the result of the multiply instruction.

In the event of certain abnormal conditions arising during a program run, control of the program is transferred to one of several unprogrammed subsequence locations, the exact one depending on the particular condition. An unprogrammed subsequence would, for example, be caused by an overflow during addition.

There are eight working subsequence locations, each of which is associated with a particular peripheral device. After completion of data transfer on a peripheral device, control transfers to the location corresponding to the device. These locations are each loaded so that the programmer may direct the sequence of instruction execution after completion of data transfer (see Section IV).

The working subsequence locations are also used in conjunction with the console "fixed start" instructions. This feature allows the operator to start processing at one of these locations or at one of the locations specially assigned to the console.

Input/Output Area Locations

When the central processor executes a card read, card punch, or print instruction, the corresponding area in memory is automatically addressed. These areas are the card read area (input), the card punch area (output), and the printer area (output). Information from a card is read into the card read area in a complete card image; information to be punched on a card is obtained from the card punch area; and information to be printed is obtained from the printer area. These input/output areas are also implicitly referenced by the corresponding edit instructions, so that the card edit instructions normally obtain the data to be edited from the card read area; the punch edit instructions normally edit into the card punch area; and the print edit instructions normally edit into the printer area.

These areas may be addressed (like other memory locations) by non-edit instructions, and data may be transferred into and out of them under control of non-edit instructions; i.e., one or more of the areas may be used for any program purpose(s) if they are not already being used for their input/output functions.

Index Registers

The Honeywell 400 contains three index registers, each of which stores 12 binary digits. The 12 bits are treated as an unsigned binary quantity and can thus represent a value in the range 0 through 4095 (decimal). In instruction words (see page 2), each index register is designated by a two-bit code:

1. 01 - index register 1;
2. 10 - index register 2;
3. 11 - index register 3;
4. 00 - no indexing (i.e., direct addressing).

When these codes appear in the index register bit positions of an instruction word (bits 7 and 8, 9 and 10, 11 and 12), the referenced address is added temporarily to the contents of the specified index register to form an effective working address (unless the bit configuration is 00, in which case there is no indexing of the corresponding address). This addition is in the form of augmenting; i.e., the contents of the index register and the address remains unchanged upon completion of the instruction. The three index registers may be used in any combination within an instruction and in any sequence within a program. Thus, an instruction which allows indexing of all three address fields may designate any one of 64 possible indexed address combinations (00 00 00 through 11 11 11; i.e., from no indexing through indexing all three addresses using index register 3). Addressing and the use of index registers in EASY language are described in detail in Section III.

Sequence Register

The sequence register is a special register which is used to control the sequence in which instructions are executed. It stores one high-speed memory location address (i.e., 12 bits) in the range 0 through 4095. In general, it contains the address of the memory location which holds the next instruction to be executed. Three types of instruction selection can be distinguished, two of which are dependent on the setting of the sequence register.

Normal Incrementing

This is the general case in which the setting of the sequence register is increased by 1, so that the next instruction is selected from the memory location following that of the current instruction.

Sequence Register Resetting

Certain instructions cause the sequence register to be completely reset, so that the next instruction may be selected from anywhere in the program. When a sequence change instruction is selected, it causes a predetermined address to be stored in the sequence register. (The resetting is unconditional in the sequence change and store index register instruction but, in other instructions, is conditional upon specified conditions being met.) The instruction stored in the memory location specified by this address is selected to be executed next. From this point, the sequence register is incremented normally until it encounters another sequence change instruction.

Subsequence Calls

The central processor can, in special circumstances, select the next instruction without either referring to or changing the setting of the sequence register. The address of the next instruction is either determined by machine logic (unprogrammed subsequence) or is specified in one of the address fields of the current instruction (programmed subsequence). In both cases, the sequence register is completely ignored; its contents are left unmodified for the execution of one instruction. This one instruction may, however, cause a sequence change, in which case the original setting of the sequence register may be stored until needed by using the store index register instruction (see Section IV).

Simultaneous Processing with Peripheral Operations

The unprogrammed subsequence calls described above serve to interrupt processing of the main program and to signal the occurrence of certain events. The programmer can take maximum advantage of this feature by using it to facilitate simultaneous processing.

Simultaneous processing in the Honeywell 400 consists of executing instructions during part of the time required for a peripheral operation. Subject to the rules described in Appendix C, simultaneous processing can greatly increase the efficiency of certain applications. For instance, the programmer may read a card in the "interlocked" mode to prevent simultaneous processing or he may read a card "without interlock" to permit simultaneous processing. Reading the card "interlocked" requires 93 milliseconds and prevents the execution of other instructions during all but 6 milliseconds; reading a card "without interlock" also requires 93 milliseconds but allows simultaneous processing during 39 milliseconds of that interval - enough time for some 300 add instructions.

Most frequently, simultaneous processing is used to speed the execution of a single program. However, it may be used to perform two completely independent programs, such as a program printing data from tape and a program sorting data.

SECTION II

PROGRAM PREPARATION

Programs to be assembled by EASY are written on preprinted coding sheets, as shown in Figure 3. The coding on these sheets is then punched on standard 80-column cards according to the fixed-field format shown in Figure 4. Normally, an instruction occupies an entire line on the coding sheet and an entire punched card, and is assembled as one machine word.

When an entire program deck, complete with all necessary control instructions, is assembled by EASY, the program is produced in machine language on magnetic tape. In addition, EASY produces a listing of the program in printed form.

As shown in Figure 4, the EAST input card contains seven fixed fields. The function of each of these fields is described below.

Card Number Field (columns 1-8)

Card numbers specify the sequence of cards in an input deck when a new program is assembled.

The card number field on an EASY coding sheet is divided into three subfields:

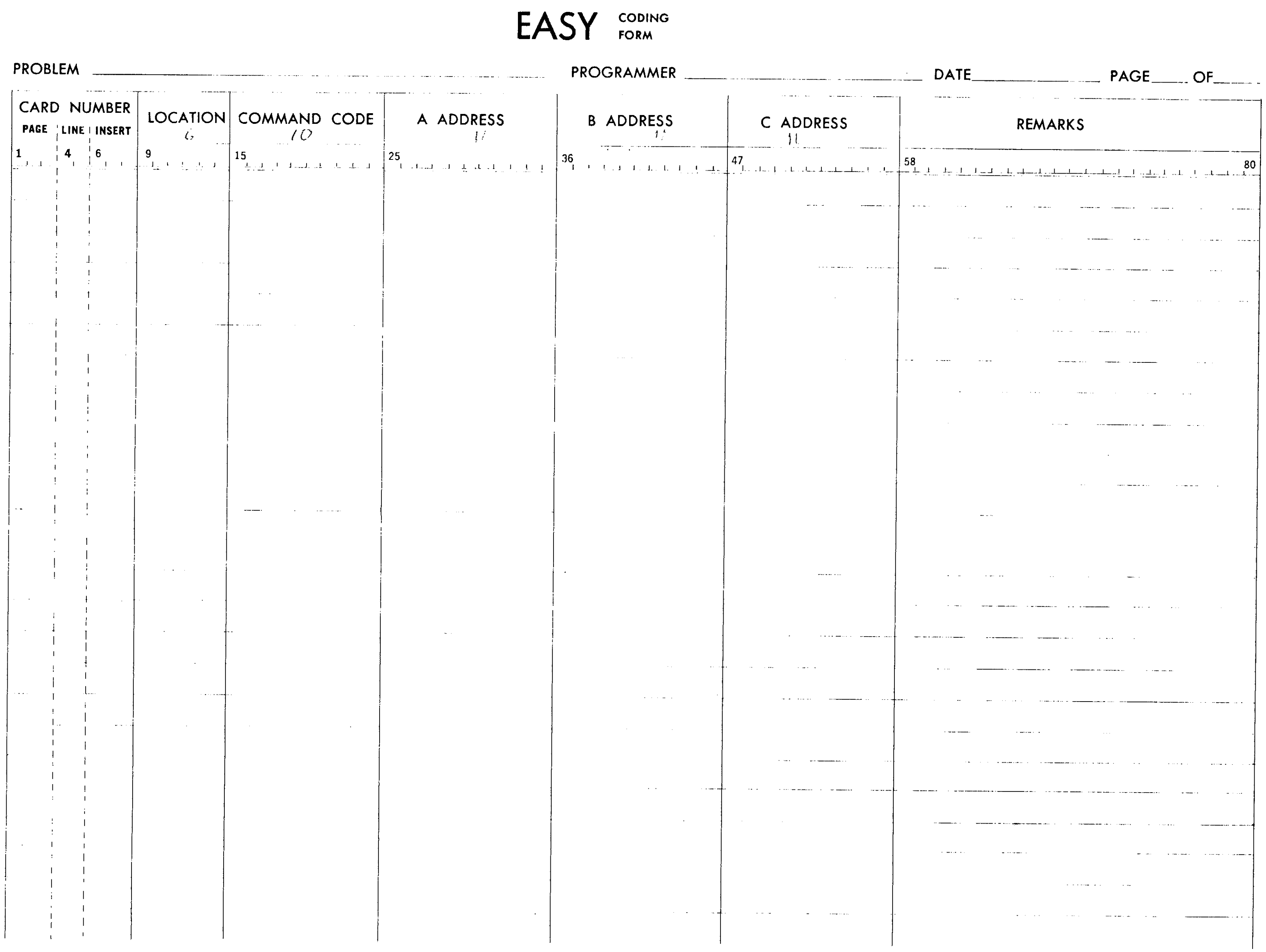
1. Page - corresponding to columns 1 through 3 on the EASY card;
2. Line - corresponding to columns 4 and 5 on the EASY card; and
3. Insert - corresponding to columns 6 through 8 on the EASY card.

The three insert columns are normally zeros; they are used only if inserts are made between two assigned numbers.

Assembly sorts the cards into order by card number (i. e., page, line and insert numbers) if requested to do so. Otherwise, any punches in this field are ignored and cards are accepted in input order.

Location Field (columns 9-14)

The location field specifies the address assigned to the word assembled from this card. It may be blank, may contain an absolute address (a number 0 through 4095), or may contain a symbolic address. If this field is blank, Assembly assigns the next higher available location. If it contains an absolute address, Assembly assigns the address specified. If it contains a symbolic tag, Assembly assigns the next higher available location and assigns the absolute value of



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ____ OF ____

CARD NUMBER			LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
PAGE	LINE	INSERT	6	10	11	11	11	
1	4	6	9	15	25	36	47	58 … 80

Figure 3. The EASY Coding Form

this address to address-field references to the same symbolic tag.

Cards with absolute addresses in the location field have no effect on the assignment of subsequent cards. That is, the next card will be assigned an address one higher than the last previous card without an absolute address in the location field.

The symbolic tags and their corresponding absolute addresses are printed out as part of the Assembly listing. The special symbols * and @ may not be used in location field.

Command Code Field (columns 15-24)

The contents of the command code field specify the type of entry defined by the whole card. The command code field might, for example, contain the mnemonic operation code of a machine instruction. In the case of an ordinary constant (i. e., unmixed), the command code field contains the code "CON"; in a mixed constant it contains one of the four terms defining the constant, the contents varying with the type of mixed constant being specified (see Section VI). The command code field may also be used to specify an EASY control instruction, a pseudo instruction for a library routine, or a descriptor card.

Address Fields (columns 25-35, columns 36-46, columns 47-57)

The three address fields correspond to the A, B, and C address groups of a Honeywell 400 machine instruction word. In general, the address fields designate the locations of operands or results. In certain instructions one or more of the address fields may contain literals or instruction parameters rather than an address; e.g., the B address in a shift instruction contains the number of digits or bits to be moved. In some instructions one or more address fields are not used, and can, in general, store information not related to the instruction. (These fields are identified in Section IV.)

In ordinary constants, the boundaries between the address fields are ignored; the constant starts in column 25 and continues in successive columns until the last character or digit of the constant is entered. A mixed constant is specified by four terms, one term in the command code field and one in each of the three address fields.

If an address is indexed, the corresponding address field contains an index register indication which follows the actual address or parameter and is separated from it by a comma (see Section I).

Remarks (columns 58-80)

On any card that represents a machine instruction or a constant, this entire field may be used by the programmer for remarks. Such information is not assembled and is solely for the programmer's convenience. The contents of the remarks field are, however, reproduced as part of the program listing. In addition, columns 74-80 may be used to identify cards as belonging to a certain program (see page 55).

A card containing only remarks may be included in a program at any point. Such a card is indicated by an "R" or a "P" followed by a comma punched in columns 9 and 10, respectively (location field). "R" causes the contents of the card to be printed as remarks on the next line, "P" causes the contents of the card to be printed at the top of the next page.

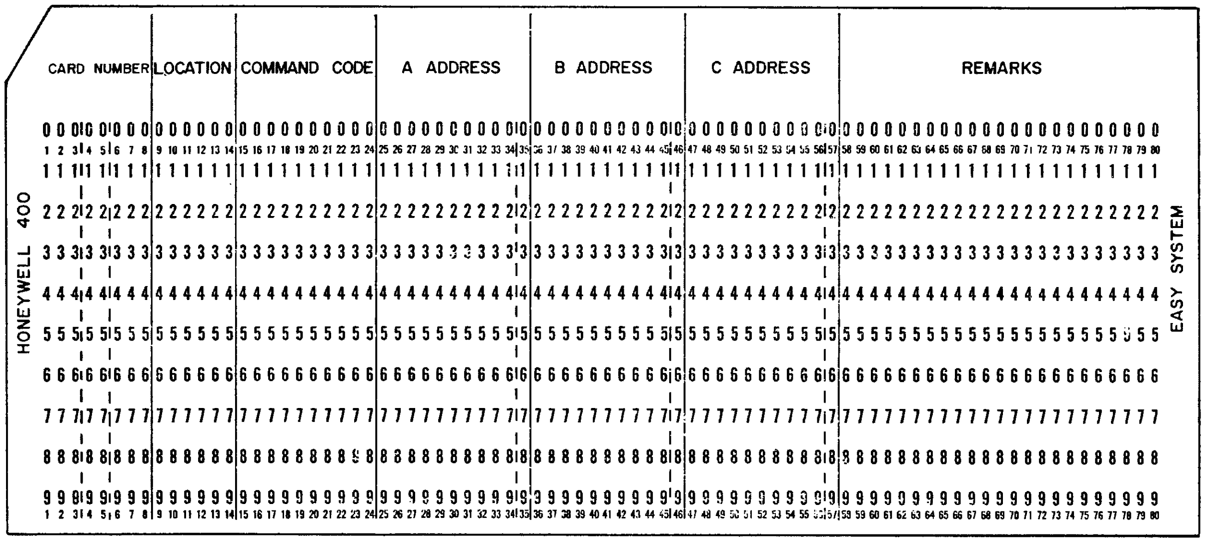


Figure 4. The EASY Input Card

SECTION III

ADDRESSING

Each memory location in the Honeywell 400 has a unique numerical designation (or address). The lowest memory location in a Honeywell 400 system has a numerical address zero (0); the highest memory location has the address 1023, or 2047, or 3071, or 4095,* depending on the memory size of the system.

Addresses written in EASY language are used to designate the high-speed memory locations involved in the execution of an instruction, and are specified in the appropriate address fields. (The addresses of memory locations are also specified in the location field when desired - see below.) Addresses may be direct or indexed (see below) and can be numeric, symbolic, or literal.

Any parameters involved in an instruction are also specified in the address fields of the instruction in EASY language. Indexing of parameter addresses is permitted in certain instructions.

NUMERIC ADDRESSING

There are three types of numeric addressing: absolute addressing, complement addressing, and constant addressing. All are characterized by the fact that the programmer actually specifies the numeric value to be placed in the address field of the machine word. All three types may be indexed.

Absolute Addressing

An absolute address consists of a number of up to four decimal digits in the range 0 through 4095. Integers do not have to be specified with leading zeros.

EASY Assembly translates each absolute address into the equivalent 12-bit configuration and stores this in the corresponding address portion of the machine instruction word; i.e., the A, B, and C address fields are assembled into bits 13 through 24, 25 through 36, and 37 through 48, respectively, of the machine instruction word.

This form of addressing can also be used in the location field. A card containing an absolute address in the location field is assembled and stored in the location specified; subsequent cards

*All numeric addresses in this manual are decimal, except where otherwise stated.

are assembled and stored following the last previous card without an absolute address in the location field. Using absolute addressing in the location field provides an easy means of assigning specified instructions to the reserved memory locations in the Honeywell 400.

Complement Addressing

Complement addressing is a method of specifying memory locations relative to the value 4096. If an absolute address (0 to 4095) is preceded by a minus sign, the address is interpreted as a value to be subtracted from 4096. For example, the address -1 is the same as the address 4095 since 4096 -1 equals 4095..

Complement addresses may not appear in the location field.

Constant Addressing

Addresses can be written in octal, hexadecimal, or alphabetic form by inserting a constant in an address field. For example, an address field may contain

```
0#7777
```

to specify in octal ("0" in the example above) the 12-bit address represented by the characters 7777 (i.e., the address 4095).

Each address-field constant is assembled as a 12-bit address. It must have the following format:

```
M#XX...X
```

where XX...X is the constant and M is mnemonic code which defines the type of constant (see page 67). Any expression that is legal for a mixed constant may also be written in an address field; characters are always left justified.

Constant addressing can not be used in a location field. Complement addressing and address arithmetic (see below) can not be used with constant addressing.

SYMBOLIC ADDRESSING

The use of symbols, rather than numbers, to identify memory locations is called symbolic addressing. There are four types: tag addressing, stopper addressing, current instruction addressing, and storage pool addressing.

Tag Addressing

Tag addressing permits the programmer to refer to a word without knowing its absolute location in memory. Instead, he may identify the word by a mnemonic tag such as "GROSS" or "INPUT".

A tag may consist of up to six alphanumeric characters, one of which must be alphabetic. As used here, the term "alphanumeric" refers to the 26 letters of the alphabet and the 10 decimal numbers. Special characters are acceptable in tags but are reserved for use in the EASY systems programs.

Every symbolic tag which appears in the location field of an EASY input card is assigned an absolute value by Assembly. Each time the tag appears in an address field, this value is substituted for the tag.

If the symbolic tag is preceded by a minus sign, the tag is assigned an absolute value and then this value is subtracted from 4096 as in complement addressing. Minus signs may not be used in the location field.

A tag must not appear in the location field more than once in a segment (see page 55). The same tag may appear in the location field in two or more different segments, unless the segments are common (see page 55).

Address Arithmetic

It is not necessary to tag every word in a program which is to be referenced by some other word in the same program. Address arithmetic allows the programmer to reference an untagged word directly in relation to a tagged one. It is at the programmer's discretion to decide which words in his program to tag; all other words can then be referenced relative to a tagged word.

Address arithmetic is a term which denotes a method of modifying addresses by the addition or subtraction of absolute values to or from symbolic tags. The address modifier consists of a sign and up to four decimal digits and is appended to a symbolic tag to designate a unique memory location relative to the location specified by the tag. Thus, the address

ASSETS + 37

refers to the memory location which is 37 locations beyond that represented by the sumbolic tag ASSETS. Similarly, the address

ASSETS - 12

refers to the memory location which is 12 locations before that represented by the tag ASSETS.

Address arithmetic is permitted in all address fields, but not in the location field. It may be used with absolute addressing and all forms of symbolic addressing.

Address modifiers are not automatically adjusted if coding is inserted or deleted in subsequent updating runs. Care must therefore be taken when address modification occurs in the vicinity of such changes to a program.

Stopper Addressing

The highest location in any system is known as the stopper location and its address is known as the stopper address. This address is 1023, 2047, 3071, or 4095 depending upon the size of the memory in the system. The stopper may be referenced by all forms of legal addressing but EASY II makes a special provision for addressing the stopper.

In the EASY language, the symbol * addresses the stopper location. EASY Assembly substitutes for this symbol the absolute value of the stopper location for the particular system on which the program will be run. Address arithmetic can be used with this symbol as with a tag, provided the modifiers are negative. For example, the address * - 23 refers to the location which is 23 locations before the stopper.

The symbol * must not appear in the location field.

Regardless of the technique used to address it, the stopper location generally has the same characteristics as any other location. However, in the ways listed below, it is unique:

1. The execution of instructions does not continue in the normal sequence once the stopper is reached. Instead the instruction in the stopper location is executed endlessly if it does not call for a change of sequence; and
2. If a series of words is transferred to this location (either beginning with it or arriving at it during the instruction), only the last word in the series is retained; the previous words are lost. This feature is frequently used to bypass unwanted information on tape. For example, if a read tape instruction is given to read a record into * - 3, the first two words are stored in the two locations preceding the stopper; the remaining words are lost except for the last word which is stored in the stopper.

Current Instruction Addressing

The symbol @ permits referencing a word relative to the word containing this symbol. Assembly interprets @ as a reference to the very instruction containing the symbol in an address field. For example, the address @ + 7 refers to the location which is seven locations beyond the instruction containing @ + 7.

Both positive and negative address arithmetic may be used with this symbol. The symbol may appear in any address field but may not appear in a location field.

Storage Pool Addressing

An EASY instruction may address a multi-purpose area called the "storage pool" by using a percent symbol (either octal character 74 or 35). The size of the pool is determined by the largest number used as address arithmetic with the % symbol. For example, if a program's largest reference to the pool is an instruction containing "% + 10" in an address field, ten locations will be reserved for the storage pool.

The storage pool will normally be assigned to locations following all other words in the program but before the literals (see below). If the programmer wishes to store the pool elsewhere, he may place a RESV (see page 59) with a % in the location field anywhere in his program. Assembly will reserve the correct number of locations at that point.

LITERALS

In addition to representing absolute and symbolic addresses, the address fields of EASY instructions may also specify the actual operands required for the execution of instructions. Such values are known as literals.

In using literals, the programmer writes alphanumeric, hexadecimal, octal or decimal characters in an address field. EASY Assembly translates each literal (i.e., the characters in one address field) into equivalent 48-bit machine coding and stores each 48-bit equivalent in an unique memory location. Assembly then places the address of this memory location in the address field containing the original literal. Thus, at execution time, that address field contains a true memory address; that address specifies the location containing the literal.

There are four types of literals which may be specified in an EASY instruction: alphanumeric (defined by the letter "A"), hexadecimal (defined by the letter "H"), octal (defined by the letter "O"), and fixed binary (defined by the letter "F"). A literal is specified in an address field by a mnemonic code which defines the type of literal (i.e., A, H, O, or F); by a right parenthesis symbol ")" which is a separator indicating that the coding is a literal rather than a true address; and by the actual literal.

Assembly eliminates duplicate literals by storing literals with the same 48-bit value in the same location; for this reason, the programmer should not change the contents of a literal location. Assembly stores literals at the end of a segment though the programmer may use the SETLOC instruction (see page 57) to control the assignment of literals.

An attempt to use a literal in an address field representing a result location or a location to which control is transferred will be noted by Assembly as an error.

Alphanumeric Literals

These literals are specified in the address field in the form

A)XX...X

where XX...X represent the alphanumeric characters the programmer desires for the literal. The first eight of these are assembled and stored as eight six-bit characters in machine language. Characters in excess of eight are dropped.

A space or blank is recognized as a valid character and is not suppressed but is assembled and stored in the same manner as any other alphanumeric character.

Hexadecimal Literals

These literals are specified in the address field in the form

H)XX...X

where XX...X represent the hexadecimal characters the programmer desires for the literal. Only the characters 0 through 9 and B through G may appear in a hexadecimal literal. These are assembled and stored as four-bit characters in machine language. Because of the limitation imposed by the size of the address field, a maximum of nine characters can be explicitly defined by the programmer. A hexadecimal literal may be signed, in which case a maximum of eight characters can be defined by the programmer.

If the first "X" is a plus sign (+) or minus sign (-), the sign is assembled (as four binary ones or four binary zeros, respectively) and stored in the high-order position (bits 1-4) of the assembled literal. The remaining explicitly defined characters are right justified in the literal word while the non-specified leading character positions are automatically filled with zeros.

If the literal is not signed, the explicitly defined characters are left justified in the literal word and the non-specified trailing character positions are automatically filled with zeros.

Octal Literals

These literals are specified in the address field in the form

O)XX...X

where XX...X represent the octal characters the programmer desires for the literal. Only the characters 0-7 may appear in an octal literal. These are assembled and stored as three-bit characters in machine language. Because of the limitation imposed by the size of the address field, a maximum of nine characters can be explicitly defined by the programmer. An octal literal may be signed, in which case a maximum of eight characters can be defined by the programmer.

If the first "X" is a plus sign (+) or a minus sign (-), the sign is assembled (as four binary ones or four binary zeros, respectively) and stored in the high-order position (bits 1-4) of the assembled literal. The remaining explicitly defined characters are right justified in the assembled literal, while the non-specified leading character positions are automatically filled with zeros.

If the literal is not signed, the explicitly defined characters are left justified in the literal word and the non-specified trailing character positions are automatically filled with zeros.

Fixed Binary Literals

These literals are specified in the address field in the form

F)XX...X

where XX...X represent the decimal characters (0-9) which are equivalent to the binary configuration desired by the programmer for the literal. The literal that is assembled consists of the 48-bit equivalent of the decimal number specified by these characters. Because of the limitation imposed by the size of the address field, a maximum of nine decimal characters can be explicitly defined by the programmer. A fixed binary literal may be signed, in which case a maximum of eight decimal characters can be defined by the programmer. Unsigned fixed binary literals are assumed to be positive and the plus sign is assembled. Unless "B" positioning (see below) is employed, the fixed binary literal is assembled such that the binary configuration is always right justified in the word; i.e., the unit bit of this configuration is in the low-order position of the word and higher power bits are in consecutive positions to the left, irrespective of whether the literal is signed or not signed.

The programmer has the option to position the binary configuration within the literal word wherever he desires by the use of "B" positioning. If the decimal characters are followed by the letter "B" and then a number in the range 5 through 48, then the unit bit of the binary configuration is stored in the position specified by that number within the literal word, and higher power bits are stored in consecutive positions to its left. The position in the literal word are numbered so that position 1 is the high-order position and position 48 is the low-order position; i.e., the positions are numbered left to right from 1 through 48. For example, if the literal representation:

F)45B24

is specified in an address field, the binary equivalent of decimal 45 is assembled and stored in the literal word such that the unit bit of the configuration is in position 24.

Although "B" positioning is allowed, decimal points are not.

INDEXED ADDRESSING

Indexing is a simple method of increasing the flexibility of instruction addressing and also provides a convenient technique for storing and modifying counters. All forms of legal addressing except literals may be used in conjunction with the three index registers; instruction addresses containing parameters (see below) may also, in general, be indexed. The restrictions on indexing parameter addresses are given with the appropriate instructions in Section IV.

Each of the three index registers in the Honeywell 400 stores 12 binary digits which represent a value in the range 0 through 4095. This value can be used as a memory address or as a counter to be incremented or modified upon the occurrence of certain conditions.

When an instruction address is indexed, it is temporarily added to the contents of the specified index register to form an effective working address (regardless of the particular meaning of this address); this working address is then used to locate the operand or store the result of the instruction, or to specify the parameters required for the execution of the instruction. The address in the machine instruction word and the contents of the index register remain unchanged upon completion of the instruction.

If an address is to be indexed, the index register indication is specified in the corresponding address field following the address (or parameter) and separated from it by a comma. The index registers are specified by the decimal numbers 1, 2, and 3, or by symbolic tags. If the index register indication is a tag, this tag must somewhere be defined by the EQUALS control instruction (see Section V) to be equal to the number of the required index register (see example below). A blank is interpreted as no indexing. Any indication in the index register position other than zero (interpreted as no indexing) or one of the valid digits (1, 2, or 3), or an assigned tag is automatically diagnosed by EASY Assembly as an error.

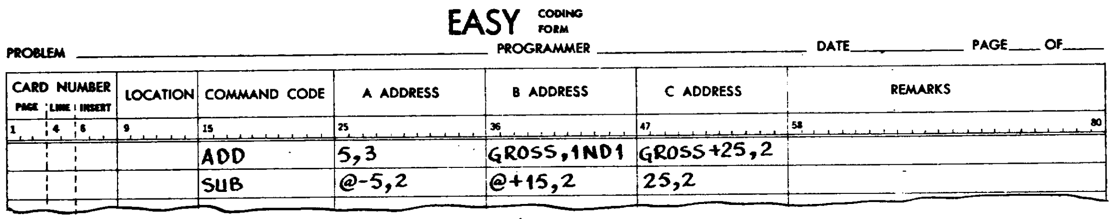
As stated above, indexing is permitted with any form of legal address (i.e., absolute, symbolic, or relative); it is also, in some cases, permitted with parameter addresses. The instructions which do not allow indexing of parameter values are those which use the B index register field to define the operation code and those in which it would be of no value to index the particular parameters in the instructions (e.g., the B address of the SCO instruction).

It is possible to index in such a way that the highest system address (i.e., the stopper address) is exceeded. At instruction time, the generated address is interpreted modulo the memory size (the actual address stored with the instruction in memory is unchanged by indexing). The address derived from this process is lower than the indexed one, and so the stopper address has been bypassed.

This process is possible in all systems, irrespective of memory size. In a system with a 3072-word memory, however, the indexing must result in an effective address greater than 4095; this address is then modified modulo 4095 to give the working effective address.

Each of the index register indications is assembled as two bits: the index register bits corresponding to the index register specified in the A address are stored in bit positions 7 and 8 (field Ai) of the machine instruction word; those for the B address indication are stored in bit positions 9 and 10 (field Bi); and those for the C address indication are stored in bit positions 11 and 12 (field Ci). The configurations corresponding to each index register are given on page 2. If indexing is attempted in the B address of instructions which use bits 9 and 10 of the machine instruction word to define the operation code, Assembly automatically diagnoses an error.

An index register is frequently used to store the base address of some processing area, while the address portions of the instructions designate specific words within that area. It is only necessary to alter the contents of the index register to move on and process a similar area without having to change the instructions.



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE 1	LINE 4	INSERT 6	LOCATION 9	COMMAND CODE 15	A ADDRESS 25	B ADDRESS 36	C ADDRESS 47	REMARKS 58 … 80
				ADD	5,3	GROSS,IND1	GROSS+25,2	
				SUB	@-5,2	@+15,2	25,2	

In the A address of the above ADD instruction, index register 3 contains the base address of a processing area. Similarly, the index registers indicated in the B and C addresses contain the base addresses of other processing areas. When the instruction is executed, the effective working addresses are formed as follows.

The value 5 is added to the contents of index register 3. The effective working address thus refers to the sixth word in the processing area which begins at the location whose address is stored in index register 3. Neither the value 5 nor the contents of the index register are disturbed by this process.

The address of the location tagged GROSS is added to the contents of the index register defined by the tag INDI to form the effective working address of the second operand in the ADD instruction. Neither the address of the location tagged GROSS nor the contents of index register INDI are disturbed by this process. The ADD instruction adds the contents of the location specified by the two working addresses and stores the result in the location whose address is formed by adding 25 to the address assigned to the location tagged GROSS and then adding this to the contents of index register 2.

Note that the tag INDI must previously have been assigned to be equal to one of the numbers 1, 2, or 3 by an EQUALS instruction (see Section V).

The addresses in the SUB instruction are interpreted in a similar fashion to those of the ADD instruction.

PARAMETERS

Address fields specified in EASY language may contain information other than memory addresses. Such information may include, for example, the number of words to be transferred, the number of positions to shift an operand, the number of characters to be edited, or the number of words to be written on magnetic tape. These values, known as parameters, are specified in the B address fields of the appropriate instructions as either absolute or symbolic values. Address fields specifying parameter values are assembled and stored as 12 bits in the corresponding address fields in the machine instruction words. Individual parameters are represented by a fixed number of bits out of the 12 and are limited by the values which can be represented by these bits. If a parameter specifies a number greater than that which can be represented by its assigned number of bits, the value is reduced modulo the maximum value of that parameter and an error is noted. For example, if a parameter is represented in the machine by four bits and is specified by a value of 17, the value which is assembled is 17 converted modulo 16; i.e., 1. Values of parameters within the limits of the machine fields but greater than the functional maximum for the instruction are assembled normally and an error is noted. In instructions where parameters are recognized by the Assembly program, any omission of leading parameters in an address must be indicated by a leading zero followed by a comma, or simply by a leading comma (see "Print and Space", Section IV). Trailing parameters may be omitted without any special indication.

In the machine words of certain instructions which contain parameter values in the B address, the corresponding index register field (i.e., bits 9 and 10) is used as part of the operation code (see Section II), and so indexing of that address is illegal. Any attempt to index such an address is automatically diagnosed by EASY as an error. In some other instructions, indexing of parameter value addresses is not permitted even though the corresponding index register field is not used as part of the operation code. Any restriction on indexing a particular instruction is noted in the appropriate place in Section IV.

UNUSED ADDRESS FIELDS AND INDEX BITS

All address fields that are interpreted by the central processor must contain valid EASY addresses or parameters. EASY Assembly converts blank address fields to 12 binary zeros; no error is indicated.

Some address fields are not used in certain instructions but must be set to zero. If any address form is written in such a field, it will be assembled as zero and an error indicated. This type of instruction is identified in Section IV and the summary in Appendix D.

In other instructions, the address fields are not interpreted by the central processor during execution and may contain any legal address form. Assembly interprets such data normally; if the field is blank, it will be assembled as all zeros. These instructions are identified in Section IV and Appendix D.

Similar remarks apply to the index bits. In certain instructions (see Appendix D), these bits can store data irrelevant to the instruction. This is possible in instructions in which an address field can not be indexed and the associated bits are not part of the operation code; in this case, data in the indexing position of an EASY instruction will be assembled with an error notation to warn the programmer that the field will not be indexed at execution time. However, if an attempt is made to index an address field which can not be indexed because the associated bits are part of the operation code, the indexing attempt will be ignored and an error will be noted.

SECTION IV

MACHINE INSTRUCTIONS

Honeywell 400 machine instructions are specified in EASY language using mnemonic operation codes. These codes are written in the command code fields to designate the type of instruction to be performed on the information referenced in the address fields.

EASY Assembly translates each machine instruction into a 48-bit machine instruction word. The four 12-bit groups of a machine instruction word (i.e., command code group, and three address groups) do not necessarily correspond to the respective four groups as specified in EASY language. The most notable difference is in indexing; EASY language requires the index register indication to be specified in the appropriate address field, whereas the index register bits are stored in the command code group in the machine word. The programmer need not be aware of this difference in arrangement unless he wants to modify machine instructions at the time of program execution. Instructions which perform any such modification must refer to the proper bit positions in the machine word. (A detailed description of the machine language format of Honeywell 400 instructions is contained in the Honeywell 400 General Information Manual; a summary chart is in Appendix D of this manual.)

EASY recognizes the following eight classes of machine instructions:

1. Arithmetic;
2. Logical;
3. Shift;
4. Decision;
5. Transfer and Sequence Change;
6. Edit;
7. Peripheral; and
8. Index and Check.

The function and format of each instruction are described in this section under the appropriate class.

ARITHMETIC INSTRUCTIONS

The Honeywell 400 performs both decimal and binary arithmetic. In decimal arithmetic, each operand is assumed to consist of 11 decimal digits and a sign (in binary form), and normal algebraic rules are observed with respect to the sign; i.e., a decimal add instruction causes operands with like signs to be added, and operands with unlike signs to be subtracted, with the result adopting the sign of the larger absolute number. Similarly, a decimal subtract instruction causes operands with like signs to be subtracted and operands with unlike signs to be added. In binary arithmetic, the operands are treated as unsigned quantities. Binary arithmetic is used primarily for instruction modification.

In installations equipped with the Multiply-Divide Option, multiple and divide instructions are assembled, interpreted, and executed in the same way as any other machine instruction. A machine not equipped with this option performs multiplication and division by the use of library routines.

Decimal Add ADD/A/B/C

This instruction adds the signed 11 decimal digit word at A algebraically to the signed 11 decimal digit word at B and stores the sum in the location specified by C. If the addition results in overflow, an unprogrammed subsequence to the reserved memory location 0017 is executed.

Decimal Subtract SUB/A/B/C

This instruction subtracts the signed 11 decimal digit word at B algebraically from the signed 11 decimal digit word at A and stores the difference in the location specified by C. If the subtraction results in overflow, an unprogrammed subsequence to memory location 0017 is executed.

Decimal Multiply MPY/A/B/C

This instruction multiplies the signed 11 decimal digit word at A by the signed 11 decimal digit word at B. The high-order signed 11 decimal digits of the result are stored in the location specified by C. The low-order signed 11 decimal digits of the result are stored in the fixed memory location 0000, known as the low-order product word (or LOP). Multiplication of operands with like signs results in a positive product; with unlike signs, in a negative product. The operands are treated as decimal fractions; i. e., the decimal point is assumed to be immediately to the left of the high-order digit in each operand. The positions of the significant digits in the two halves of the result reflect this convention. The examples below illustrate these points.

EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE LINE INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	9	15	25	36	47	58 80
		MPY	MULTA	MULTB	PRODC	

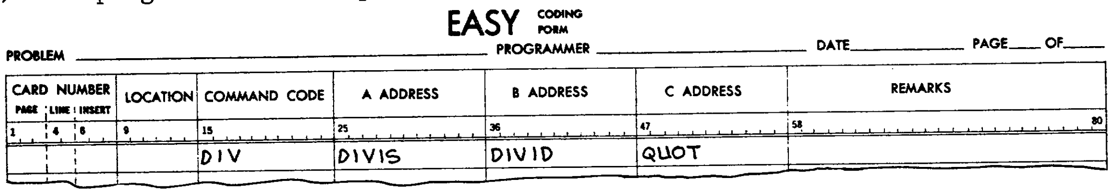
If the contents of the memory locations tagged MULTA, and MULTB are as shown, the contents of the memory location tagged PRODC and LOP after execution of the instruction are as given:

1.	Multiplicand	MULTA	+00000000005
	Multiplier	MULTB	+00000000050
	High-Order Product	PRODC	+00000000000
	Low-Order Product	LOP	+00000000250
2.	Multiplicand	MULTA	+50000000000
	Multiplier	MULTB	-05000000000
	High-Order Product	PRODC	-02500000000
	Low-Order Product	LOP	-00000000000

<u>Decimal Divide</u> DIV/A/B/C

This instruction divides the signed 11 decimal digit word at B (dividend) by the signed 11 decimal digit word at A (divisor). The signed 11 decimal digit quotient is stored in the location specified by C. The signed 11 decimal digit remainder is stored in the reserved memory location 0010, known as the remainder word. Division of operands with like signs results in a positive quotient; with unlike signs, in a negative quotient. The remainder takes the sign of the dividend. Since the operands are treated as decimal fractions, the positions of the significant digits in the two halves of the result reflect this convention. If the A and C addresses are the same, the result is unspecified; the instruction operates normally if the B and C addresses are the same.

If the absolute value of the dividend is greater than or equal to the absolute value of the divisor, an unprogrammed subsequence to memory location 0017 is executed.


EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE ¦ LINE ¦ INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 ¦ 4 ¦ 6	9	15	25	36	47	58 80
		DIV	DIVIS	DIVID	QUOT	

If the contents of the memory locations tagged DIVID and DIVIS are as shown, then the contents of the memory location tagged QUOT and the remainder word after the execution of the instruction are given below:

1.	Dividend	DIVID	+50000000000
	Divisor	DIVIS	-60000000000
	Quotient	QUOT	-83333333333
	Remainder	REM	+20000000000
2.	Dividend	DIVID	-10000000000
	Divisor	DIVIS	+30000000000
	Quotient	QUOT	-33333333333
	Remainder	REM	-10000000000
3.	Dividend	DIVID	-00000000005
	Divisor	DIVIS	-00000000060
	Quotient	QUOT	+08333333333
	Remainder	REM	-00000000020
4.	Dividend	DIVID	+00004000000
	Divisor	DIVIS	+30000000000
	Quotient	QUOT	+00013333333
	Remainder	REM	+10000000000

All of these examples illustrate the rule of zeros in the quotient. The rules of zeros states that the number of leading zeros in the quotient is equal to the difference between the number of leading zeros in the dividend and the divisor. The term "leading zeros" refers to those zeros appearing between the sign and the first significant digit. This digit is itself a zero in a dividend in which the first non-zero digit is greater than the first non-zero digit of the divisor (see example 4 above).

The examples also illustrate that the remainder is formed exactly as it would be in ordinary arithmetic and that the first non-zero digit of the remainder is in the same digit position as the first non-zero digit of the divisor.

Binary Add BAD/A/B/C

This instruction adds the unsigned binary quantity at A to the unsigned binary quantity at B and stores the sum in the location specified by C. Any overflow is disregarded.

Binary Subtract BSU/A/B/C

This instruction subtracts the unsigned binary quantity at B from the unsigned binary quantity at A and stores the difference in the location specified by C. Any overflow is disregarded; if the B operand is greater than the A operand, the result is the 2's complement of the absolute value of the difference.

LOGICAL INSTRUCTIONS

The four logical instructions manipulate words on an individual bit basis, combining bits from two words to form a third. All operands are treated as 48-bit words in which each bit is unrelated to any other bit. There are no restrictions on indexing any address of a logical instruction.

Two of the logical instructions, extract and substitute, use the concept of masking. In both cases, the B address field specifies either a mask or the location of a mask. A mask is defined as a particular configuration of binary ones and zeros which, when combined with another operand (specified by A) according to prescribed rules, passes into the result word only those bits of that operand which correspond to binary ones in the mask. The remaining portions of the result word correspond to the binary zeros in the mask. In the extract instruction, these remaining bit positions are cleared to zero, while in the substitute instruction, they are left unchanged.

There are two ways of designating masks; both, however, result in 48-bit configurations. The first method specifies a location in the desired address field; in general, the location thus referenced contains a 48-bit data constant (see Section VI), though there is no reason why it could not contain a machine instruction. The second method of specifying masks permits the programmer to define the actual mask he requires by writing its EASY language representation in the required address field (see "Literals", Section III). This again results in a 48-bit configuration. Both methods are illustrated in the examples given with the definitions of the instructions themselves (see below).

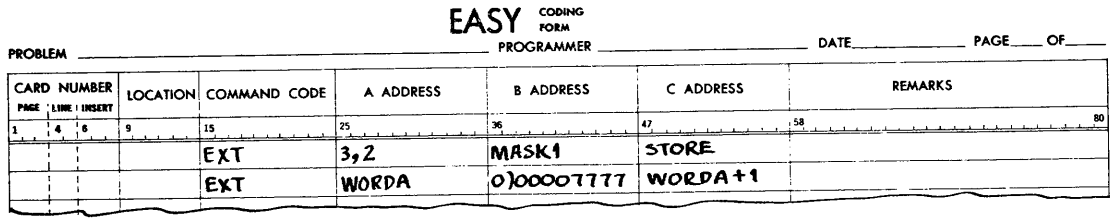
The substitute instruction is used, in general, to "pack" information fields of different types and lengths into a single memory location, while the extract instruction can be used to "unpack" these fields whenever required.

<u>Extract</u> EXT/A/B/C

This instruction passes the word in the location specified by A through the mask specified in the B address and places the result in the location specified by C. The portions of the result word corresponding to the unmasked portions of the A operand are set to zero.

This is equivalent to combining the corresponding bits of the A and B operands according to the following rule:

> If corresponding bit positions in the word at A and the word at B both contain binary ones, the result contains a 1 in this position. In all other cases, the result contains a binary 0.



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE ¦ LINE ¦ INSERT (1 ¦ 4 ¦ 6)	LOCATION (9)	COMMAND CODE (15)	A ADDRESS (25)	B ADDRESS (36)	C ADDRESS (47)	REMARKS (58–80)
		EXT	3,2	MASK1	STORE	
		EXT	WORDA	0)00007777	WORDA+1	

The first instruction combines the word in the location whose address is three greater than the address stored in index register 2 with the contents of the location tagged MASK (i. e., the mask required), according to the extract rule. The result is stored in the location tagged STORE.

The second instruction combines the word in the location tagged WORDA with the 48-bit configuration defined by the literal in the B address (i. e., with the binary equivalent of octal 0000777700000000, since the literal is left justified in its location), according to the extract rule. The result is stored in the location whose address is one greater than that of the location tagged WORDA.

Substitute SST/A/B/C

This instruction passes the word specified by A through the mask specified in the B address and places the result in the location specified by C. In contrast to the extract instruction, the portions of the result word corresponding to the unmasked portions of the A operand are left unchanged by the instruction.

This is equivalent to combining the corresponding bits of the A and B operands according to the following rule:

> Wherever the B operand contains a binary 1, the corresponding bit of the A operand is stored in the corresponding bit position of the result word. All other positions in the result word (specified by C) are unaffected.

EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE ¦ LINE ¦ INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 ¦ 4 ¦ 6	9	15	25	36	47	58 … 80
		SST	3,2	MASK1	STORE	
		SST	WORDA	0)00007777	WORDA+1	

These two instructions perform the same functions as described under extract, except that the operands are combined according to the substitute rule rather than the extract rule.

Half Add HAD/A/B/C

This instruction adds the contents of the locations specified by A and B in binary fashion without carries and stores the result in the location specified by C.

This is equivalent to combining the corresponding bits of the A and B operands according to the following rule:

> If the corresponding bit positions in the A and B operands have the same value, the result contains a binary 0 in this position. In all other cases, the result contains a binary 1.

Superimpose SMP/A/B/C

This instruction superimposes the 48-bit A and B operands in such a way that the result word contains a binary 1 in every position in which a 1 existed in either A or B or both, and the result is stored in the location specified by C.

This is equivalent to combining the corresponding bits of the A and B operands according to the following rule:

> If the corresponding bit positions of the A and B operands both contain binary zeros, the result contains a binary 0 in this position. In all other cases, the result contains a binary 1.

SHIFT INSTRUCTIONS

In shift instructions, the word specified by the A address is shifted to the left or right, cyclically or non-cyclically depending on the instruction, a number of decimal or binary digits as specified by the programmer. The decimal shift instructions operate on the 11 digits of the word at A, shifting these non-cyclically up to 11 places to the left or right; the sign digit is not affected. The binary shift instruction operates on all 48 bits of the word, shifting them cyclically up to 48 places (moving left).

The shift instructions thus provide a means of manipulating data fields within a single word. A shift instruction could, for example, be used in conjunction with a substitute or extract instruction (see above) to pack or unpack a word in the required manner and format.

The parameter n in the B address of these instructions may be of any legal address format except literal. If it is symbolic, the symbol should be equated to an integer using an EQUALS instruction.

Decimal Shift Right, Preserving Sign SRP/A/n/C

This instruction shifts the word at A, excluding the sign, n ($0 \leq n \leq 11$) decimal digit positions to the right. All digits shifted out at the right (or low-order) end of the word are lost, and a corresponding number of zeros are added at the left of the word, immediately following the sign. The resulting word is stored in the location specified by C. The sign supplied with the result will be four zeros if the sign digit of the A operand was all zeros. Otherwise, it will be four ones.

Decimal Shift Left, Preserving Sign SLP/A/n/C

This instruction shifts the word at A, excluding the sign, n ($0 \leq n \leq 11$) decimal digits to the left. Digits shifted out of the 11th (i. e., second digit) high-order position are lost and are replaced by the corresponding number of zeros at the right (or low-order) end of the word. The resulting word is stored in the location specified by C. The sign delivered with the result is the same as in the case of the decimal shift right instruction.

Binary Shift Left SLB/A/n/C

This instruction shifts the 48 bits of the word at A left n bit positions in a cyclic fashion. That is, the bits shifted out at the left (or high-order) end of the word re-enter the word at the right in the order in which they were shifted out. The resulting word is stored in the location specified by C. If n is greater than 48, it is reduced modulo 48 at execution time.

DECISION INSTRUCTIONS

In decision instructions, the word at A is compared with the word at B to determine whether a particular relationship exists between them. If the specified relationship between the operands is met, the sequence register is set to the address specified by C, and the program branches to an alternate path. If the relationship is not met, the sequence register is incremented by unity, and the next instruction in sequence is executed. There are no restrictions on indexing any address in a decision instruction.

Inequality Comparison, Alphanumeric NAC/A/B/C

This instruction compares the word at A with the word at B, bit-by-bit, for inequality. If the two words are not identical, the sequence register is changed to select the next instruction from the location whose address is specified by C. If the operands are equal (i. e., the inequality relationship is not met), the next instruction in sequence is executed.

Inequality Comparison, Numeric NNC/A/B/C

This instruction compares the signed 11 decimal digit word at A for inequality with the signed 11 decimal digit word at B. If the two words are not equal, the sequence register is set to the address specified by C; if the two words are equal, the next instruction in sequence is executed. The high-order four bits (i. e., the sign bits) are compared as a unit, in contrast to the manner in which the digits are compared. Since any non-zero configuration in the four sign bits is interpreted as a plus sign, the two operands could be considered equal although having unequal sign bits. Plus zero and minus zero are considered equal.

Less than or Equal Comparison, Alphanumeric LAC/A/B/C

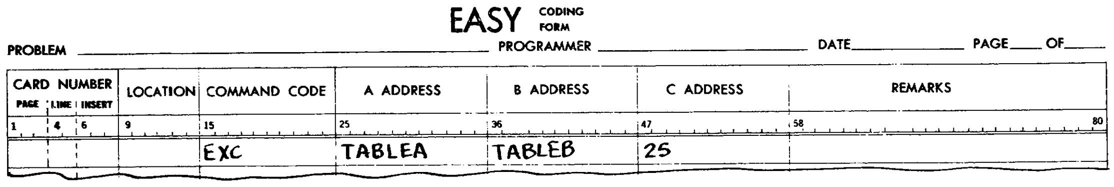
This instruction compares the 48-bit word at A, bit-by-bit, with the 48-bit word at B. If the word at A is less than or equal to the word at B, the sequence register is set to the address specified by C. If this relationship is not satisfied (i. e., if A is greater than B), processing continues in sequence.

Less than or Equal Comparison, Numeric LNC/A/B/C

This instruction compares the signed 11 decimal digit word at A with the signed 11 decimal digit word at B. If the word at A is algebraically less than or equal to the word at B, the sequence register is set to the address specified by C; otherwise, processing continues in sequence. Apart from plus zero being considered equal to minus zero, any positive number is greater than any negative number.

Extended Comparison EXC/A/B/C

Starting with the word at A, this instruction compares consecutive words with a corresponding number of consecutive words starting with the word at B. The comparison is performed on a bit-by-bit basis, since the operands are considered as unsigned 48-bit words. The comparison terminates as soon as a difference between corresponding A and B operands is found. If the "A" word is less than the "B" word, the sequence register is set to the address specified by C; if the "A" word is greater than the "B" word, processing continues in sequence. The instruction terminates if and only if a difference between the A and B operands is found. (Note: If this does not occur before one of the comparisons attempts to use the stopper location, the contents of this location are used as one of the operands for all successive comparisons.) At the termination of the instruction, the value of A + n (where n is the number of consecutive comparisons made and found equal) is stored in memory location 0013, called the termination address word (see Appendix A).



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE LINE INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	9	15	25	36	47	58 80
		EXC	TABLEA	TABLEB	25	

Let the location tagged TABLEA and TABLEB be the starting locations of two tables in memory which contain the following values:

Location	Contents	Location	Contents
TABLEA	17	TABLEB	17
TABLEA + 1	18	TABLEB + 1	18
TABLEA + 2	14	TABLEB + 2	14
TABLEA + 3	12	TABLEB + 3	13
TABLEA + 4	16	TABLEB + 4	16

The instruction in the example above compares the contents of the location tagged TABLEA with the contents of the location tagged TABLEB. Since these are equal, the instruction then compares the contents of the location one beyond that tagged TABLEA with the contents of the location one beyond that tagged TABLEB. Since these are also equal, the instruction continues. Since the contents of the location three locations beyond that tagged TABLEA (i.e., TABLEA + 3) are less than the contents of the location three locations beyond that tagged TABLEB (i.e., TABLEB + 3), the sequence register is set to the value 25. The instruction thus terminates after four pairs of words have been compared, and the value of TABLEA + 3 (i.e., the value three greater than the address of the location tagged TABLEA) is stored in the termination address word.

TRANSFER AND SEQUENCE CHANGE INSTRUCTIONS

There are eight instructions for transferring words in memory and changing the program path. Five of these instructions cause changes in the sequence of instruction execution; of these, one instruction also transfers one word to a different location, another causes a single instruction to be executed out of sequence without altering the contents of the sequence register, and a third performs no other operation save incrementing the sequence register normally (i. e., by unity). A sixth instruction transfers a number of consecutive words in memory to different consecutive locations; a seventh instruction has the ability to stop the computer; the eighth instruction stalls the central processor during an acceleration period.

Transfer and Sequence Change TSC/A/B/C

This instruction transfers the word at A to the location specified by B, and sets the sequence register to the address specified by C. The sequence register is changed before the transfer.

Transfer n Words TSN/A/n/C

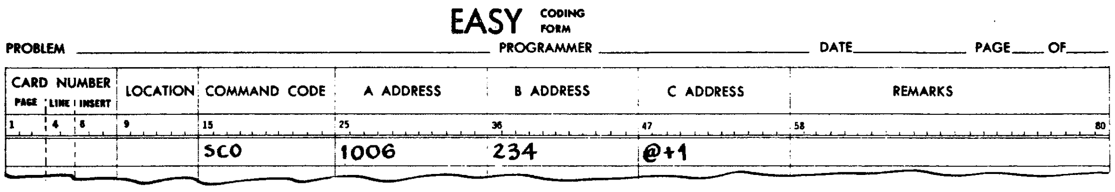
This instruction transfers the number of words specified in the B address (i. e., n) from consecutive memory locations starting at the location specified by A to consecutive memory locations starting at the location specified by C. The parameter n may range in value from 0 through 4095; and may be of legal address format except literal. If it is symbolic, the symbol should be equated to an integer using the EQUALS instruction. Processing continues in sequence after this instruction.

Sequence Change SCH/ / /C

This instruction sets the sequence register to the address specified by C. The A and B addresses are not used but are assembled; therefore they may contain any legal address format.

Sequence Change on Option SCO/A/n/C

This instruction changes the sequence of instruction execution depending on the setting of the four breakpoint switches on the operator's console. If any of the values which n can take coincide with the number of a console breakpoint switch set to the "ON" position, then the sequence register is set to the address specified by A. If there is no such coincidence, the sequence register is set to the address specified by C. The parameter n can consist of any one or all four of the digits 1, 2, 3, 4, any combination of two or three of these digits, or a symbolic tag. The value (or values) desired for n are written in any order (e.g., 312) in the B address field; a tag should be defined by an EQUALS instruction. Indexing is not permitted in the B address of this instruction.



EASY CODING FORM

PROBLEM ____ PROGRAMMER ____ DATE ____ PAGE ___ OF ___

CARD NUMBER PAGE | LINE | INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	9	15	25	36	47	58 80
		SCO	1006	234	@+1	

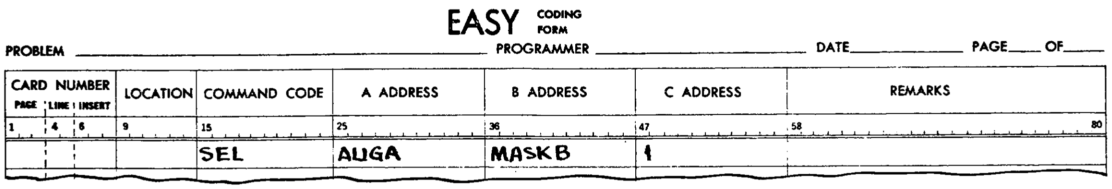
The instruction in the example above interrogates the breakpoint switches 2, 3, and 4 on the operator's console. If any of these is set to the "ON" position, the sequence register is set to the value 1006; otherwise, the sequence register is set to the address one beyond the address of the location containing the current instruction; i. e., the sequence does not change (since this is the normal setting of the sequence register for the next instruction).

<u>Select</u> SEL/A/B/C

Execution of the select instruction results in a programmed subsequence to a location whose address is determined as described below; its execution leaves the sequence register unchanged. If the selected instruction is not another subsequence call instruction or a sequence change instruction, control returns to the address specified in the sequence register upon execution of the selected instruction.

The address of the one instruction to be executed in the subsequence mode is determined as follows:

1. The C address of the select instruction, as written by the programmer, contains the base address;
2. If the C address is indexed, the contents of the referenced index register augment the base address to form the effective C address; and
3. A second augmenter, determined by the A and B addresses of the select instruction, is formed and added to the effective C address. The four 12-bit groups of the word at A are combined with the four 12-bit groups of the word at B according to the extract rule. (See appropriate instruction above.) The four extracted 12-bit groups are superimposed, and the resulting 12 bits are binary added to the effective C address. In general, the configuration of the word at B is such that it will block three of the four 12-bit groups in the word at A and select from 1 to 12 bits of the remaining group as an augmenter to the C address. The definition of the instruction, however, allows a more sophisticated use; namely, the selection of a configuration of the word at B which would provide for superimposing one or more bits from each of the four 12-bit groups to form the augmenter.


EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE______ PAGE____ OF____

CARD NUMBER PAGE | LINE | INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 | 4 | 6	9	15	25	36	47	58 … 80
		SEL	AUGA	MASKB	1	

Let the contents of the locations tagged AUGA and MASKB be as follows:

AUGA	000000001111	000011110000	000111110100	000000000001
MASKB	111111111111	000000000000	000000000000	000000000000

When the SEL instruction above is executed, the augment produced by superimposing the four 12-bit groups obtained by extracting AUGA through MASKB is 000000001111.

This address is temporarily added to the effective C address. This produces the address 000000010000 which is the address of the location containing the selected instruction.

The C address of this instruction is unchanged at the completion of the instruction.

No Operation NOP/ / /

This instruction causes the system to proceed to the next instruction without performing any other action. Thus, it normally has the effect of simply incrementing the sequence counter; however, if executed as the result of a subsequence call (e.g., as the instruction stored in a reserved memory location), it simply returns control to the sequence counter. The address fields and the Bi field are ignored (see "Unused Address Fields and Index Bits", Section III).

Halt HLT/ /n/

This instruction stops the central processor if n is a blank or if any of the other values which n can take coincide with the number of a console breakpoint switch set to the "ON" position. In addition to a blank, n can consist of any one of the digits 1, 2, 3, 4, any combination of two, three or all four of these digits, or a symbolic tag. The value (or values) desired for this parameter are written in any order (e.g., 412) in the B address; a tag should be defined by an EQUALS instruction. The sequence register is incremented normally (i.e., by unity). In addition, n may take the value 0; in this case, when the instruction is executed, the sequence register is incremented normally, the central processor does not halt, and instruction execution proceeds as usual. Indexing is not permitted in the B address of this instruction. The A address field must be left blank. The C address field is not used but is assembled normally and may contain any legal address format.

Stall, SUP SUP/A/ /

If selected during the acceleration interval of a read card or punch card instruction, this instruction stalls the central processor until completion of data transfer and then allows processing to continue normally. If selected outside the acceleration interval, it acts as a no operation instruction. Thus, this instruction is useful in delaying execution of an instruction which might not be completed during the acceleration interval of the reader or punch.

The A address field must indicate, in any legal address format, the address of the SUP instruction itself. The B address must be left blank. The C address is not used but is assembled; therefore it may contain any legal address format.

EDIT INSTRUCTIONS

There are four input edit instructions which handle the information from punched cards, four output edit instructions which handle information to be punched on cards, and a further three output edit instructions which handle information to be printed. In addition, a twelfth instruction prepares a signed decimal word for editing and subsequent printing or punching. This last instruction allows convenient modification of information (such as the positioning of special characters). The edit instructions can edit fields from and to alphanumeric, octal, and signed and unsigned decimal format.

The eight edit instructions which manipulate card information fields do so in a way which enables the programmer to work with characters rather than words. The four card edit instructions convert a specified number of card columns from the card image form in the card read area (see "Reserved Memory Locations", Section I) to the appropriate internal forms and store the converted information starting in the high-speed memory location specified by the programmer. Similarly, the four punch edit instructions convert information from internal form to card image form and store this information in the card punch area in columns specified by the programmer.

The three print edit instructions convert information from internal form to the form required for printing and store this information in the print area as specified by the programmer.

All edit instructions specify a high-speed memory location in the A address field; that is, in input editing, the initial storage location after editing, and in output editing, the source of the data to be edited. The B address field designates the position of the first (i.e., leftmost) edited character in the word specified by A and also the number of consecutive characters to be edited. The C address field specifies the relative position in the card or print storage area where the first character to be edited is obtained (input editing) or stored (output editing).

Except for signed decimal, all edit instructions operate on the specified number of characters; word boundaries are ignored. The signed decimal edit operates on a single word only. None of the 11 edit instructions nor the prepare decimal edit instructions can be indexed in the B address; all can be indexed in the A and C address fields.

In all of the edit instructions, the parameters in the B address field may be absolute or symbolic; symbolic parameters should be defined by an EQUALS instruction.

In the following descriptions of the edit instructions, the normal maximum values of the parameters are given in parentheses. Generally, these maximums are merely the limits imposed by the size of the image areas; for example, C in the card edit instructions designates a column position relative to the base of the image area and therefore does not usually exceed 80 (which designates the last such column in the reserved image area). However, values beyond these maximums may be specified and assembled normally (though a notation indicating a possible error will appear in the listing). If such values also exceed the machine fields in which they will be represented, they are assembled modulo their limiting values: n is assembled modulo 256; C is assembled modulo 4096; m is assembled modulo 16. Thus, the results of exceeding the maximums given below are as follows:

1. If n is greater than the normal limit of the image area (80 for card editing, 120 for print editing), editing continues outside the image area to a maximum of 255 characters.

2. If C is greater than its normal limit (80 for card editing, 120 for printing), editing starts outside the image area at a card column or print position located relative to the base address of the image area. For example, if C 81 in a card edit instruction, editing starts in bits 1-12 of the first location beyond the image area. The following formulas are helpful in determining the value that should be specified in C to begin editing at the high-order 12 bits of a certain location:

   Card read edit - $C = 4(W-54)$

   Card punch edit - $C = 4(W-74)$

   Print edit - $C = 8(W-39)$

   where W is the address of the location at which editing will start. For example, to start editing at location 354, the C field of a card edit instruction should have effective value of 1200:

   $$C = 4(354 - 54)$$

   The value of 1, 200 may be specified in the C address field or it may be obtained by indexing.

3. If m is greater than its normal limit (8 for alphabetic editing, 11 or 12 for decimal editing, and 16 for octal editing), m is interpreted as follows:

M	Octal Edit	Alphabetic Edit	Decimal Edit
1 (0001)	1	1	1
2 (0010)	2	2	2
3 (0011)	3	3	3
4 (0100)	4	4	4
5 (0101)	5	5	5
6 (0110)	6	6	6
7 (0111)	7	7	7
8 (1000)	8	8	8
9 (1001)	9	1	9
10 (1010)	10	2	10
11 (1011)	11	3	11
12 (1100)	12	4	12
13 (1101)	13	5	11
14 (1110)	14	6	12
15 (1111)	15	7	12
16 (0000)	16	8	1

<u>Card Edit, Alphanumeric</u> ECA/A/m, n/C

This instruction edits n consecutive characters of alphanumeric data and stores the edited data in memory beginning with the mth ($1 \leq m \leq 8$) alphanumeric character position of the word at A. The input data are obtained from consecutive column representations in the card storage area beginning with the column specified by C ($1 \leq C \leq 80$). Detection of an illegal punch configuration results in an unprogrammed subsequence to memory location 0022 (see Appendix A).

Card Edit, Unsigned Decimal ECU/A/m, n/C

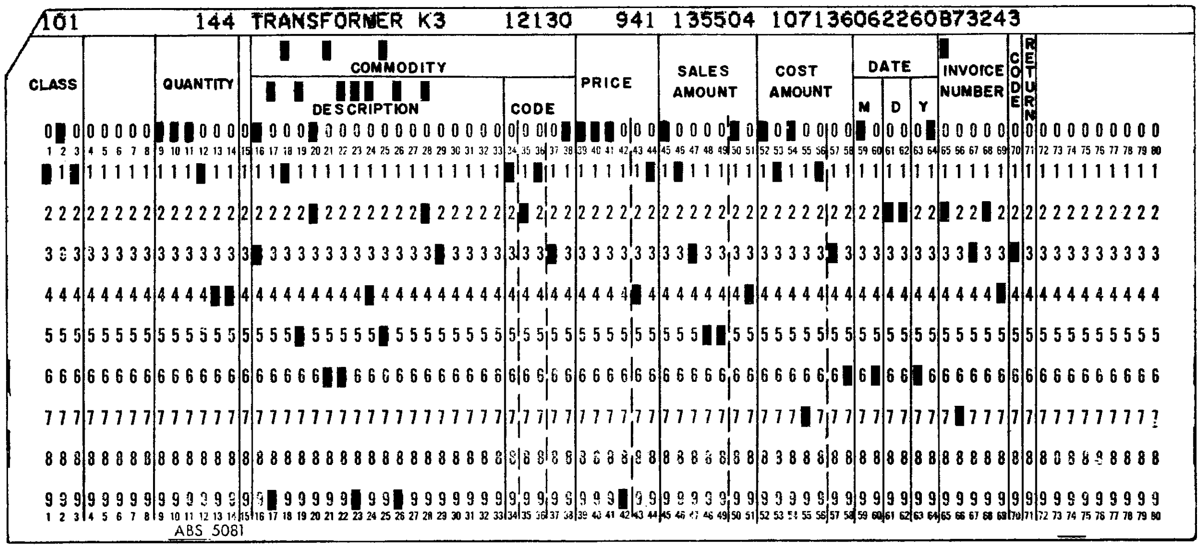
This instruction edits n consecutive characters of decimal data and stores the edited data in memory beginning with the mth ($1 \leq m \leq 12$) decimal digit position of the word at A. The input data are obtained from consecutive column representations in the card storage area, starting with the column specified by C ($1 \leq C \leq 80$). Any attempt to edit a configuration other than a single punch, 0 through 9, per column, results in an unprogrammed subsequence to memory location 0022.

Card Edit, Signed Decimal ECD/A/m, n/C

This instruction edits n consecutive characters of decimal data and stores the edited data in memory, beginning with the mth decimal digit position in the word at A. The input data are obtained from consecutive column representations in the card storage area, starting with the card column specified by C ($1 \leq C \leq 80$). The sign of a negative word is assumed to be overpunched (in the "X" row) in the column specified by C. The absence of an overpunch in this column indicates that the sign of the word is positive. The edit instruction converts the sign to four binary zeros (0000) if negative and to four binary ones (1111) if positive and places these sign bits in the high-order four bits of the word at A. Thus, m (the digit position of the first edited character) cannot be equal to 1, but lies anywhere in the range 2 through 12. The instruction edits into only one memory word; digits exceeding word capacity are disregarded. Unspecified digit positions in the word at A are filled with zeros. Detection of an illegal punch (i. e., any configuration other than a single punch, 0 through 9, per column, and the sign punch in the specified column) results in an unprogrammed subsequence to memory location 0022.

Card Edit, Octal ECO/A/m, n/C

This instruction edits n consecutive characters of octal data and stores the edited data in memory, beginning with the mth ($1 \leq m \leq 16$) octal digit position of the word at A. The input data are obtained from consecutive column representations in the card storage area, starting with the card column specified by C ($1 \leq C \leq 80$). Detection of an illegal punch (i. e., any configuration representing a digit other than 0 through 7) results in an unprogrammed subsequence to memory location 0022.



EASY CODING FORM

PROBLEM ____ PROGRAMMER ____ DATE ____ PAGE ____ OF ____

CARD NUMBER (PAGE / LINE / INSERT)	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	8	15	25	36	47	58 80
		ECU	CARDA	10,3	1	
		ECD	CARDA+1	7,6	9	
		ECA	CARDA+2	1,18	16	
		ECA	CARDA+3	1,5	34	
		ECD	CARDA+6	7,6	39	
		ECD	CARDA+7	6,7	45	
		ECD	CARDA+8	6,7	52	
		ECU	CARDA+9	7,6	59	
		ECA	CARDA+10	1,5	65	
		ECA	CARDA+11	8,1	70	
		ECA	CARDA+12	8,1	71	

Figure 5. Sample Coding for Card Editing

Sample Card Edit Coding

Figure 5 illustrates an input card and the instructions which might be used to edit it. Use of these instructions produces the following results.

1. The three digits of the Class field (columns 1-3) are edited as an unsigned decimal field and stored in decimal digit positions 10, 11, and 12 of word CARDA.

2. The six digits of the Quantity field (columns 9-14 of the input card) are edited as a signed decimal number and stored in the low-order digit positions of word CARDA + 1. Since column 9 of the input card does not contain an 11 overpunch, the sign of the Quantity field is assumed to be positive. The four sign bits (1111) are stored in the high-order digit position of CARDA + 1. Digit positions 2 through 6 of the result are cleared to zeros, since they are unspecified in the signed decimal edit instruction.

3. The contents of columns 16 through 33 of the input card are stored in alphanumeric form in locations CARDA + 2, CARDA + 3, and CARDA + 4. The contents of the six low-order character positions in location CARDA + 4 remains unchanged.

4. The five digits of the Code field are stored in alpahnumeric form in the memory word at CARDA + 5, beginning with the high-order character position. The contents of the three low-order character positions remain unchanged.

5. The digits of the Price, Sales Amount, and Cost Amount fields are each edited in the same way as the Quantity field (above). Unused digit positions in each of the three locations CARDA + 6, CARDA + 7, and CARDA + 8 are cleared to zeros.

6. The six digits of the Date field are edited as an unsigned decimal number and stored in the low-order six digit positions of the location CARDA + 9. The contents of the high-order six digit positions remain unchanged.

7. The five-digit Invoice Number and the two one-digit fields Code and Return are stored in alphanumeric form in locations CARDA + 10, CARDA + 11, and CARDA + 12, respectively. The contents of the unused character positions in these three locations remain unchanged.

Punch Edit, Alphanumeric PCA/A/m, n/C

This instruction edits n (≤ 80) consecutive characters of alphanumeric data into the card punch area in card image form. The edited characters are stored in consecutive card column image positions, starting with the position specified by C ($1 \leq C \leq 80$). The data to be edited (output data) are obtained from consecutive character positions in memory, beginning with the m^{th} ($1 \leq m \leq 8$) alphanumeric character of the word at A.

Punch Edit, Unsigned Decimal PCU/A/m, n/C

This instruction edits n (≤ 80) consecutive characters of decimal data into the card punch area in card image form. The edited characters are stored in consecutive card column image positions, starting with the position specified by C ($1 \leq C \leq 80$). The data to be edited are obtained from consecutive character positions in memory, beginning with the m^{th} ($1 \leq m \leq 12$) decimal digit of the word at A.

If any one of the following bit configurations is encountered in a decimal digit position it is edited as follows:

Bit Configuration	Punched Digit	Bit Configuration	Punched Digit
1010 (Blank)	Blank	1101	Minus Sign
1011	Asterisk	1110	Dollar Sign
1100 (Skip)	Blank	1111	Plus Sign

These non-decimal bit configurations may be placed in a word by the use of the prepare decimal edit instruction (see page 41).

Punch Edit, Signed Decimal PCD/A/m, n/C

This instruction edits n ($1 \leq n \leq 11$) consecutive characters of decimal data into the card punch area in card image form. The edited characters are stored in consecutive card column image positions, starting with the position specified by C ($1 \leq C \leq 80$). The data to be edited are obtained from consecutive character positions in memory, beginning with the m^{th} decimal digit of the word at A. Since the first digit in the word at A is assumed to be the sign digit, m is restricted to the range from 2 through 12. If the sign digit consists of four binary zeros (0000), the card column specified at C is edited to include an "X" overpunch to indicate a negative sign. If the four sign bits are not all zeros, the word is considered positive, and no overpunch is included. This instruction operates on a single word. If the parameters m and n are such that the word at A is exceeded, editing stops.

Punch Edit, Octal PCO/A/m, n/C

This instruction edits n (≤ 80) consecutive characters of octal data into the card punch area in card image form. The edited characters are stored in consecutive card column image positions, starting with the position specified by C ($1 \leq C \leq 80$). The data to be edited are obtained from consecutive character positions in memory, beginning with the m^{th} ($1 \leq m \leq 16$) octal character of the word at A.

Print Edit, Alphanumeric EPA/A/m, n/C

This instruction edits n (≤ 120) consecutive characters of alphanumeric data and stores the edited data in the print area in consecutive print image positions, starting with the one specified by C ($1 \leq C \leq 120$). The data to be edited are obtained from consecutive character positions beginning with the m^{th} ($1 \leq m \leq 8$) alphanumeric character of the word at A.

Print Edit, Decimal EPD/A/m, n/C

This instruction edits n (≤ 120) consecutive characters of decimal data and stores the edited data in the print area in consecutive print image positions starting with the positions specified by C ($1 \leq C \leq 120$). The data to be edited are obtained from consecutive character positions, beginning with the m^{th} ($1 \leq m \leq 12$) decimal digit of the word at A.

Wherever a decimal digit to be edited contains four binary ones (1111), the digit is edited as a plus sign (+); if any four-bit decimal digit contains the configuration 1101, the digit is edited as a minus sign (-). Similarly, the configuration 1010 is edited as a blank ( ), 1011 as an asterisk (*), and 1110 as a dollar sign ($). The configuration 1100 (skip) is edited in both this instruction and the punch edit, unsigned decimal instruction as a blank. (The skip configuration is processed in a different manner in the prepare decimal edit instruction.)

The print edit, decimal instruction is ordinarily preceded by the prepare decimal edit instruction (see below). The latter can edit leading zeros in the field and prepares for proper editing of the sign. These two instructions together provide a type of editing similar to card edit, signed decimal and punch edit, signed decimal. Note, however, that the print edit, decimal instruction operates on n digits rather than a single word.

Print Edit, Octal — EPO/A/m, n/C

This instruction edits n (≤ 120) consecutive characters of octal data and stores the edited data in the print area in consecutive print image positions, starting with the position specified by C ($1 \leq C \leq 120$). The data to be edited are obtained from consecutive character positions beginning with the m^{th} ($1 \leq m \leq 16$) octal character of the word at A.

Prepare Decimal Edit — PDE/A/p_1, p_2/C

This instruction prepares the word at A for subsequent output editing and punching or printing. The prepared word is stored at the location specified by C. The preparation of the word proceeds under control of the two parameters p_1 and p_2 which are written in the B address of the instruction. If p_1 is a digit from 0 through 9, *, $, or "B" (for blank), this digit or character replaces the digit in the sign position (i.e., high-order four bits) of the word at A. If p_1 is a plus sign (+), a minus sign (-), or an "S" (all three indicating sign), the sign is preserved (though the configuration representing the sign in the prepared word may be different from the configuration representing the sign in the word at A).

The parameter p_2 controls the processing of all leading zeros (outside of the sign position) excluding the three low-order digits, which are always preserved. The value of p_2 may be any one of the digits 0 through 9, *, $, +, -, "B" (for blank), or "F" (indicating floating). If p_2 is any one of these characters (except an "F"), then that character replaces all leading zeros in the digit positions 2 through 9. If p_2 is "F", then the character in the sign position (which is controlled by parameter p_1) is floated to the right through all leading zeros until it reaches either a non-zero digit or digit position 9; the character replaces the right-most leading zero through which is has floated, while the leading zeros to the left of the floated character are replaced by blanks.

PERIPHERAL INSTRUCTIONS

The instructions in this group position magnetic tapes, transfer information from memory to output devices and from input devices to memory, position input and output devices, and provide communication between the central processor and the console.

For card readers and card punches, the time between the interpretation of instructions (calling for card reading or punching) and the beginning of actual transfer of information into or out of the central processor (acceleration time) is great enough to permit a number of other central processor operations to be executed. This time may be used by the programmer for such a purpose if the device is operating in the "without interlock" mode. The acceleration time is matched by a similar interval after data transfer has been completed. This deceleration interval may also be used by the central processor for other operations, regardless of the mode of operation of the device.

Each peripheral device which has the option of using the acceleration time has two operation codes; e.g., read card without interlock (RCW) and read card interlocked (RCI). Both of these cause the transfer of one card's worth of data in memory; the former, however, permits central processor operations during the acceleration (and, of course, during the deceleration) interval.

The parameters in the B address field of the magnetic tape instructions may be absolute or symbolic; if they are symbolic, they should be equated to integers using EQUALS instructions (see Section V). The parameter t should have a value in the range 0-7; this represents the "logical address" of the tape being manipulated (see page 82).

Read Magnetic Tape RDT/A/t, b/

This instruction reads one record from tape t into consecutive memory locations starting with the location specified by A. Tape is always read in the forward direction. At the conclusion of data transfer, the address of the location into which the last word from tape was read is automatically stored in the A address portion of reserved location 0013. When a read tape instruction immediately follows a write tape instruction (see below), the reading and writing are performed simultaneously. When a read tape instruction is executed, an address, one greater than the address of this instruction, is stored in the A address portion of location 0021. This enables the read tape instruction to be located by subsequent instructions if desired. The A address portion of the reserved location is unchanged until another read tape instruction is executed, at which time the new setting of the sequence register increased by 1 is stored there.

For a normal read, the parameter specified in the B address, b, is zero. If b is specified as one of the values 1 through 9, indicating one of nine channels on tape, the read instruction containing this specification is executed ignoring the corresponding tape channel (channel 9 is the parity channel); the ignored channel is then automatically regenerated so as to retain correct parity across the eight channels in memory and the channel being regenerated.

If an error occurs during a normal read (b = 0), an unprogrammed subsequence to the reserved memory location 0018 is initiated at the end of the read. During a read where $1 \leq b \leq 9$, error checking is suspended. Note that indexing the B address prevents simultaneous write/read operations. The C address field must be left blank.

This instruction can be used to correct all single-channel errors in a record by rereading that record with the channel containing the errors(b) specified in the B address. The channel containing the errors may be located using orthowords (see "Check Parity Instruction", below).

Reading part or all of a record into the stopper location is a useful method of bypassing unwanted sections of magnetic tape (see "Stopper Address ", Section III).

Write Magnetic Tape — WRT/A/t, n/

This instruction writes one record of n ($\leq$511) words, starting with the word at A, onto tape t. Parameter n should include an allowance for two orthowords (see page 50). Tape is always written in a forward direction. If a write tape instruction is followed immediately by a read tape instruction, the reading and writing are performed simultaneously. If a tape error occurs during writing, an unprogrammed subsequence to reserved memory location 0019 is initiated at the end of the write. If the end of tape is sensed during writing, an unprogrammed subsequence to location 0020 is initiated at the end of the write; one additional record of up to 511 words may now be written on this tape. When a write tape instruction is executed, an address, one greater than that of this instruction, is stored in the C address portion of the reserved location 0021. This enables the tape write instruction to be located by subsequent instructions if desired. The C address portion of this location is unchanged until another write tape instruction is executed, at which time the new setting of the sequence register increased by 1 is stored there.

A write instruction should not be issued to a tape unit immediately after a read instruction to this same unit. Successful writing can be guaranteed only when the previous operation performed on that tape unit was a write, a rewind, or a backspace.

The C address field must be left blank.

Rewind Tape — RWT/ /t/

This instruction rewinds tape t to its physical beginning. Central processor operations continue as soon as the rewind instruction has been interpreted, i.e., the actual rewinding proceeds in parallel with central processor operations. However, 100 milliseconds must elapse before the execution of another rewind instruction. If another rewind instruction is received before the end of this 100 ms interval, the machine stalls until the end of the interval is reached. The interval may be used for central processor operations other than the rewind instructions. There are no other restrictions on the operations which may be executed during a tape rewind save for the following three:

1. A backspace tape (BST) instruction directed to the tape unit which is being rewound suspends further operations directed to that tape until a manual reset is made on it; the tape unit is effectively interlocked. A backspace tape instruction directed to any other tape unit is executed normally.
2. A read tape (RDT) instruction is executed normally unless it is directed to the unit being rewound. In the latter case, further machine operations are suspended until the tape has been rewound.
3. A write tape (WRT) instruction is executed normally unless it is directed to the unit being rewound. In the latter case, further machine operations are suspended until the tape has been rewound.

The A and C address fields are not interpreted at execution time but are assembled normally and may contain any legal address format.

Backspace BST/ /t/

This instruction backspaces tape t by one record. Central processor operations continue as soon as the instruction has been interpreted, i. e., the actual backspacing proceeds in parallel with other central processor operations. Save for the following exceptions, there are no restrictions on the operations which may be executed during a tape backspace.

1. Any tape instruction directed to the tape unit which is being backspaced suspends machine operations until the backspace instruction has been completed.
2. A read tape (RDT) or backspace (BST) instruction issued for any tape unit suspends machine operations until the backspace instruction has been completed.

The A and C address fields are not interpreted at execution time but are assembled normally and may contain any legal address format.

Read Card, Without Interlock RCW/ / /

This instruction reads the information from the next card in the card feed into the card image area in memory. During the acceleration time (i. e., between the initiation of the instruction and the beginning of data transfer), other central processor operations may be executed. These are performed under control of the sequence register. When the transfer of data commences, the current reading of the sequence register is frozen, and further central processor operations are suspended. When data transfer is complete, a subsequence to the reserved location 0026 is executed. This location contains an instruction which controls the program path after completion of data transfer. The programmer sets up this instruction so that control returns immediately to the point where central processor instructions were suspended or so that control returns to that point after other instructions have been executed out of sequence.

If a read error occurs during card reading, an unprogrammed subsequence to location 0025 is executed upon completion of data transfer. All address fields are ignored at execution time but are assembled normally.

Read Card, Interlocked RCI/ / /

This instruction reads the information from the next card in the card feed into the card image area in memory. During the acceleration time, no other central processor operations are possible; at the completion of data transfer, processing continues in sequence and no subsequence is executed.

If a read error occurs during execution of the instruction, an unprogrammed subsequence to the fixed location 0025 is executed upon completion of data transfer. All address fields are ignored at execution time but are assembled normally.

Reject REJ/ /n/

This instruction causes the card currently at the reading station of the card reader to be stacked in one of two particular pockets in the card reader. When issued in the deceleration interval of a read card instruction, it rejects the card just read. (Three pockets are used on the card reader; cards not rejected are automatically stacked in the third pocket.) If pocket 1 is desired, the parameter n takes the value 1; if pocket 2, n takes the value 2. This pocket selection device is an option on the Honeywell 400. If this option is not present in the device, selection of pocket 2 results in normal stacking (i.e., selection of pocket 3). The A and C address fields are ignored at execution time but are assembled normally.

Punch Card, Without Interlock PCW/ / /

This instruction punches the information from the card punch image in memory onto the next card in the card punch. Between the initiation of the instruction and the beginning of data transfer, other central processor operations are possible. These are performed under control of the sequence register. When the transfer of data commences, the current reading of the sequence register is frozen, and further central processor operations are suspended. When data transfer is complete, a subsequence to the reserved location 0028 is executed. This contains an instruction which controls the program path after completion of data transfer. The programmer sets up this instruction so that control returns immediately to the point where central processor operations were suspended, or so that control returns to that point after other instructions have been executed out of sequence.

If a card punch error occurs during punching, an unprogrammed subsequence to the reserved location 0027 is executed upon completion of data transfer on the following card. All address fields are ignored at execution time but are assembled normally.

Punch Card, Interlocked PCI/ / /

This instruction punches the information from the card punch image in memory onto the next card in the card punch. During the acceleration time, no other central processor operations are possible; at the completion of data transfer, no subsequence occurs and processing continues in sequence.

If a punch error occurs during execution of the instruction, an unprogrammed subsequence to the reserved location 0027 is executed upon completion of data transfer on the following card. All address fields are ignored at execution time but are assembled normally.

Offset Stack OFS/ / /

This instruction causes the card currently at the checking station in the card punch to be offset in the stacker in the card punch. In this way, an error card can be identified without the programmer having to examine each of the cards in the stacker. An error for a particular card is indicated at the time of the next card punch instruction; following the completion of the punching of this second card, an error subsequence is made to location 0027. To offset the error (i.e., the first) card, the offset stacking instruction must be issued in the deceleration interval of this second card. The A and C address fields are ignored at execution time but are assembled normally.

Print and Space PRS/ /HE, L/

This instruction prints one line on the high-speed printer from data obtained from the print storage area in memory. Spacing of the form is governed by the print format information specified by the programmer in the B address. An "H" written in the address specifies skipping to the head of the page; an "E" specifies testing for end of form; and "L" is the number of lines to be spaced after printing. L is restricted to a maximum of 63. There are three significant combinations of parameters:

```
H
E, L
 , L
```

Notice that where parameters are omitted in leading positions (see "Parameters", Section III), the omission is indicated by the leading comma.

After completion of printing and before the advancing of the paper, processing continues in sequence. If the system is equipped with the print storage option, however, a subsequence occurs to the reserved location 0024 after completion of printing and before advancing of the paper. This storage device, however, enables central processor operations to occur in parallel with the printing operation since it feeds the printer independently of the central processor. If a print error occurs during execution of the print instruction, an unprogrammed subsequence to the reserved location 0023 is executed.

The A and C address fields are not interpreted at execution time but are assembled normally.

Type Alphanumeric, Console TAC/A/M/

This instruction prints the word at A on the console printer in alphanumeric format. If the letter "M" (more to follow) is specified in the B address, carriage return is inhibited. If carriage return is required, the B address is left blank. Indexing is not permitted in the B address of this instruction. The C address field is not interpreted at execution time but is assembled normally.

Type Octal, Console TOC/A/M/

This instruction prints the word at A on the console printer in octal format. If the letter "M" (more to follow) is specified in the B address, carriage return is inhibited. If carriage return is required, the B address is left blank. Indexing is not permitted in the B address of this instruction. The C address field is not interpreted at execution time but is assembled normally.

Type Decimal, Console TDC/A/M/

This instruction prints the word at A on the console printer in decimal format. If the letter "M" (more to follow) is specified in the B address, carriage return is inhibited. If carriage return is required, the B address is left blank. Indexing is not permitted in the B address of this instruction. The C address field is not interpreted at execution time but is assembled normally.

General-Purpose Peripheral Instructions

A number of different peripheral devices can be connected to a Honeywell 400 in addition to card equipment and printers. The instructions associated with each of these devices have similar formats in EASY language. The address specified in the A address field always denotes the starting location into which the first unit of information is to be read, or from which the first unit is to be recorded as output. The number specified in the B address field denotes the number of units of information to be read from the device into memory, or to be recorded from memory on the device. Each device has a particular unit of information associated with it.

After completion of data transfer on each of the peripheral devices a subsequence to a reserved location associated with the device is executed. Each device is also associated with a reserved location to which an unprogrammed subsequence is executed in the case of an error.

INDEX AND CHECK INSTRUCTIONS

There are four instructions for manipulating the contents of the three index registers and the sequence register (see Section I), and two instructions for generating checking and control information.

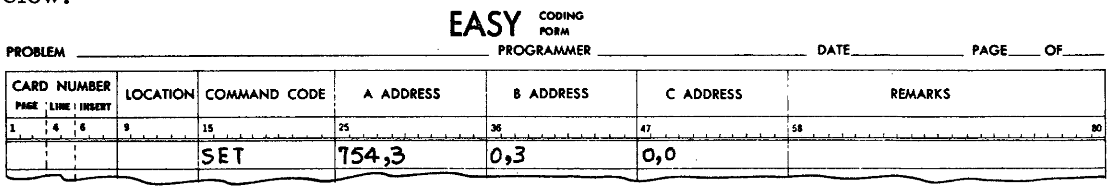
The index instructions are used for setting, storing, restoring, and testing and incrementing the contents of the index registers, and for storing, and restoring the contents of the sequence register.

The check instructions are the compute orthocount instruction (which generates the two orthowords for a designated number of words), and the check parity instruction, which checks the internal parity of a specified number of words in memory.

Set Index Register

SET/A, IR_a/B, IR_b/C, IR_c

This instruction adds the value "A" to the contents of index register IR_a and stores the result in index register 1; similarly, it adds the value "B" to the contents of index register IR_b and stores the result in index register 2; and adds the value "C" to the contents of index register IR_c and stores the result in index register 3. The values "A", "B", and "C" must be specified by legal forms of addressing. If any index register indication (i.e., IR_a, IR_b, or IR_c) is zero, the corresponding base value (i.e., "A", or "B", or "C", respectively) is stored in the associated index register (i.e., 1, 2, or 3, respectively). This is the only instruction in which a specific index register is implicitly related to a specific address. All index register address additions are completed before the index registers are set to their new values. This point is illustrated in the example below.


EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE______ PAGE____ OF____

CARD NUMBER PAGE LINE INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	8	15	25	36	47	58 80
		SET	754,3	0,3	0,0	

Let index register 3 originally contain the value 200 (in binary form).

The above SET instruction adds 754 to the contents of index register 3 (i. e., to 200) and stores the sum of 954 in index register 1; it adds 0 to the contents of index register 3 (before the addition of the A address) and stores this sum (= 200) in index register 2; and, since the C address index indication is 0, as is C itself, it stores 0 in index register 3.

Thus, after the instruction, index register 1 is set to 954; index register 2 to 200; and index register 3 to 0.

<u>Store Index Register</u> STX/A/ /C

This instruction stores the contents of the three index registers and sequence register in the word at A. Since the registers all contain 12 bits, the whole of the word at A is filled. After the instruction has been completed, the sequence register is set to the address specified by C. The B address field is not interpreted at execution time but is assembled normally.

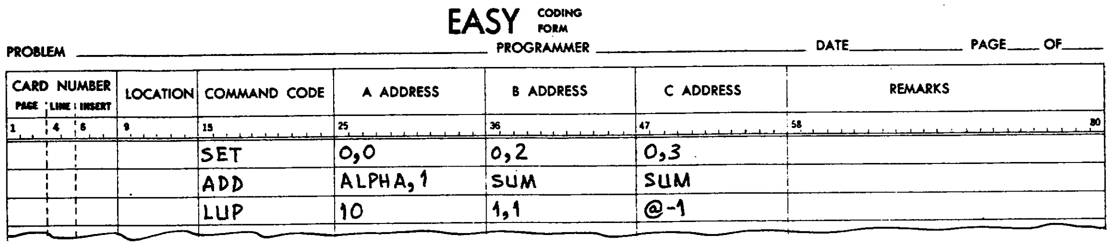
<u>Restore Index Register</u> RTX/A/ /C

This instruction stores the high-order 36 bits of the word at A, in 12-bit segments, in index registers 1, 2, and 3, respectively; and stores the low-order 12 bits of the word at C in the sequence register. The B address field is not interpreted at execution time but is assembled normally.

<u>Test Index and Increment</u> LUP/A/I, IR_i/C

This instruction compares the value "A" (or effective "A", if the A address is indexed), with the contents of index register IR_i. If the contents of IR_i are less than "A", the increment I is added to the contents of IR_i and the sum is stored in that index register. The sequence register is then set to the address specified by C. If, however, the contents of IR_i are greater than or equal to "A", the next instruction in sequence is executed. If the programmer desires to decrement the index register by d, the value of I must be such that I + d = 4096.

The instructions below illustrate the manner in which the index register instructions are used to facilitate coding



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE______ PAGE___ OF___

CARD NUMBER PAGE LINE INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	9	15	25	36	47	58 80
		SET	0,0	0,2	0,3	
		ADD	ALPHA,1	SUM	SUM	
		LUP	10	1,1	@-1	

The SET instruction sets index register 1 to 0; index register 2 is unchanged (since 0 is added to its contents and the result stored in index register 2); similarly, index register 3 is unchanged.

The ADD instruction adds the contents of the location tagged ALPHA to the contents of the location tagged SUM, and stores the result in the latter location. The LUP instruction then compares the value 10 with the contents of index register 1. Since the contents of this index register are less than 10 (in fact, they are 0, since the ADD instruction does not alter them), the increment of 1 is added to index register 1, and the sequence register is set to the location specified by the current instruction minus 1; i.e., to the ADD instruction. This instruction is therefore executed again, but this time it adds the contents of the location one location beyond that tagged ALPHA (since it is indexed by index register 1 whose contents are now unity) to the contents of the location tagged SUM, and again stores the result in the latter location. This process continues until the LUP instruction is executed with the index register 1 having contents of 10, at which time the next instruction in sequence is executed. At this stage, the ADD instruction has been executed 11 times, and the 11 consecutive words, beginning with the word in the location tagged ALPHA, have been summed and stored in the location tagged SUM.

Compute Orthocount COC/A/n/C

This instruction performs an orthocount of n ($\leq$ 511) consecutive words in memory, starting with the word specified by A, generates two orthowords, and stores these in two consecutive locations specified by C and C + 1. If the number of words orthocounted (n) is even, then the orthoword in the location specified by C represents the orthocount of the odd words (1, 3, 5, etc.,) in the sequence of n words, and the orthoword in the location specified by C + 1 represents the orthocount of the even words (2, 4, 6, etc.). If n is odd, orthoword 1 contains the orthocount of the even words, and orthoword 2 contains the orthocount of the odd words. Each orthoword is the complement of the binary half add (binary add without carry) of the words associated with it. The parameter n may be of any legal address format except literal. If it is symbolic, it should be equated to an integer using EQUALS instruction.

The technique of Orthotronic Control incorporated in the Honeywell 400 assumes that every record written on tape includes two orthowords (computed as described above for a record of n words). The compute orthocount instruction is also used in conjunction with the check parity instruction (see below) to reconstruct data in which an error has been detected. The erroneous word is identified by the check parity instruction, and the channel in which the error occurred is identified by computing two new orthowords for the record in error (including the original orthowords for that record). The channel is identified by examination of these new orthowords.

Check Parity CHP/A/n/C

This instruction generates parity for n ($\leq$ 4095) consecutive words in memory, starting with the word specified by A and compares the generated parity with the parity of the word stored in memory. If a word with incorrect parity is detected, the parity bits of the offending word are corrected, its address is stored in the A address portion of the check parity word, and a subsequence to the location specified by C is performed.

The instruction terminates either when a word with incorrect parity is detected (as described above) or when the specified number of words have been checked. If no incorrect words are discovered, the instruction terminates, the contents of the check parity word are unaltered, and the next instruction in sequence is executed.

The parameter n may be of any legal address format except literal. If it is symbolic, it should be equated to an integer using the EQUALS instruction.

SECTION V

ASSEMBLY CONTROL INSTRUCTIONS

EASY language includes a group of instructions which the programmer uses to control the assembly of his program. These are punched one per card like machine instructions, but in general, they are not translated into machine instructions. Instead, they are simply interpreted to provide control data required by the Assembly program.

PROGRAM Director

The PROGRAM control instruction identifies the beginning of an EASY program. This card, known as the "PROGRAM Director", must appear in the input to Assembly at the beginning of each new program.

The card has a dual significance: the location field of the PROGRAM Director specifies the way in which the updating program (LAMP-PSP) will handle the subsequent program; the remaining fields specify the way in which Assembly will handle the subsequent program. The A address field contains the program name; the B address field is not used; the C address field specifies sorting, identification, and punching options; the "Remarks" columns may be used to describe the system on which the assembled program will be run. Thus, the general format of the control instruction is as follows:

EASY CODING FORM

PROBLEM ________ PROGRAMMER ________ DATE______ PAGE___ OF___

CARD NUMBER PAGE | LINE | INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 | 4 | 6	9	15	25	36	47	58 — 80
	(ACTION)	PROGRAM	(PROG.NAME)		(OPTIONS)	(CONFIG. CODE AND ID DATA)

Detailed specifications for the use of these columns are given below:

Columns	Contents	Explanation
1 - 8	Card Number	See Section II.
9 - 14	NEW, RENEW, or blank	NEW - Directs LAMP-PSP to add all segments up to the next PROGRAM card to the master program tape unless another program of the same name already exists on the tape; in that case, LAMP-PSP will not add this program. RENEW - Directs LAMP-PSP to add this program to the master program tape and to eliminate any existing program with the same name. Blank - Directs LAMP-PSP to add the following segments as directed by the SEGMENT control cards. This field is blank when each segment

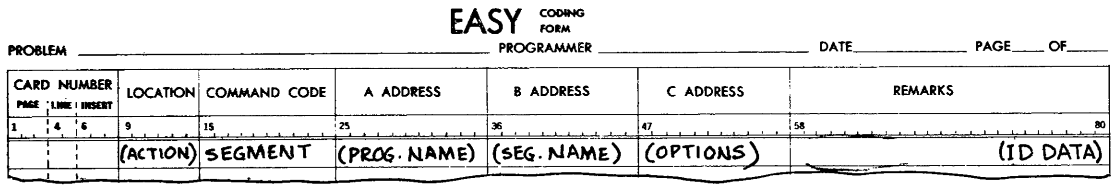
Columns	Contents	Explanation
		is to be handled differently or when not all segments are affected.
15 - 24	PROGRAM	
25 - 35	Program Name	Up to six alphanumeric characters.
36 - 46	Blank	
47	S or blank	S - Assembly will check the sequence of card numbers for the input program and sort the cards if they are out of order; all cards must contain card numbers. Blank - Assembly will take sorting specifications from SEGMENT cards for this program.
48	Blank	
49	I or blank	I - Assembly will match columns 74-80 of every card in this program with columns 74-80 of this card. It will not assemble cards which do not match but will list these cards with an error indication. If Assembly finds a SEGMENT card which does not match, it will not assemble that card and all cards following it to the next SEGMENT or PROGRAM card. Blank - Assembly will take checking specifications from SEGMENT cards for this program.
50	P or blank	P - Assembly will punch a card deck containing the binary coding for this program. This will be in addition to writing a tape - the standard Assembly output. Blank - Assembly will take punch option from SEGMENT cards for this program.
51 - 57	Blank	
58	1, 2, 3, 4 or blank	1, 2, 3 or 4 - An indication of the number of 1024 word modules of the memory in the machine on which the assembled program will be run. That is, 1 for a system with 1024-word capacity, 2 for a system with 2048-word capacity, etc. This number is used in determining the stopper address. Blank - Assembly assumes that the program will be run on the machine on which it is assembled. It uses an appropriate stopper address as supplied by Monitor.
59	S or blank	S - Multiply and divide instructions are assembled as legal instructions. "S" appears here when the system on which the program will be run contains the Multiply/Divide Option. Blank - Multiply and divide instructions will be assembled with error notations.
60 - 73	Blank	

Columns	Contents	Explanation
74 - 80	Identification Characters	When column 49 contains an I, these columns must contain the characters used to identify cards in this program.

SEGMENT Director

The SEGMENT control instruction marks the beginning of each segment in a program. A segment is defined as a section of coding that is read into memory as a unit; normally, programs are divided into segments so that these units can overlay each other in memory. Every segment of a program, including the first, must be preceded by a SEGMENT Director.

Like the PROGRAM Director, the SEGMENT Director has significance both to LAMP-PSP and to the Assembly Program. The location field of the SEGMENT Director specifies the way in which LAMP-PSP will handle the subsequent segment; the remaining fields specify the way Assembly will handle the subsequent segment. The A address field contains the program name; the B address field contains the segment name; the C address field specifies sorting, identification and punching options; the "Remarks" columns may contain identification data. Thus, the general format of this control instruction is as follows:



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE | LINE | INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 | 4 | 6	9	15	25	36	47	58 … 80
	(ACTION)	SEGMENT	(PROG. NAME)	(SEG. NAME)	(OPTIONS)	(ID DATA)

Detailed specifications for the use of these columns are given below:

Columns	Contents	Explanation
1 - 8	Card Number	See Section II.
9 - 14	NEW, RENEW, or blank	NEW - Directs LAMP-PSP to add all coding up to the next SEGMENT card to the program identified in the A address field unless that program already has a segment of the same name; in that case, LAMP-PSP will not add this segment. RENEW - Directs LAMP-PSP to add this segment to the specified program on the master program tape and to eliminate any existing segment within that program having the same name. Blank - Normally indicates that action for all segments has been specified on the PROGRAM Director. If the action codes on both the PROGRAM and SEGMENT Directors are blank, all segments will be assumed to be NEW.
15 - 24	SEGMENT	
25 - 35	Program Name	Up to six alphanumeric characters.

Columns	Contents	Explanation
36 - 46	Segment Name	Up to six alphanumeric characters.
47	S or blank	S - Assembly will check the sequence of card numbers for this segment; if the input cards are out of order, Assembly will sort them. All cards must contain card numbers. Blank - If column 47 of the PROGRAM Director contained an "S", continue sorting the entire program by card numbers. Otherwise, accept cards in order of their appearance in the input deck and generate line numbers for use in the printed listing.
49	I or blank	I - Assembly will match columns 74-80 of every card in this segment with columns 74-80 of this card. It will not assemble cards which do not match but will list these cards with an error indication. Blank - If column 49 of the PROGRAM Director contained an "I", Assembly continues checking cards against the PROGRAM Director. Otherwise, it does not perform this check.
50	P or blank	P - Assembly will punch a card deck representing the binary coding in this segment; it will also write the coding on tape - the standard Assembly output. Blank - If column 50 of the PROGRAM Director contained a "P", Assembly will punch a card deck for this segment. Otherwise, it will not punch cards for this segment.
74 - 80	Identification Characters	Must contain the characters used to identify this segment's cards when column 49 contains an "I".

Establishing Common Segments

In order to allow communication between segments, the programmer may define one or more segments as "common". Assembly will then assign addresses so that the common segment will normally remain in memory during the execution of n subsequent segments. Common segments thereby provide a method for controlling the operation of a program.

In order to define a segment as common, the command code SEGMENT is followed by a comma and a number, n, from 1 to 99. That segment will then be common to the next n segments. Since segments of a program are assembled in alphabetical order by segment name, names must be chosen so that the common segment will precede the correct segments after assembly.

Common segments may be overlapped or "nested". For instance, two segments might be common to a third.

Tags defined in a common segment may be referenced in the n segments following. Therefore, a tag defined in a common segment must be unique in all the following n segments.

Loading Segments into Memory

During a LAMP-PSP run, the programmer specifies the segment to be loaded first when his program is executed. Thereafter, whenever a new segment is needed, the programmer may load it into memory with a Read Segment instruction. This instruction has the following format:

```
L, RDSEG/segname
```

where "segname" is the name of the segment to be loaded. This instruction loads the specified segment and sets the index registers and sequence counter according to the BEGIN instruction (see below). If the programmer wishes to retain the present contents of the index registers, he may use the instruction:

```
L, RDSEGX/segname.
```

Set Location Counter

Tag/SETLOC/Address

In general, the Assembly Program automatically assigns addresses and allocates memory for EASY programs; it does this using a counter called the Current Location Counter. However, the programmer can modify and control this process using a SETLOC instruction which sets a specified value into the Current Location Counter. This instruction causes Assembly to assign subsequent instructions and constants to memory locations starting at the one specified in the SETLOC itself.

Commonly, a SETLOC instruction will precede the first instruction in a segment or program. When so used, it has the effect of allowing the programmer to specify the beginning of the area to be assigned to his program. If a SETLOC is not included at the beginning of the program or segment, Assembly will assign addresses beginning at 300 (decimal). Location 300 is the first location not reserved for use by the hardware or by the Monitor program.

This instruction contains the word SETLOC in the command code field and a memory address in the A field. In the location field, it can contain a tag which will be assigned the address in the A field.

The expression in the A address field may be an absolute address, a symbolic address (a special character or tag except @ which is illegal), or a combination of up to six integers and symbols. Plus and minus signs may be used to combine integers and symbols. If the resulting expression has a value greater than 4095, it will be interpreted modulo 4096 and the result will be used. If the expression is negative, it will be subtracted from 4096 and the result will be used.

Any symbol used in the A field must previously have been assigned to a memory address or to an integer. That is, it must have previously appeared in a location field as a tag or must have been defined by an EQUALS instruction.

In addition to controlling the assignment of instructions and constants, a SETLOC may also control the assignment of literals. Normally, Assembly will assign literals starting at the address specified by the Current Location Current upon completion of a segment. Therefore, if a SETLOC is the last instruction of a segment, literals will be stored starting at the location specified by that SETLOC.

Repeat REP/n

The REPEAT instruction causes the constant or the instruction immediately following it to be assembled the number of times specified by n (a number 1 - 4096 in the A address field). The words repeated are assigned consecutive memory addresses starting with the address specified by the Current Location Counter at the time the REPEAT is given. Thus, the instruction following REPEAT is assembled and later executed n times, or the constant immediately following REPEAT is assembled and stored n times. The value n must be a number; it cannot be symbolic.

Equals Tag/EQUALS/value of tag

One method of assigning absolute values to symbolic tags is through the use of the EQUALS instruction. This instruction defines a tag as equal to the value in its A address field.

One useful application of the EQUALS instruction is in defining parameters. A parameter may be defined by an EQUALS and referred to symbolically throughout the program. If the parameter must later be changed, only one line of coding need be revised.

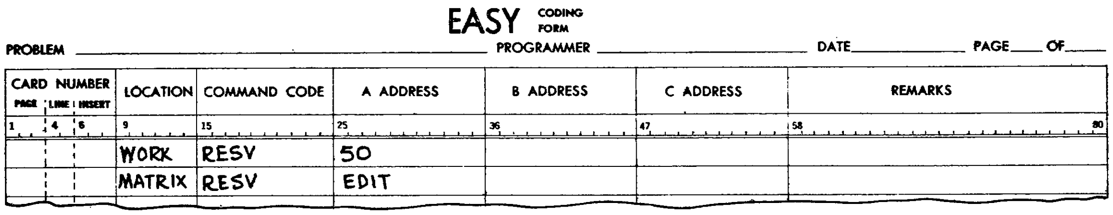
The symbolic tag to be defined appears in the location field; the word EQUALS appears in the command code field; the value which defines the tag appears in the A address field. This value may be an integer, a symbolic address (except @ which is illegal), or a combination of up to six integers and symbols. Plus and minus signs may be used to combine integers and symbols. If the resulting expression has a value greater than 4095, it will be interpreted modulo 4096 and the result will be used. If the expression is negative, it will be subtracted from 4096 and the result will be used.

Any symbol used in the A field must previously have been assigned to a memory address or to an integer. That is, it must have previously appeared in a location field as a tag or must have been defined by an EQUALS instruction.

Reserve Starting Location/RESV/No. of Locations

This instruction reserves a block of consecutive memory locations starting at the address specified in the location field of the RESERVE instruction itself. The A address field specifies the number of locations to be reserved.

The location field of a RESERVE instruction contains a tag representing the address of the first location to be reserved. The command code field contains "RESV". The A address field may contain an integer or an expression of up to six integers and symbols; the value of the A field will be computed by determining the value of the symbols and adding or subtracting as indicated. Any symbols used must have been defined previously. If the total value of the A field is greater than 4095, it will be interpreted module 4096; if the value in the A field is negative, it will be subtracted from 4096 and the result will be used.



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE LINE INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	9	15	25	36	47	58 80
	WORK	RESV	50			
	MATRIX	RESV	EDIT			

The first example above reserves 50 locations starting at the location tagged WORK. The second example reserves EDIT locations starting at the location tagged MATRIX (where EDIT has been previously assigned an integer value by the EQUALS instruction).

Begin Loc/BEGIN/A/B/C

The BEGIN instruction has two functions:

1. To designate the location of the first instruction of a segment to be executed.
2. To preset the three index registers to their required initial values.

The address of the first instruction to be executed appears in the location field; the code BEGIN appears in the command code field; the value to be set into index register 1 appears in the A field; the value for 2 appears in the B field; and the value for 3 appears in the C field. If any field is left blank, the corresponding register will be set to zero. If the segment is loaded by a L, RDSEGX instruction, the index registers will not be reset but will retain their existing values; however, the sequence counter will be reset.

The location field and all address fields may contain any type of legal address; indexing is not permitted. If a symbolic tag appears in the location field, that tag must also appear in the location field of another instruction. Thus, the BEGIN instruction is an exception to the rule that tags may appear in only one location field. It is also an exception to the rule that address arithmetic may appear in a location field: BEGIN tags may have address arithmetic.

The BEGIN instruction must appear in the Assembly input card deck following the last program word in a segment; alternatively, when cards are to be sorted by line number, the BEGIN must have a line number such that it will be sorted into this position.

<u>Exit</u> Tag/EXIT

The EXIT instruction notifies Monitor that a program has been completed. Assembly interprets the EXIT as a sequence change instruction allowing Monitor to load the next program. If there is a tag in the location field, it will be assigned to the memory location of the sequence change instruction.

SECTION VI

CONSTANTS

The EASY language permits the programmer to represent constants in five forms. EASY Assembly converts these forms into the appropriate internal structures in machine language. The five types of constants are alphanumeric, hexadecimal, octal, fixed binary, and mixed (i. e., a single line of coding with separate expressions in each address field and in the command code). All constants are assembled as 48 bits and stored in single memory locations (with the possible exception of alphanumeric constants which may specify more than 48 bits, which then are stored in several consecutive memory locations).

The first four types of constants are specified by the constant code (CON) punched in the command code field, followed by a mnemonic code (which identifies the type of constant), the number sign (#), which separates the code from the actual value and the actual constant required. The mnemonic code (consisting of the letters "A" for alphanumeric, "H" for hexadecimal, "O" for octal, and "F" for fixed binary, respectively), the number sign and the actual constant are punched in consecutive columns beginning with the first column in the A address.

Mixed constants are specified by four expressions, one expression in the command code and one in each of the three address fields. The leading term in each of these expressions is again a mnemonic code identifying the type of constant the subsequent expression defines. The second term in each expression is the constant separater and definer (i. e., the number sign).

The location field of a card containing a constant may specify an absolute address or a legal symbolic tag (see "Symbolic Addressing", Section III). The memory location designated in this way is where the 48-bit, one-word constant (or the first word of a multi-word constant) is stored.

Alphanumeric Constants

These constants are specified in the command code field and the address fields of the card, ignoring field boundaries, in the form

```
CON / A#XX...X
```

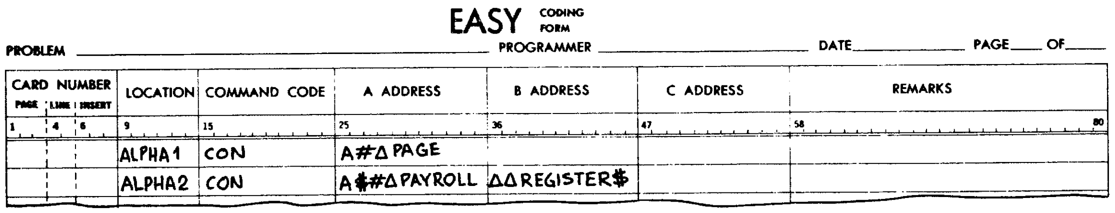
where XX...X represent the alphanumeric characters the programmer desires for the constant. The first eight of these are assembled and stored as eight six-bit characters in machine language. Characters in excess of eight, counting from the number sign, are dropped. A blank or space is recognized as a valid character and is not suppressed; blanks or spaces are assembled and stored in the same manner as other alphanumeric characters.

Alphanumeric constants of more than one word are specified in the form

CON / At#XX. . . Xt

where t stands for any symbol the programmer chooses to terminate the constant. All of the characters specified between the number sign and the terminating symbol are assembled in machine language. The only restriction on the number of characters which can be specified is imposed by the number of columns available to specify the constant. A constant may use all 33 columns (25-57) of the three address fields, ignoring address boundaries. Four of these columns are used to specify a multi-word constant (i. e., the first three and final one), and so 29 columns are left to specify the actual alphanumeric characters required. A multi-word alphanumeric constant can therefore consist of up to 29 characters. The specified characters are assembled and stored in as many consecutive memory locations as needed to encompass all of the constant. The final location of this sequence may not contain eight six-bit characters if the number of specified characters is not an exact multiple of eight. In this case, the six-bit characters which are in that location are left justified within it, and the remaining positions in that word are automatically set to blanks.

If the terminating symbol is not specified, the constant is interpreted as an ordinary one-word alphanumeric constant as described above; in this case, characters in excess of eight are dropped.


EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE______ PAGE____ OF____

CARD NUMBER PAGE	LINE	INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1	4	6	9	15	25	36	47	58 80
			ALPHA1	CON	A#Δ PAGE			
			ALPHA2	CON	A$#ΔPAYROLL	ΔΔREGISTER$		

The first example above defines an alphanumeric constant of one word which is assembled and stored in the location tagged ALPHA1. The high-order six bits of this word consist of the six-bit representation of a blank. The following 24 bits consist of the four six-bit representation of the characters P, A, G, and E, respectively. The last 18 bits consist of three successive six-bit representations of blanks.

The second example defines an alphanumeric constant of more than one word which is terminated by the dollar sign ($). The first eight characters (i. e., blank, P, A, Y, R, O, L, and L) are assembled and stored in the given order in the location tagged ALPHA2. The next eight characters (i. e., two blanks, R, E, G, I, S, and T) are assembled and stored in that order in the location specified by ALPHA2 + 1 (i. e., the location one beyond that tagged ALPHA2). The next two characters are assembled and stored in that order left justified in the location specified by ALPHA2 + 2. The rest of this last location consists of trailing blanks.

Hexadecimal Constants

These constants are specified in the command code field and the address fields of the card ignoring field boundaries, in the form

CON / H#XX...X

where XX...X represent the hexadecimal characters the programmer desires for the constant. Only the characters 0 through 9 and B through G, are legal.

The maximum number of characters which can be defined is 12, and these are stored as four-bit characters in machine language. When a constant has been prematurely terminated by an illegal character or has not been wholly defined, the unspecified character positions in the assembled word are set to zero. The positions of these zeros in the assembled word depend on the following sign convention.

If the constant begins with a plus (+) or minus (-) sign, the sign is assembled (as four binary ones or four binary zeros, respectively) and stored in the high-order positions of the assembled word. A maximum of 11 further characters can be explicitly defined by the programmer. All explicitly defined characters are right justified in the assembled word and zeros fill the positions between the sign bits and the assembled characters.

If the constant is unsigned, all explicitly defined characters are left justified in the assembled word, while the remaining positions (if any) of that word are filled with zeros.

EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE______ PAGE___ OF___

CARD NUMBER			LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
PAGE	LINE	INSERT						
1	4	6	9	15	25	36	47	58 … 80
			HEX1	CON	H#123456789	BCDEFG		
			HEX2	CON	H#+12345678	9Δ BCD		

The first example above defines a hexadecimal constant which consists of the 12 four-bit characters in machine language which represent the hexadecimal characters 1, 2, 3, 4, 5, 6, 7, 8, 9, B, C, and D, respectively. Together these form the 48-bit constant which is stored in the location tagged HEX1. The three characters (E, F, and G) following the assembled 12 are dropped, since a hexadecimal constant consists of only one word.

The second example above defines a signed hexadecimal constant which is assembled as +00123456789, since the blank (an illegal character) causes the premature termination of the constant after a sign and nine other hexadecimal characters have been explicitly defined. Observing the sign convention, these nine characters are right justified and the leading character positions are filled with zeros. The constant is stored in the location tagged HEX2.

Octal Constants

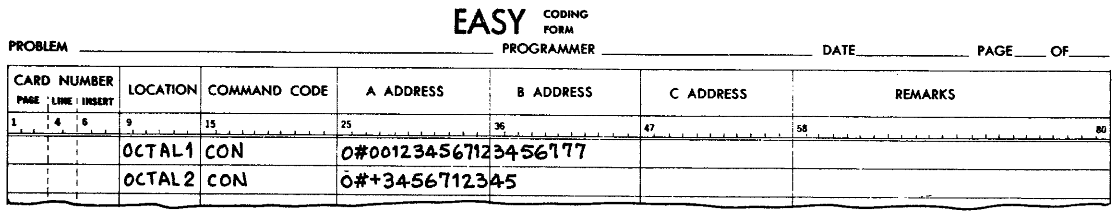
These constants are specified in the command code field and the address fields of the card, ignoring field boundaries, in the form

CON / O#XX...X

where XX...X represent the octal characters the programmer desires for the constant. Only the characters 0-7 are legal. The maximum number of characters which can be defined is 16, and these are assembled and stored as three-bit characters in machine language. When a constant has been prematurely terminated by an illegal character or has not been wholly defined, the unspecified positions in the assembled word are set to zero. The positions of these zeros in the assembled word depend on the following sign convention.

If the first character is a plus (+) or minus (-) sign, the sign is assembled (as four binary ones or four binary zeros, respectively) and stored in the high-order positions of the assembled word. A maximum of a further 15 characters can be explicitly defined by the programmer. (If the maximum number is specified, the first character must not be greater than 3, since there are only 44 bits to represent 15 characters; one of these must be represented in only two bits, and thus takes a maximum value of 3.) If not all 15 are defined, there are no restrictions on the legal values they can take. All explicitly defined characters are right justified in the assembled word, and zeros fill the positions between the sign bits and the assembled characters. The zeros are known as leading zeros.

If the constant is not signed, all explicitly defined characters are left justified in the assembled word, while the remaining positions (if any) are filled with zeros. These zeros are known as trailing zeros.


EASY CODING FORM

PROBLEM ________ PROGRAMMER ________ DATE ________ PAGE ____ OF ____

CARD NUMBER PAGE 1	LINE 4	INSERT 6	LOCATION 9	COMMAND CODE 15	A ADDRESS 25	B ADDRESS 36	C ADDRESS 47	REMARKS 58 … 80
			OCTAL1	CON	0#00123456712	3456777		
			OCTAL2	CON	0#+34567123	45		

The first example above defines an unsigned octal constant which consists of the three-bit representations of the octal digits 0, 0, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, and 7. The 17th and 18th characters (two 7's) are dropped even though they are valid octal characters. The constant is stored in the location tagged OCTAL1.

The second example above defines a signed octal constant which is assembled as +000003456712345. Since the constant is signed, the characters are right justified. The assembled word is stored in the location tagged OCTAL2.

Fixed Binary Constants

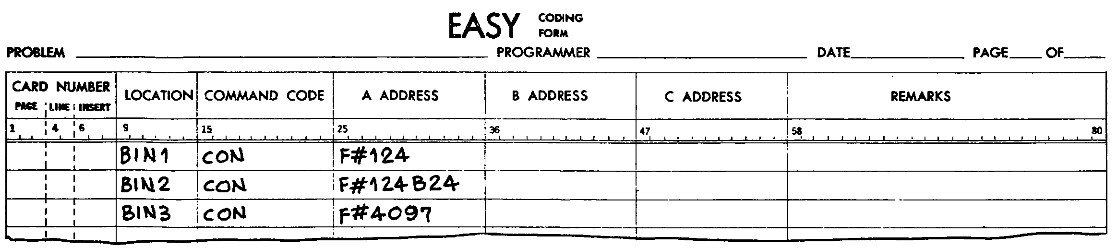
These constants are specified in the command code field and address fields of the card, ignoring field boundaries, in the form

CON / F#XX...X

where XX...X represent the decimal equivalent of the binary configuration the programmer desires for the constant.

The decimal digits may be signed, in which case the sign is assembled (as four binary ones for a plus or four binary zeros for a minus) and stored in the high-order position (bits 1-4) of the constant word. If no sign is specified, the number is assumed to be positive, and four ones are inserted in bits 1 through 4. The largest decimal number which can be represented by the remaining 44 bits is 17, 592, 186, 044, 415 ($=2^{44}-1$). Thus, a programmer can specify a sign and up to 14 decimal digits, provided the value of the number does not exceed the stated maximum for 44 bits. Save for the availability of "B" positioning (see below), the binary configuration is always right justified in the constant word; that is, the unit bit of the configuration is in the low-order position. A fixed binary constant is terminated by the first occurrence of a blank. The characters to the left of this blank are assumed to comprise the whole constant and are assembled as such.

The exception to automatic right justification of the constant is by the use of "B" positioning. If the decimal digits specifying the constant are followed by the letter "B" and then a number in the range 5 through 48, the unit bit of the configuration is positioned in the constant according to that number, and the remaining bits of the configuration are positioned in consecutive positions to the left of the unit bit (i.e., the configuration is right justified within a particular field). The bit positions are numbered from high to low order (left to right), so that position 1 is the high-order position and position 48 is the low-order position. The sign convention described above holds for "B" positioning. The second example below illustrates "B" positioning.



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE	LINE	INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1	4	6	9	15	25	36	47	58 80
			BIN1	CON	F#124			
			BIN2	CON	F#124B24			
			BIN3	CON	F#4097			

The first example above defines a fixed binary constant which consists of the binary configuration of decimal 124 right justified in the constant word. This word is stored in the location tagged BIN1. The configuration in the low-order 12 bits of this word is

000001111100

which is equal to decimal 124. The rest of the word consists of zeros, except for the high-order four bits which are all ones (indicating the assumed plus sign).

The second example above defines exactly the same constant as the first example, except the binary configuration is not right justified within the word itself. The "B" positioning indicates that the unit bit of the configuration is stored in position 24, and so the 12-bit configuration 000001111100 appears in bits 13 through 24 in the constant. The right justification thus occurs in the field consisting of the bit positions 13 through 24. The constant is stored in the location tagged BIN2.

The third example above defines a fixed binary constant which consists of the binary configuration of decimal 4097, right justified in the constant word. This word is stored in the location tagged BIN3. The configuration in the low-order 24 bits of this word is

000000000001000000000001

which is equal to decimal 4097. The rest of this word consists of zeros, except for the high-order four bits which are all ones (indicating the assumed plus sign).

As mentioned under "Arithmetic Instructions", Section IV, binary arithmetic is used mainly for instruction modification. In this case, fixed binary constants must be available to be used as operands in instructions performing such modifications. The example below illustrates one use of fixed binary constants for instruction modification.

EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE______ PAGE____ OF____

CARD NUMBER PAGE | LINE | INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	9	15	25	36	47	58 80
	FIXBIN	CON	F#-40B36			
		SST	FIXBIN	0)-77770000	SHIFT	
	SHIFT	SLB	ALPHA	10	ALPHA	

The fixed binary constant in the first line of coding defines a constant which consists of the binary configuration of decimal 40. This configuration is stored so that the unit bit is in position 36 of the constant word and the rest of the configuration in successive positions to its left. The remaining portions of the constant word consist of binary zeros including four zeros for the sign. The constant is stored in the location tagged FIXBIN.

The SST instruction substitutes the contents of FIXBIN (i. e., the constant) through the mask defined by the octal literal in the B address of the instruction. This mask is such that only bit positions 25 through 36 of the constant are stored in the result location (SHIFT), while the remaining positions in the result location are unaltered. The result location contains the machine instruction word of the binary shift left instruction.

The effect of the SST instruction on the machine instruction word of the shift instruction is to set the bits 25 through 36 (i. e., the B address) to the binary equivalent of decimal 40, while leaving the other 36 bits of the word unaltered. The SLB instruction now specifies a binary shift left of the word at ALPHA, 40 places rather than a shift of 10 places as it originally specified.

Mixed Constants

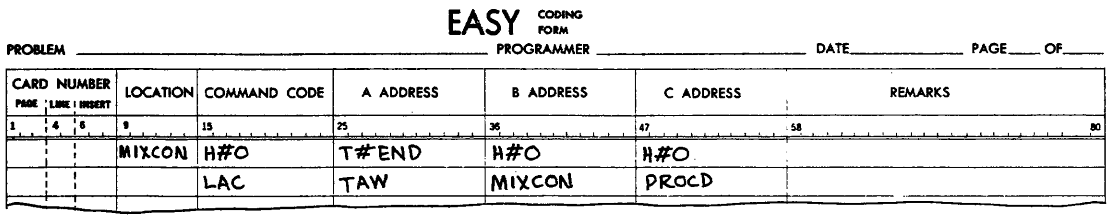
These constants are specified in four expressions, one expression in the command code and one in each of the three address fields of the card, in the form

M#XX...X M#XX...X M#XX...X M#XX...X

Each of the expressions is assembled and stored as 12 bits of the 48-bit constant in memory. The letter "M" stands for the mnemonic code which specifies the type of constant each expression represents, and XX...X stand for the characters appropriate to each expression. The mnemonic code consists of one of the letters "A" (for an alphanumeric expression), "H" (for a hexadecimal expression), "O" for an octal expression), "F" (for a fixed binary expression), or "T". The mnemonic code T indicates that the expression which it precedes is assembled as a tag address; that is, the 12 bits assembled for that expression are the address of the tag XX...X rather than the 12-bit representation of XX...X itself.

Since each expression is represented in memory by 12 bits, characters in excess of two in an alphanumeric expression, three in a hexadecimal expression, four in an octal expression, are dropped. A decimal equivalent of a fixed binary term must be less than 4096. In a "tag" expression (with mnemonic code "T"), the XX...X can be any legal address format except constant or literal, but can not be indexed. In a fixed binary expression within a mixed constant, the unit bit of the binary configuration is in the low-order bit of the corresponding field of the word in memory; "B" positioning is not allowed.

The example below defines a mixed constant which is stored in the location tagged MIXCON. The first, third, and fourth 12-bit portions of this constant are all zeros; the second 12 bits consist of the address assigned to the location tagged END.



EASY CODING FORM

PROBLEM ____ PROGRAMMER ____ DATE ____ PAGE ___ OF ___

CARD NUMBER PAGE LINE INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 4 6	9	15	25	36	47	58 80
	MIXCON	H#O	T#END	H#O	H#O	
		LAC	TAW	MIXCON	PROCD	

The instruction shown above compares the contents of the location tagged TAW (the termination address word) with the contents of the location tagged MIXCON, testing for a less than or equal relationship between them. If the contents of the termination address word are less than or equal to the constant, the sequence changes to the location specified by the tag PROCD.

SECTION VII

LIBRARY ROUTINES AND GENERATORS

The EASY system includes a library of checked out routines which represent frequently-used coding. These routines are available for easy insertion into new programs wherever desired. Each routine in the library is requested by means of a pseudo instruction which specifies the desired routine and all information (parameters) required for its execution. These EASY pseudo instructions are included in a program in exactly the same way as machine instructions. When a program is assembled, EASY recognizes each pseudo instruction, obtains the corresponding coding from the master program tape, and incorporates it into the program. The library section of the master program tape is maintained by the LAMP-PSP program which enables new routines to be added to it and existing routines to be deleted or modified.

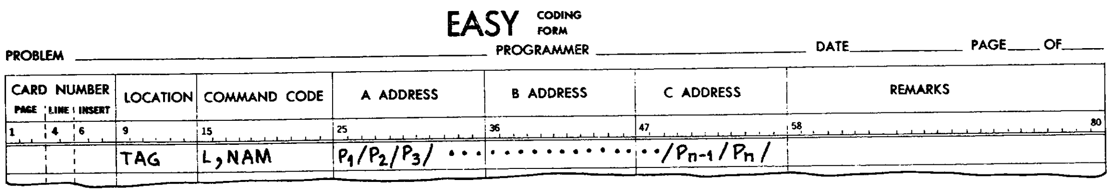
The specifications for every routine are documented on a specification sheet prepared by the programmer who wrote the routine.

Using Library Routines

Routines are stored in the EASY library in the symbolic language in which they are originally written and in generalized form (i. e., with parameters instead of specific values). When a program being assembled requests a routine from the library, the routine is specialized (i. e., particular values are substituted for the parameters) according to the values designated in the pseudo instruction to meet the needs of the requesting program. The routine is inserted in the program at the point where the pseudo instruction was specified.

The programmer must exercise care in using address arithmetic in the vicinity of the pseudo instruction, since address modifiers are not automatically adjusted if coding is inserted.

The pseudo instruction which is used to call a macro routine is known as a call instruction; it has the format shown below.



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE	LINE	INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1	4	6	9	15	25	36	47	58 80
			TAG	L, NAM	$P_1/P_2/P_3/$ ···	············	··$/P_{n-1}/P_n/$	

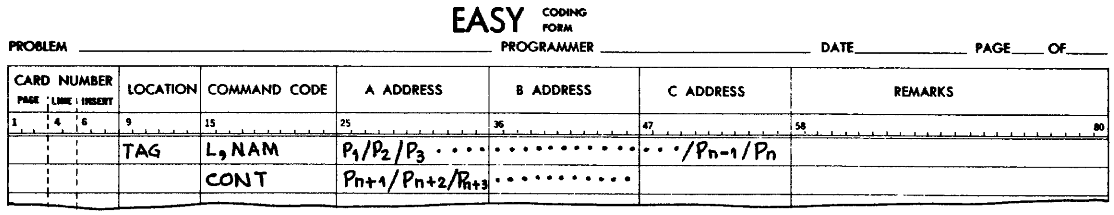
The use of a tag in the location field is optional. If the macro instruction is tagged, EASY Assembly assigns this tag to the first word of the assembled macro routine in the object (i. e., machine language) program. Any reference to this tag within the same program is interpreted as any other tagged location reference (i. e., it refers to the first word of the assembled routine).

The letter "L" in the command code field is a control character which indicates that the following code is the name of a library macro routine. The control character is followed by a comma and a name which specifies the desired routine. This name consists of up to six alphanumeric characters, one of which must be alphabetic. The name is represented here by "NAM".

The codes $p_1/p_2/\dots\dots\dots\dots/p_{n-1}/p_n$ represent the various parameters used in the macro routine, and are punched beginning with the left-most position in the A address field. These parameters specify actual values or the locations of the operands involved in the execution of the routine. The type and format of these parameters can be obtained from the specification sheet.

A call instruction can contain as many parameters as will fit in the address fields; a parameter can consist of up to 16 characters. Individual parameters are separated from each other by slashes and can cross address field boundaries. The parameters may not, however, extend beyond column 57 (i.e., beyond the address fields).

If more parameters are desired than can be specified on one card, "continue" cards may be used to extend the number of parameters which can be specified by a macro instruction. (If such cards are to be used in the EASY 800 system the last valid (non-blank) character on the macro instruction card must be a slash; the last character on a "continue" card must not be a slash.) The code "CONT" is punched in the command code field of a "continue" card, and then further parameters are punched as desired beginning in the high-order position of the A address field. The restrictions on the number and size of parameters in this card(s) are the same as for the macro instruction card itself. The total number of parameters must not be greater than 25. Individual parameters cannot be split between two cards. The "continue" card is illustrated below.



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE	LINE	INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1	4	6	9	15	25	36	47	58 … 80
			TAG	L, NAM	$P_1/P_2/P_3$	. . . .	. . . $/P_{n-1}/P_n$	
				CONT	$P_{n+1}/P_{n+2}/P_{n+3}$	. . . .		

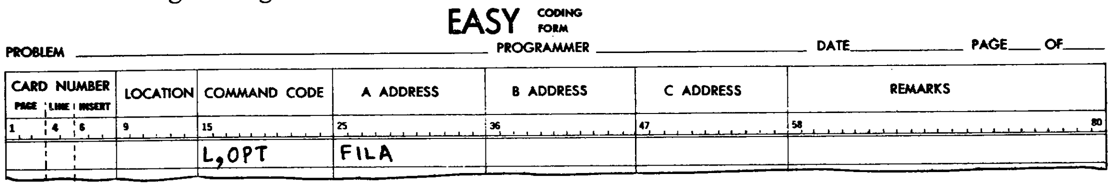
Generators

In addition to the library routines, EASY provides three "generator" routines which are used to handle tape reading and writing, card reading and punching, and printing a report. These routines differ from the library routines in the following ways:

1. The programmer requests one of the generator routines by including a special descriptor card in the input to the Assembly Program. This card contains parameters describing the format of the input or output data.

2. The Assembly program interprets these parameters and generates a single block of instructions capable of performing all appropriate operations on the data described. Assembly inserts this all-encompassing routine at the point where the descriptor card appeared.

3. The programmer then performs input or output operations by including linkage calls in his program. Each linkage call transfers control to a certain portion of the generated routine, performs a specified series of operations, and returns control to the main program. For example, the following linkage call



EASY CODING FORM

PROBLEM ______ PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE | LINE | INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1 | 4 | 6	9	15	25	36	47	58 80
		L,OPT	FILA			

rewinds the tape containing file A, reads the tape label record, checks it against control information, and returns control to the next instruction in the main program.

Detailed specifications on the use of the generators may be found in another publication. In general, the programmer will find the generators useful as a means of freeing himself of the need for concern with controlling peripheral operations and allowing more concentration on the actual processing of data.

SECTION VIII

SAMPLE EASY PROGRAM

To clarify concepts discussed previously, this section describes a sample EASY program. The sample is a relatively short program dealing primarily with peripheral and edit operations. It is shown not as a typical application but because it illustrates a good percentage of the major features of the machine and of the assembly language including: simultaneous processing; use of console breakpoint switches and fixed starts; use of reserved locations and corrective routines; numeric, symbolic, and indexed addressing; literals; constants; and control instructions.

Description of the Sample Program

The sample is basically a program for reading punched cards, punching duplicates, and printing a listing of the cards. To allow variations of the basic program, the programmer has included two "breakpoint options". That is, he has written the program so that the setting of the console breakpoint switches determines whether the cards will simply be punched or whether they will be both punched and printed; in addition, the setting of these switches determines whether new card numbers will be generated for the input cards.

Figure 6 is a block diagram for the sample program. It shows that the main routine starts by reading a card into the card image and then transferring it to the punch image. If breakpoint 2 is off, the card is punched immediately; otherwise, the data in the card number columns is replaced and then the card is punched.

After punching the card, the program tests breakpoint 1: if 1 is set, the program edits the punch image into the print image, prints a line, and then returns to the beginning of the loop and reads the next card; if 1 is not set, the program returns immediately to the first instruction in the loop.

The process is repeated until a special END card is recognized. The program then returns control to Monitor through the EXIT instruction.

Control Instructions

The first lines of coding in Figure 7 are three control instructions. None of these lines produces any machine coding; instead, they direct the assembly of this sample.

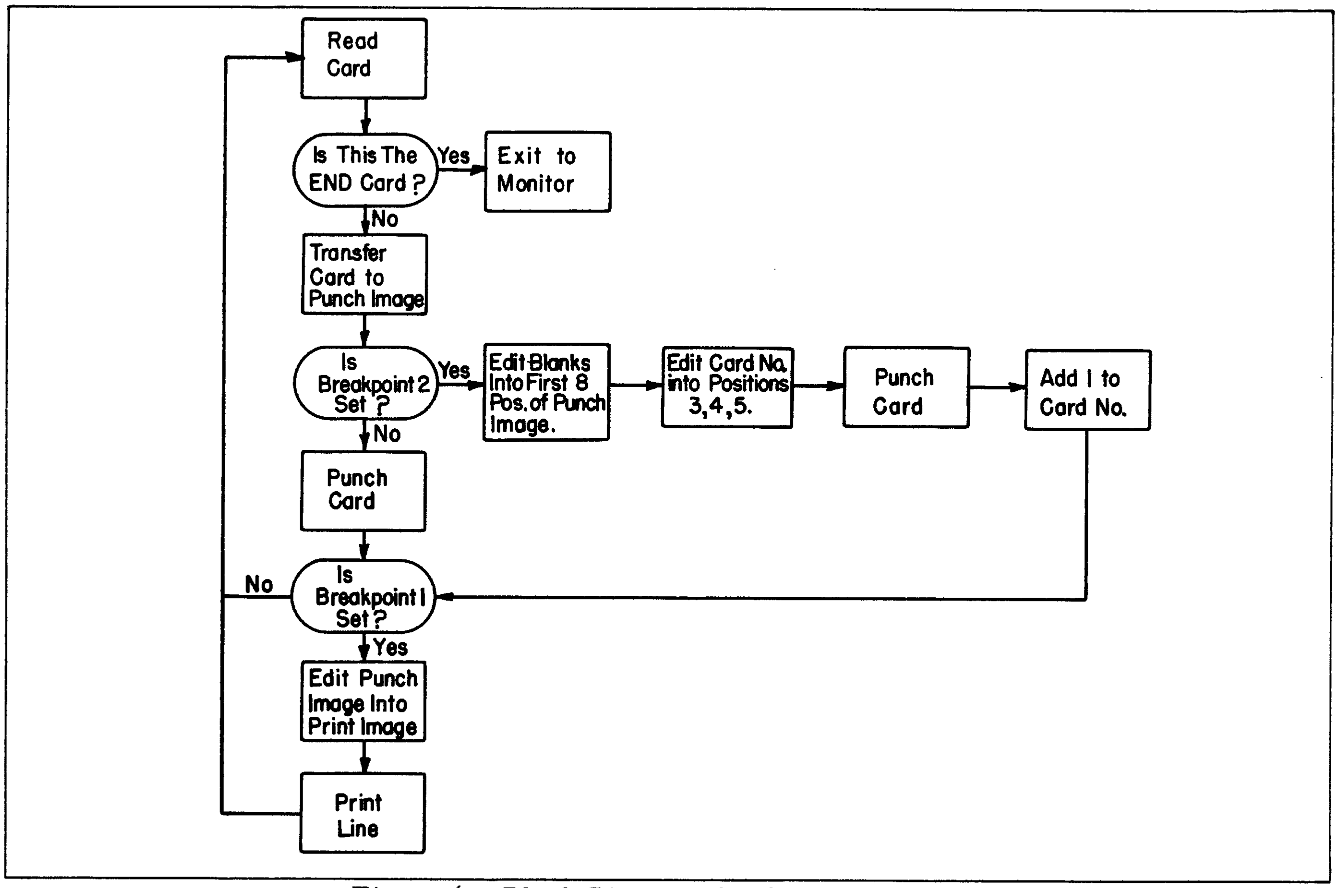


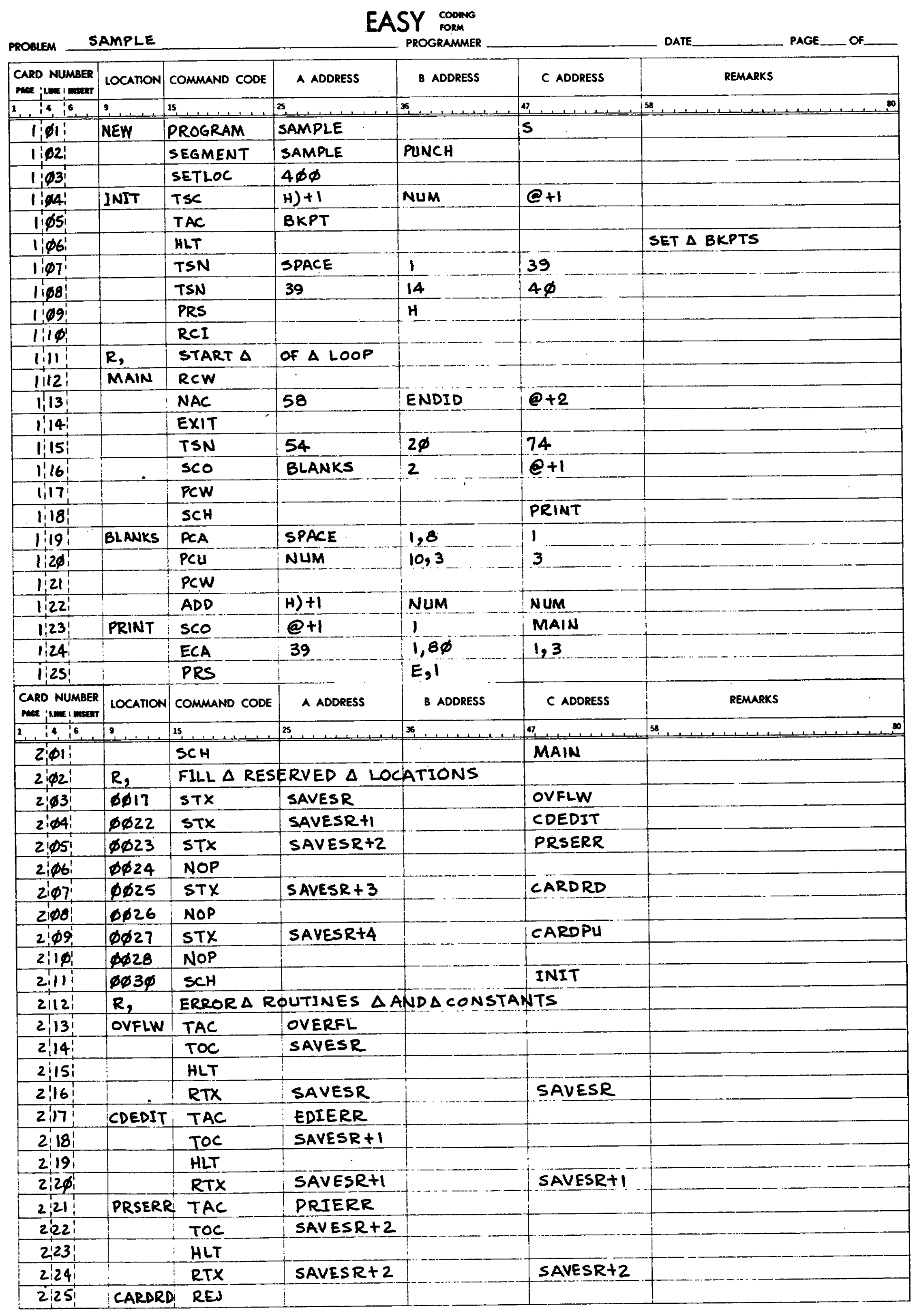
Figure 6. Block Diagram for Sample Program

The PROGRAM and SEGMENT instructions assign the program name "SAMPLE" and the segment name "PUNCH". They also direct Assembly to check the sequence of cards for this program (the "S" in column 47) and to add this program to the master program tape ("NEW" in the location field).

The SETLOC instruction causes Assembly to assign absolute addresses beginning with address 400. If the SETLOC had not been used, the Assembly Program would have assigned addresses beginning with 300, the first location not reserved for use by the hardware or by the Monitor program.

Setting Up a Counter for Card Numbers

The first instruction in the main program is a TSC which sets the value 1 (a literal) into the location tagged NUM and transfers control to the next instruction. The location NUM will be used as a counter for card numbers; that is, when punching new card numbers, the value to be punched will be taken from NUM and NUM will be incremented by one.



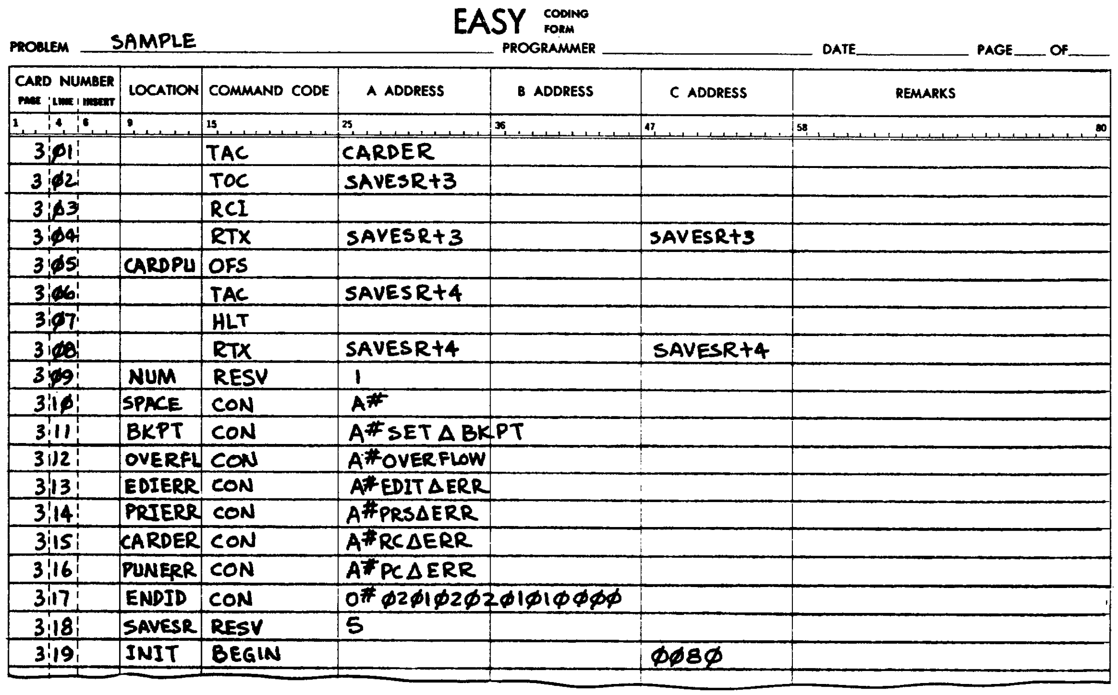
EASY CODING FORM

PROBLEM SAMPLE PROGRAMMER ______ DATE ______ PAGE ___ OF ___

CARD NUMBER PAGE	LINE	INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1	4	6	9	15	25	36	47	58 80
1	Ø1		NEW	PROGRAM	SAMPLE		S	
1	Ø2			SEGMENT	SAMPLE	PUNCH		
1	Ø3			SETLOC	4ØØ			
1	Ø4		INIT	TSC	H)+1	NUM	@+1	
1	Ø5			TAC	BKPT			
1	Ø6			HLT				SET Δ BKPTS
1	Ø7			TSN	SPACE	1	39	
1	Ø8			TSN	39	14	4Ø	
1	Ø9			PRS		H		
1	1Ø			RCI				
1	11		R,	START Δ	OF Δ LOOP			
1	12		MAIN	RCW				
1	13			NAC	58	ENDID	@+2	
1	14			EXIT				
1	15			TSN	54	2Ø	74	
1	16			SCO	BLANKS	2	@+1	
1	17			PCW				
1	18			SCH			PRINT	
1	19		BLANKS	PCA	SPACE	1,8	1	
1	2Ø			PCU	NUM	1Ø,3	3	
1	21			PCW				
1	22			ADD	H)+1	NUM	NUM	
1	23		PRINT	SCO	@+1	1	MAIN	
1	24			ECA	39	1,8Ø	1,3	
1	25			PRS		E,1		

CARD NUMBER PAGE	LINE	INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1	4	6	9	15	25	36	47	58 80
2	Ø1			SCH			MAIN	
2	Ø2		R,	FILL Δ RESERVED Δ LOCATIONS				
2	Ø3		ØØ17	STX	SAVESR		OVFLW	
2	Ø4		ØØ22	STX	SAVESR+1		CDEDIT	
2	Ø5		ØØ23	STX	SAVESR+2		PRSERR	
2	Ø6		ØØ24	NOP				
2	Ø7		ØØ25	STX	SAVESR+3		CARDRD	
2	Ø8		ØØ26	NOP				
2	Ø9		ØØ27	STX	SAVESR+4		CARDPU	
2	1Ø		ØØ28	NOP				
2	11		ØØ3Ø	SCH			INIT	
2	12		R,	ERRORΔ ROUTINES ΔANDΔCONSTANTS				
2	13		OVFLW	TAC	OVERFL			
2	14			TOC	SAVESR			
2	15			HLT				
2	16			RTX	SAVESR		SAVESR	
2	17		CDEDIT	TAC	EDIERR			
2	18			TOC	SAVESR+1			
2	19			HLT				
2	2Ø			RTX	SAVESR+1		SAVESR+1	
2	21		PRSERR	TAC	PRIERR			
2	22			TOC	SAVESR+2			
2	23			HLT				
2	24			RTX	SAVESR+2		SAVESR+2	
2	25		CARDRD	REJ				

Figure 7. Coding for Sample Program



EASY CODING FORM

PROBLEM SAMPLE PROGRAMMER DATE PAGE OF

CARD NUMBER PAGE	LINE	INSERT	LOCATION	COMMAND CODE	A ADDRESS	B ADDRESS	C ADDRESS	REMARKS
1	4	6	9	15	25	36	47	58 80
3	ø1			TAC	CARDER			
3	ø2			TOC	SAVESR+3			
3	ø3			RCI				
3	ø4			RTX	SAVESR+3		SAVESR+3	
3	ø5		CARDPU	OFS				
3	ø6			TAC	SAVESR+4			
3	ø7			HLT				
3	ø8			RTX	SAVESR+4		SAVESR+4	
3	ø9		NUM	RESV	1			
3	1ø		SPACE	CON	A#			
3	11		BKPT	CON	A#SETΔBKPT			
3	12		OVERFL	CON	A#OVERFLOW			
3	13		EDIERR	CON	A#EDITΔERR			
3	14		PRIERR	CON	A#PRSΔERR			
3	15		CARDER	CON	A#RCΔERR			
3	16		PUNERR	CON	A#PCΔERR			
3	17		ENDID	CON	0#ø2ø1ø2ø2ø1ø1øøø			
3	18		SAVESR	RESV	5			
3	19		INIT	BEGIN			øø8ø	

Figure 7 (cont). Coding for Sample Program

Typing a Console Message

The next two instructions (TAC and HLT) type the message "SET BKPT" at the console and halt the system. This halt allows the operator to set the console breakpoint switches. After setting the brekapoints, the operator can press the console START button to resume processing at the next instruction.

The TAC instruction types the word tagged "BKPT" at the console. This word is set up with an alphanumeric constant - a common practice with console messages. Alternatively, a literal could be written in the A address field of the TAC.

Filling the Print Area with Blanks

To insure that data currently in the print image is not printed as part of this program's listing, it is necessary to set the print image to blanks. Here, this is accomplished with two TSN instructions and an alphanumeric constant.

The first TSN fills the first word of the print image with blanks; it does this by transferring the alphanumeric constant of blanks (SPACE) into location 0039. The second TSN fills the remaining 14 words of the print image with blanks by transferring 0039 into 0040, 0040 into 0041, 0041 into 0042, etc.

When the print image is filled with blanks, the PRS instruction moves the form in the printer to the head of the next page (after printing a line blanks).

Reading the First Card

The next instruction, an RCI, reads the first card into the card image area of memory. Significantly, this read operation is interlocked to prevent simultaneous operations. The reason for this is quite obvious: since subsequent instructions process data from this card, they must not be executed until the read operation is completed.

Beginning Simultaneous Operations

The Honeywell 400's capability for simultaneous operations allows the programmer to process data from the first card while reading the second card. Therefore, the next instruction is an RCW which initiates the second read and then releases the central processor to execute the subsequent instructions. These instructions process the first card, clearing the image area before data transfer begins on the second card.

This sequence of read instructions has another advantage: it facilitates setting up a loop for reading and processing cards simultaneously. The RCW is the first instruction in this loop. Each time through the loop, the sample program initiates a new read and processes data from the previous read.

Clearing the Card Read Image Area

During this type of simultaneous operation, the programmer's most immediate concern is to transfer the data from one card out of the image area before the data from the next card arrives in the area. If the card reader is operating at full speed, there is at least 39 milliseconds to accomplish this; if the card reader is operating more slowly, a longer time is available. In this sample, a NAC checks for the END card and a TSN clears the area, transferring the data into the punch image. Since these operations require less than one millisecond, this instruction clears the area without any possibility of interference.

The NAC compares location 58 (the card image for card columns 17-20) to an octal constant representing the card code for "END". If the two are equal, the program ends and control returns to Monitor.

Testing Console Breakpoint 2

The next instruction, the SCO, tests breakpoint 2. If 2 is ON, the sequence counter is reset to the address for BLANKS; otherwise, processing continues in sequence. In effect, this means that if breakpoint 2 is ON, new card numbers will be edited into columns 3, 4, and 5; otherwise, the card will be punched with its present card numbers.

Editing New Line Numbers

The instruction tagged "BLANKS" edits blanks into columns 1 through 8 of the card image area; it does this using a constant of all blanks. The next instruction, a PCU, edits a new card number into columns 3, 4, and 5; this number is obtained from a counter, "NUM", which was originally set up with the value "1" and then incremented each time through the loop. The next two instructions punch the card with the new card number (PCW) and increment the counter (ADD).

Testing Breakpoint 1 and Printing

The instruction tagged "PRINT" tests breakpoint 1. If # 1 is OFF, control immediately returns to MAIN and the loop is repeated. If # 1 is ON, control goes to the next instruction (@ + 1). This instruction, an ECA, edits the 12-bit code used in the punch image to the 6-bit printer code and places the 80 card characters in the first 80 print positions. Then, the PRS prints the card data and spaces a line; finally, the SCH returns control to MAIN and the loop is repeated.

Normally, ECA instructions are used to edit data from the card read area. However, in this case, the card data has been transferred to the punch area and must be edited from that area. To accomplish this, the programmer has indexed the "1" in the C address field by the value 80 (which was set into index register 3 by the BEGIN instruction at the end of the program). This technique causes editing to begin 81 columns from the base of the read image area and thus locates the first character to be edited in the first column of the punch image area.

Instructions for Reserved Memory Locations

To direct the path of the program after an error signal and after completion of a peripheral operation, the programmer stores STX and NOP instructions in the reserved memory locations. In the sample, the STX instructions are stored in the locations associated with an error signal and the NOP instructions are stored in the locations associated with the completion of a peripheral operation. Though appropriate for this program, these assignments do not reflect a general rule; in practice, the "completion" locations frequently contain STX instructions.

The STX instructions serve as the entry to a subroutine; that is, they store the current setting of the index and sequence register and then reset the sequence register to a new address.

This new address is the first instruction in a routine which takes some corrective action and then restores the registers to their original setting. Thus, the program branches to perform the corrective action and then resumes processing where it was interrupted. If an STX were stored in a "completion" location, it could be entrance to a routine which cleared the input area and issued the next peripheral instruction.

The NOP instructions, in effect, ignore the subsequence call. That is, they immediately return control to the next instruction in the main program; they do not affect any of the registers.

Use of a decimal address in the location field (as shown in Figure 7) is the simplest method of assigning instructions to reserved locations. The sample loads only those reserved locations required by this program; the other locations retain their previous settings (for example, the magnetic tape error location retains an entry to the Orthocorrection routine supplied by Monitor).

Console Fixed Starts

Location 0030 is loaded with a SCH which transfers control to the first instruction in the program. This location is associated with console fixed start F 3. Thus, if the operator executes the fixed start 3, he restarts the program from the beginning and resets the counter NUM.

Error Routines and Constants

The final section of the sample program is primarily concerned with the routines for various error conditions and with constants. The BEGIN control instruction appears as the last line of the program.

If an error is detected during a peripheral, edit or arithmetic operation, it is signalled by an unprogrammed subsequence to a reserved memory location. The previous part of the sample program set up these locations to store the index and sequence register and to transfer control to one of the routines in this section. Generally, each of these routines types an error message and the contents of the stored registers at the console, and then halts. For instance, the overflow routine types

```
OVERFLOW
xxxx xxxx xxxx xxxx
```

at the console and halts, where xx...x is the contents of the registers in octal at the time the error signal interrupted processing. If the operator depresses the START button on the console, the registers are restored and processing continues in sequence.

The other routines are arranged in a similar fashion with the exception of the routine for card read errors. This routine rejects the bad card into a special pocket on the reader, types a message at the console, issues an RCI and then returns to continue processing the previous card.

Most constants are used in typing console messages and have been explained previously. The only exception is the ENDID constant which represents the card image of the characters E, N, D, blank.

Two RESERVE instructions are used: one reserves a location for the counter NUM; the other reserves five consecutive locations for the storage of registers after a subsequence call. Both are necessary to allow Assembly to assign memory addresses to these locations.

Finally, the BEGIN instruction sets the registers at the time program is loaded. The sequence counter is set to 400, the address assigned to the tag "INIT". Registers 1 and 2 are set to zeros; register 3 to set to a value of 80.

APPENDIX A

RESERVED MEMORY LOCATIONS

The reserved memory locations can be divided into the following categories: special purpose, unprogrammed and working subsequence, fixed start, and input/output. All reserved locations are directly addressable, though some contain information which is of use only to the computer itself and is of no interest to the programmer.

Special-Purpose Locations

The special-purpose locations store checking and control information, and information relating to previously executed instructions. Except as noted below, the entire 48-bit word is used to store a single piece of data.

The locations are assigned as follows:

Octal	Decimal	
0000	0000	Low-Order Product Word
0001-0011	0001-0009	Multiplication Operation Words
0012	0010	Remainder Word
0013	0011	General-Purpose Word
0014	0012	Check Parity and Select Word
0015	0013	Termination Address Word
0016	0014	Special Check Word for Card Reader
0017	0015	Indirect Tape Address Word
0020	0016	Special Register Word
0025	0021	RDT and WRT Address Word

The low-order product word receives the low-order signed 11 decimal digits of the result of a multiply instruction.

The multiplication operation words contain partial products at the conclusion of the multiply instruction; they are, in general, of no interest to the programmer. Location 0001 is also a general-purpose working location used by the compute orthocount and punch card instructions.

The remainder word receives the signed 11-decimal-digit remainder at the conclusion of a divide instruction.

The general-purpose word is used by the central processor to implement various instructions; its contents are immaterial to the programmer.

The A address portion of the check parity and select word contains the address of the first bad word (i. e., the first word with incorrect parity) detected in a check parity instruction. The C address portion contains the address of the instruction chosen by the selection instruction. The command code portion may be used for any purpose by the programmer; the B address portion is not useable.

The A address portion of the termination address word contains the address of the last word read into memory be a read tape instruction. The C address portion contains the address of the "A" operand which terminated the extended compare instruction. The remaining portions can not be used by the programmer.

The C address portions of the special check word for the card reader functions as part of the hardware; the contents of that portion and the B address portion are unpredictable and of no use to the programmer. The command code and the A address portions are available to the programmer.

The high-order 24 bits of the indirect tape address word specify the correspondence between physical magnetic tape units and logical tape addresses. The 24 bits are divided into eight groups of three bits; each group corresponds to one of the logical addresses 0, 7, 6, 5, 4, 3, 2, or 1 respectively, reading from high-order to low-order. The octal number defined by the three bits in a particular group represents the physical tape unit which is associated with that group. For example, if the high-order three bits of the word are 110, then physical tape unit 6 is addressed at logical tape unit 0 (since the first three bits correspond to the logical address0). The remainder of the word can not be used by the programmer.

The special register word contains the three index registers and sequence register, each in 12-bit form. Index registers 1, 2, and 3 occupy the first three 12-bit portions of the word, respectively; the sequence register occupies the low-order12 bits of the word.

When a read tape instruction is executed, the address one greater than that of the instruction is stored in the A address portions of the RDT and WRT address word. Similarly, whenever a write tape instruction is executed, the address one greater than that of the instruction is stored in the C address portion of the RDT and WRT address word. The contents of the A address and C address portions are unchanged until another read tape or write tape instruction is executed, at which time the new setting of the sequence register increased by 1 is stored in the appropriate portion. The command code and B address portions are available to the programmer.

Unprogrammed Subsequence Locations

Each of the unprogrammed subsequence locations should be set up with an instruction to control the program path after the occurrence of certain events listed below. Normally, these locations will contain an STX instruction as the entry to a subroutine or an NOP instruction to return control to the main program.

Octal	Decimal	
0021	0017	Addition Overflow, Subtraction Overflow, Division Overcapacity
0022	0018	Magnetic Tape Read Error
0023	0019	Magnetic Tape Write Error
0024	0020	End of Magnetic Tape (Writing)
0026	0022	Card Edit Invalid
0027	0023	Print Error
0031	0025	Card Read Error
0033	0027	Card Punch Error
0035	0029	Error on #1 input channel
0037	0031	Error on #2 input channel
0041	0033	Error on #3 input channel
0043	0035	Error on #1 output channel
0045	0037	Error on #2 output channel

Working Subsequence Locations

Each working subsequence location stores an instruction to direct the program path after data transfer has been completed on a peripheral device. These locations are also used in conjunction with the console fixed start operations explained below.

Octal	Decimal	
0030	0024	Printer with option (fixed start 0)
0032	0026	Card Reader operating without interlock (fixed start 1)
0034	0028	Card Punch operating without interlock (fixed start 2)
0036	0030	Device on #1 input channel (fixed start 3)
0040	0032	Device on #2 input channel (fixed start 4)
0042	0034	Device on #3 input channel (fixed start 5)
0044	0036	Device on #1 output channel (fixed start 6)
0046	0038	Device on #2 output channel (fixed start 7)

Fixed-Start Locations

When the central processor is stopped, processing may be resumed at certain locations using special console instructions. For example, the console typein F 0 will cause the next instruction to be selected from decimal location 0024, the location reserved for the printer subsequence call (see above).

There are 10 such fixed-start locations. The fixed starts F 0 through F 7 are identified above; the console instruction F 8 causes a start at location 0094 (octal 0136); the console instruction F 9 causes a start at decimal location 0095 (octal 0137). Fixed starts 8 and 9 are used by the Monitor program and are not generally available to the programmer.

Input/Output Areas

Octal	Decimal	
0047-0065	0039-0053	Printer Area: A print instruction automatically addresses these 15 locations as the source of data to be printed. Each character to be printed is represented in this area by six bits. Thus, the 15 words in the area represent 120 characters, the equivalent of one line of print. The first six-bit group represents the first character in the line; the second group represents the second left-hand character, etc.
0066-0111	0054-0073	Card Read Area: Execution of either of the two card read instructions results in the transfer of card data into this area. Each punch in the card is read as a 1; each no-punch is read as a zero. The area is divided into 12-bit groups corresponding to the 12 punch positions in a card column; thus the first 12-bit group in 0054 corresponds to column 1 of the card being read, the second group corresponds to column 2, etc. There is a 1 in the high-order bit position of the first 12-bit group there is a 9-punch in the first card column; a 0 is stored in the second bit position if there is no 8-punch, etc.
		This area is also used by the console "bootstrap" instruction; this instruction reads the next record on tape into memory beginning at decimal location 0054 and continuing for as many locations as required.
0012-0135	0074-0093	Card Punch Area: Execution of either of the two card punch instructions causes the central processor to punch the contents of this area into a card. The punching is accomplished in a manner similar to reading: each 1 bit causes a punch, each 0 bit causes no punch. The correspondence between bit positions and card columns is the same as described for the card read area.

APPENDIX B

SIMULTANEOUS OPERATIONS

In the Honeywell 400, the processing of certain peripheral instructions occupies the central processor only a portion of the time required to complete these operations. The intervals not required to execute these peripheral instructions may be used by the central processor for other operations. This capability is referred to as the simultaniety of the Honeywell 400.

Simultaniety raises the possibility of interference between instructions. This section discusses the machine logic which prevents interference and points out the programmer's responsibilities if he wishes to process instructions simultaneously.

SIMULTANEOUS PROCESSING RULES FOR THE HIGH-SPEED CARD READER

The cycle time (i. e., the time taken to read one card) of the high-speed card reader operating at a rate of 650 cards per minute is 93 milliseconds. The interval is divided as follows:

Acceleration Interval	-	33 milliseconds (minimum);
Data Transfer Interval	-	54 milliseconds (fixed);
Deceleration Interval	-	6 milliseconds (maximum for full-speed operation).

The acceleration interval lies between the issuance of a read card instruction and the first row impulse. The deceleration interval lies between the completion of data transfer and the issuance of the next card read instruction. These two intervals are variable depending upon when the programmer issues the next read instruction; however, their combined time will equal 39 milliseconds if the card reader is operating at full speed.

Acceleration Interval

This interval is available for simultaneous processing only when the read card instruction is issued "without interlock". The 33-millisecond interval is divided into three smaller intervals. The machine controls and restricts these subintervals as follows:

1. Part One (first 13 milliseconds) - Any instruction may be initiated in this subinterval. However, if any one of the instructions below is issued in this subinterval, the system stalls until data transfer is complete (i. e., until the end of the data transfer interval:

    Read Card (with or without interlock)
    Punch Card (with or without interlock)
    Print (without storage option)
    Type on Console (Alpha, decimal or octal)
    Stall

2. Part Two (from 13th to 31st millisecond) - Any instruction may be initiated in this subinterval. However, initiating one of those instructions listed in part one or one of those listed below stalls the system until the end of the data transfer interval:

   Decimal Multiply
   Decimal Divide
   Read Tape
   Write Tape
   Peripheral Input or Output

3. Part Three (from 31st to 33rd millisecond) - The system is interlocked such that a new instruction can not be initiated during this interval. If system operations are to proceed normally, instructions already in process must terminate within these two milliseconds (programming rules are given below). If an in-process instruction extends beyond these two milliseconds, a program check occurs and the system comes to an immediate stop (i. e., it does not complete either the read card instruction or the instruction in process). This interval is long enough to permit the execution of the instruction normally stored in a memory location reserved for unprogrammed subsequence; thus even if a subsequence call is generated at the end of the acceleration interval it can be recorded for later processing.

   If an instruction stalls the system for any reason described above, it is not executed; instead, the sequence register remains set to the address of the unexecuted instruction. After the end of data transfer and any subsequence that occurs, control returns to the sequence register and processing continues in sequence.

   In the ways described above the machine circuits automatically prevent the initiation of an instruction which would normally continue beyond the acceleration interval. However, the programmer must restrict instructions which could exceed this interval if they are used to process more than a certain number of items. Therefore, the number of words ("n") in the following multi-word instructions should not exceed the number shown below:

TSN	40 words
EXC	20 words
WRT and RDT	50 words for 48K transfer rates
	100 words for 96K transfer rates
	140 words for 133K transfer rates
COC	78 words
CHP	80 words

   If these limits are observed, no error will occur regardless of the point in the acceleration interval at which the instructions are initiated. Some care should also be taken with edit instructions when the number of characters to be edited (n) exceeds its normal values (80 for card editing, 120 for print editing). In this case, the edit instructions should be limited so that their execution time does not exceed 2 milliseconds.

   Another restriction on the use of the acceleration interval is that a read or write instruction should not be addressed to a rewound tape during the acceleration interval. In addition, the first or second record of a Honeywell 800 tape should not be read during this interval.

   The stall instruction may be used to eliminate possible conflicts. For instance, if a stall proceeds a read tape, the read instruction will not be processed until the end of data transfer; thus, no control error can occur.

Deceleration Interval

This six-millisecond interval is available whether the read card instruction is issued with or without interlock. The interval begins at the end of data transfer and ends when the next read card instruction is issued. The characteristics of this interval are:

1. Any instruction may be issued in this interval;
2. The next read card instruction must be issued within this interval to maintain full reading speed.

SIMULTANEOUS PROCESSING RULES FOR CARD PUNCHES

The cycle time of the standard-speed card punch operating at a speed of 100 cards per minute is 600 milliseconds. Similarly, the cycle time of the high-speed punch operating at a rate of 250 cards per minute is 240 milliseconds. The cycle times are divided as follows:

	Standard-Speed Card Punch	High-Speed Punch
Acceleration Interval	91 milliseconds (minimum	55 milliseconds (minimum)
Part I	first 71 milliseconds	first 35 milliseconds
Part II	next 18 milliseconds	next 18 milliseconds
Part III	last 2 milliseconds	last 2 milliseconds
Data Transfer Interval	502 milliseconds (fixed)	178 milliseconds (fixed)
Deceleration Interval	7 milliseconds (maximum for full speed)	7 milliseconds (maximum for full speed)

The acceleration interval lies between the issuance of a punch card instruction and the first row pulse. The deceleration interval lies between the completion of data transfer and the issuance of the next punch instruction. The characteristics of these intervals are similar to those of the card reader; the rules governing multi-word instructions also apply.

SIMULTANEOUS PROCESSING RULES FOR THE PRINTER

The print cycle is divided into two intervals as follows:

Printing	53 milliseconds
Spacing	14 milliseconds for first line plus 8 milliseconds for each additional line

The extent of simultaneous operations possible with the printer depends upon whether the system is equipped with the print storage option. If the system is so equipped, central processor operations of any type are possible during all but 1.29 milliseconds of the printing cycle and during all of the spacing time. There are no restrictions on these operations. If the system does not include the print storage option, simultaneous operations are possible only during the spacing time; again, there are no restrictions.

SIMULTANEOUS TAPE OPERATIONS

Certain tape-handling operations can be performed simultaneously if the instructions are issued in the correct sequence. The rules are listed below:

1. Processing continues in parallel with the actual rewinding or backspacing of tape. There are no restrictions on the instructions which may be performed in this interval except those on other tape instructions as explained in Section IV.
2. If a write instruction is followed immediately by a read instruction, the read and write operations occur simultaneously.

PRIORITY PROCESSING

During interlocked operations, the central processor executed subsequences as they occur. However, during simultaneous operations, several subsequence calls may be pending at the end of data transfer.

The central processor recognizes such multiple demands in ascending order of reserved memory locations (see Appendix A). That is, when several subsequence calls are pending, the central processor performs the single instruction stored in each memory location involved beginning at the lowest numbered such locations. After executing only the instruction stored in a given memory location, the central processor recognizes the next demand and executes the instruction stored in the corresponding reserved memory location.

Once all demands have been recognized, the central processor begins executing subsequences in descending order of reserved locations.

Commonly, the instruction stored in a reserved memory location will be an STX instruction. Thus, when several subsequence calls are pending, the STX instruction in the lowest numbered location stores the setting of the special register word (i. e., the return to the main program) and resets the sequence register to the address of the first instruction in a subroutine. The machine then senses for the next outstanding subsequence call. If the instruction in its reserved location is also an STX, the new setting of the sequence register is stored (effectively, the entry to the previous subroutine is stored) and the sequence register is again reset. This process continues until all outstanding subsequence calls have been recognized.

When all subsequence calls have been recognized, the machine continues processing at the location specified in the C address field of the final STX. Thus, the first subroutine processed is the one associated with the subsequence call for the highest-numbered reserved location.

The final instruction of a subroutine is usually an RTX which restores the sequence register setting stored by the STX for that routine. In this case, each RTX restores the entry to the subroutine associated with the subsequence call for the next lower reserved location, and the RTX for the final subroutine returns control to the main program.

In summary, when STX and RTX instructions are used as described, priority processing has the effect of first recording a string of subroutine entries, then processing the subroutines in descending order of reserved locations, and finally returning control to the main program.

If the instructions STX and RTX are not used as described, the pattern of program execution will not be as described above but will depend on the instructions stored in the reserved location. The machine will still recognize and execute instructions in reserved locations, one after another according to location number.

APPENDIX C

TAPE, FILE, AND RECORD INDENTIFICATION

All of the EASY systems programs, as well as all the tape handling and other standard routines furnished by Honeywell, use certain conventions to identify information recorded on magnetic tape. Every tape is identified by means of a tape label record. The tape label and end-of-information records define, respectively, the beginning and end of useful information on the tape. Each file or program on tape is bounded by beginning and end identification records. Segments are preceded by begin segment identification records. Finally, each record on tape is identified by a banner word as an identification record, a record of program coding, or a data record. The banner word also contains a record count which is used in tape positioning, plus control information if the record is to be printed or punched.

Tape Label Record

The first record on every tape is a tape label. All programs furnished by Honeywell assume the existence of such a record and preserve the first three words of this record. If all programs used at an installation observe this convention, these three words may be used to establish automatic tape handling accounting procedures based upon the identification of the physical reel.

The maximum number of words in a tape label record is 511. In the case of a data file or work tape, care must be exercised in processing this record since its length varies and its structure differs from that of the other records on the tape.

Word 1	Banner Word - This word of the tape label record has the octal configuration SSSS xxxx 0020 xxxx, where the S's represent special octal characters. These four digits represent control information. The next four digits are irrelevant. The contents of bits 28 through 32 identify the record as a tape label. The record count is irrelevant since record counting begins with the second record on tape. (See page 93 for the binary configuration of a banner word.)
Word 2*	Tape Identification.
Word 3*	Unspecified (contents preserved by EASY).
Words 4-11	Unspecified (may be used without restriction).
Words 12 to 12 + n-1	Special Systems Routines.
Words 12 + n to 12 + n+1	Orthowords.

*If the information in these words is in standard alphanumeric code, it will appear in recognizable form if the tape is printed.

File and Program Identification Records

These records are used to identify the beginning and end of each file on a data tape. On a program tape, they are used to identify the beginning and end of each program.

Word 1	Banner Word - Bits 28 through 32 specify the type of information identified by this record (see below).
Word 2	Name of File or Program (up to six alphanumeric characters).
Word 3	Reel Number (two low-order decimal digits) of File Identification Record - The reel number is used primarily for multi-reel files and appears in both the beginning and end file identification records, varying from 01 for the first reel to hex GG for the end identification record of the last reel. The contents of this word are unspecified for a program identification record.
Word 4	Date Obsolete and Date Written (begin file or program records only) - Each date comprises two decimal digits for year, two for month and two for day.
Words 5 to 5 + n-1	File Parameters (sort parameters in file identification records).
Words 5 + n to 5 + n+1	Orthowords.

Segment Identification Records

These records are used to identify the beginning and end of each segment on a program tape.

Word 1	Banner Word - Bits 28 through 32 have the configuration 01001 (see below).
Word 2	Name of Program (six alphanumeric characters).
Word 3	Name of Segment (six alphanumeric characters).
Word 4	Date Obsolete and Date Written (begin segment records only) - Each date comprises two decimal digits for year, two for month, and two for day.
Words 5 to 5 + n-1	Special Parameters.
Words 5 + n to 5 + n+1	Orthowords.

End-of-Information Records

The end-of-information record signals the end of useful information on tape. The last end file identification record should be followed by an end-of-information record and a dummy record. If an additional file is to be stored on the same tape, the end-of-information record must be written over and a new end-of-information record must be written at the end of the new file. However, a program may use the tape area beyond the end-of-information record for work space without having to destroy the end-of-information record.

Word 1	Banner Word - Bits 28 through 32 have the configuration 10001 (see below).
Words 2-4	Unspecified.
Words 5-6	Orthowords.

Banner Words

The first word of every record is a banner word which should contain a record count in bit positions 33 through 48. The record count starts with a value of 1 in the record following the tape label and continues in ascending sequence through all included files to the last record on the tape. Programs which include restart provisions make use of the record count to position tapes. Since the banner word must also serve as a control word on tapes which are to be printed or punched, bit positions 1 through 30 are reserved for control information. The contents of bits positions 31 and 32 specify the type of record which follows the banner word, as follows:

Bits 31-32	00 = printer or punch record
	01 = identification record
	10 = program coding record
	11 = data record

In the case of an identification record, bit positions 28 through 30 are used to specify the type of identification record, as follows:

Bit 28	0 = beginning
	1 = end
Bits 29-30	00 = information
	01 = file or program
	10 = segment
	11 = other

Note that the record type of most records can be determined by examining the contents of banner word bits 31 and 32. If these bits contain the configuration 01, then the contents of bits 28 through 30 must also be examined.

APPENDIX D

INSTRUCTION AND TIMING SUMMARIES

Honeywell 400 instructions are listed below in alphabetic order by mnemonic codes; brief descriptions of the instructions and a summary of their execution times are shown. These times are given in microseconds, except where otherwise stated. If addresses are indexed, 9.25 microseconds must be added to the execution time for each address indexed in that instruction, whether or not the same index register is used more than once. Full descriptions of each instruction can be found in Section IV where they are grouped according to function. The machine language format of the instructions is shown at the end.

The notation A, B, or C stands for the parameter or memory location address specified in the A, B, or C address fields, respectively, of the instruction. Similarly, the notation (A), (B), or (C) stands for the contents of any memory location specified in the A, B, or C address field.

The superscript numbers in the timing summary refer to the notes at the end of the table.

Mnemonic Operation Code	Instruction	Description	Basic Time in Microseconds
ADD	Decimal Add	Adds (A) to (B), stores result in C; treats operands as signed 11 decimal digits.	$111+64.75T^{(1)}$
BAD	Binary Add	Adds (A) to (B), stores result in C; treats operands as unsigned binary numbers.	101.75
BST	Backspace Tape	Backspaces specified magnetic tape by one record.	$-^{(2)}$
BSU	Binary Subtract	Subtracts (B) from (A), stores result in C; treats operands as 48-bit numbers.	101.75
CHP	Check Parity	Checks parity of n words; corrects parity of first bad word then subsequences to C.	92.5+18.5n
COC	Compute Orthotronic Count	Computes the orthocount for n consecutive words, beginning with the word at A. It stores first orthoword in C; second in C + 1.	120.25+18.5n

Mnemonic Operation Code	Instruction	Description	Basic Time in Microseconds
DIV	Decimal Divide	Divides (B) by (A), stores result in C, and stores remainder in remainder word; treats operands as assigned 11 decimal digits.	Avg. 5.374ms; T = 9.25 $[185+8(Q_1+Q_2+\ldots..Q_n)]$ Q = Magnitude of Quotient.
ECA	Card Edit, Alpahnumeric	Edits n consecutive characters of alphanumeric data from card area; stores edited data in memory, beginning with specified position in word at A.	74+11.56n C odd 74+13.87n C even
ECD	Card Edit Signed Decimal	Edits n consecutive characters of decimal data from card area; stores edited data in one word, beginning with specified position in word at A.	83.25+10.8n C odd 83.25+12.34n C even
ECO	Card Edit, Octal	Is the same as ECA, except that data is edited into octal format.	74+10.4n C odd 74+11.56n C even
ECU	Card Edit, Unsigned Decimal	Is the same as ECA, except that data is edited into decimal format.	74+10.5n C odd 74+12.34n C even
EPA	Print Edit, Alphanumeric	Edits n consecutive alphanumeric characters, beginning with the one specified in word at A, into the print area in consecutive positions, beginning with one specified by C.	74+11.56n
EPD	Print Edit, Decimal	Is the same as EPA, except data is edited from decimal format into print area.	74+11.56n
EPO	Print Edit, Octal	Is the same as EPA, except that data is edited from octal format into print area.	74+11.56n
EXC	Extended Compare	Compares (A) with (B), bit by bit, then (A + 1) with (B + 1), etc., until two operands are found unequal. If "A" operand is less than "B", sequence changes to C.	46.25+74n(3)
EXT	Extract	Places (A) in word at C wherever (B) contains a 1 bit; places 0 bits in all other positions in word at C.	111
HAD	Half Add	Adds (A) to (B) without carries; treats operands as unsigned	92.5

Mnemonic Operation Code	Instruction	Description	Basic Time in Microseconds
		binary numbers; stores result in C.	
HLT	Halt	Stops the central processor, depending on the setting of the console breakpoint switches and on (B).	64.75
LAC	Less than or Equal Comparison, Alphanumeric	Compares (A) to (B) bit by bit; sequence changes to (C) if (A) ≤ (B). Otherwise, continues in sequence.	111
LNC	Less than or Equal Comparison, Numeric	Compares (A) and (B); treats operands as signed 11 decimal digit words; sequence changes to C if (A) ≤ (B).	111(4)
LUP	Test Index and Increment	Compares A with contents of index register associated with B. If contents of this index register are less than A, the instruction increments them by B, sequence changes to C.	Jump: IR_i=2: 92.5(5) IR_i=1 or 3: 101.75 No Jump: IR_i=2: 64.75(5) IR_i=1 or 3: 74.0
MPY	Decimal Multiply	Multiplies (A) by (B); treats operands as signed 11 decimal digits; stores result with sign in C, low-order result with sign in low-order product word.	1258+55.5n n = no. of non-zero digits in multiplier
NAC	Inequality Comparison, Alpahnumeric	Compares (A) with (B) bit by bit. If (A) ≠ (B), sequence changes to C.	111
NNC	Inequality Comparison, Numeric	Compares (A) with (B); treats operands as signed 11 decimal digits. If (A) ≠ (B), sequence changes to C.	111(6)
NOP	No Operation	Passes to next instruction, performing no other action.	46.25
OFS	Offset Stack	Offsets the card in the card punch feed so as to protrude slightly from the stack.	92.5 + unit mech. time(7)
PCA	Punch Edit, Alpahnumeric	Edits n consecutive alphanumeric characters, beginning with the one specified in word at A, into the card punch area in consecutive columns, beginning with the one specified by C.	74+13.87n

Mnemonic Operation Code	Instruction	Description	Basic Time in Microseconds
PCD	Punch Edit, Signed Decimal	Is the same as PCA, except that data is edited from decimal format into punch area, and operates only on one word.	$74+13.87n$ for $n \leq 6$ $83.25+13.87n$ for $n > 6$
PCI	Punch Card, Interlocked	Punches the contents of the card punch area onto one card. Central processor interlocked until completion of data transfer.	55.5 + unit mech. time(7)
PCO	Punch Edit, Octal	Is the same as PCA, except that data is edited from octal format into punch area.	$74+13.87n$
PCU	Punch Edit, Unsigned Decimal	Is the same as PCA, except that data is edited from decimal format into card punch area.	$74+13.87n$
PCW	Punch Card, Without Interlock	Punches the contents of the card punch area onto one card. Central processor not interlocked and central processor operations are possible during acceleration interval.	55.5 + unit mech. time(7)
PDE	Prepare Decimal Edit	Inserts special characters, suppresses leading zeros, floats high characters in (A) according to parameters at B. Stores result in (C).	$83.25+18.5n$(8)
PRS	Print and Space	Prints the contents of the print area on the high-speed printer, and spaces the form as specified by B.	Without Storage Option 55.5 + unit mech. time(2) With Storage Option 1193.25
RCI	Read Card, Interlocked	Reads the contents of one card into the card read area. Central processor is interlocked until the completion of data transfer.	55.5 + unit mech. time(7)
RCW	Read Card Without Interlock	Reads the contents of one card into the card read area. Central processor not interlocked and so central processor operations are possible during the acceleration interval.	55.5 + unit mech. time(7)
RDT	Read Tape	Reads one record from the specified magnetic tape and stores in consecutive locations beginning	-(2)

Mnemonic Operation Code	Instruction	Description	Basic Time in Microseconds
		with A. If tape channel is also specified, it regenerates that channel simultaneously.	
REJ	Reject Card	Rejects the card currently in the card feed into one of two pockets as specified in B.	92.5 + unit mech. time(7)
RTX	Restore Index Register	Stores the high-order three 12-bit groups of (A) in the index registers 1, 2, 3, respectively; stores low-order 12 bits of (C) in the sequence register.	83.25
RWT	Rewind Tape	Rewinds the specified magnetic tape to its physical beginning.	92.5 + unit mech time(7)
SCH	Sequence Change	Changes sequence register setting to the address specified by C.	46.25
SCO	Sequence Change on Option	Changes sequence register setting to address specified by A if setting of the console breakpoint switches and (B) coincide. Otherwise set sequence register to the address specified by C.	74
SEL	Select	Modifies C using (A) and (B); then makes a programmed subsequence to the modified address.	120.25
SET	Set Index Register	Adds A into index register specified in Ai and stores result in index register 1; adds B to index register in Bi and stores result in index register 2; adds C to index register specified in Ci and stores result in index register 3.	74
SLB	Binary Shift Left	Shifts (A) to the left the specified number of bits; the move is cyclic, so that the bits shifted off the left end enter the word at the right.	64.75+9.25n(9)
SLP	Decimal Shift Left, Preserving Sign	Shifts (A) to the left n decimal digits preserving the sign digits. Digits shifted off the left end are lost and replaced by zeros at the right end.	64.75+9.25n

Mnemonic Operation Code	Instruction	Description	Basic Time in Microseconds
SMP	Superimpose	Places a 0 bit in all positions of (C) where both (A) and (B) contain 0 bits; places 1 bits in all other positions of (C).	111
SRP	Decimal Shift Right, Preserving Sign	Same as SLP, except that (A) are shifted to the right.	64.75+9.25n
SST	Substitute	Places (A) in (C) in all positions where (B) contains a 1 bit; leaves remaining bit positions in (C) unchanged.	111
STX	Store Index Register	Stores the contents of the three index registers and the sequence register in A. Sets sequence register to C.	83.25
SUB	Decimal Subtract	Subtracts (B) from (A); treats operands as signed 11 decimal digits; stores result in C.	111+64.75T(1)
SUP	Stall	During the acceleration interval of the card reader and punch, this instruction stalls the central processor; outside this interval, it has the effect of NOP.	Stalls till end of data transfer; 64.75 if issued outside of acceleration interval.
TAC	Type Alphanumeric, Console	Prints (A) on the console printer in alphanumeric form.	100-200ms per character
TDC	Type Decimal, Console	Prints (A) on the console printer in decimal form.	100-200ms per character
TOC	Type Octal, Console	Prints (A) on the console printer octal form.	100-200ms per character
TSC	Transfer and Sequence Change	Transfers (A) to location B; sequence changes to location C.	83.25
TSN	Transfer n Words	Transfers n words from consecutive memory locations, beginning with word at A, to consecutive memory locations beginning with C.	46.25+37n
WRT	Write Tape	Writes one record of the specified number of consecutive words from memory, beginning with A, onto tape.	-(2)

NOTES

1. T is derived from the following table:

Signs of operands A	Signs of operands B	T $\|A\| > \|B\|$	T $\|A\| < \|B\|$	
+	+	0	0	Addition
+	-	1	2	
-	-	0	0	
-	+	1	2	
+	+	0	1	Subtraction
+	-	1	1	
-	-	1	1	
-	+	1	1	

2. For Model 404-1, 5.5 ms plus 0.125 n; for Model 404-2, 5.5 ms plus 0.09 n; for Model 404-3, 11.0 ms plus 0.250 n. (n is the number of words read, written or backspaced.)

3. n = number of pairs of words compared.

4. If $|A| = |B|$, and the sign of (A) is positive and the sign of (B) is negative, add 64.75 microseconds.

5. IR_i is the number (i.e., 1, 2, or 3) of the index register associated with the B address. Thus, for a Jump, the time is 101.75 microseconds for index registers 1 and 3, and 92.5 microseconds for index register 2. Similarly, for a No jump, the times are 74 and 64.75 microseconds, respectively.

6. If $|A| = |B|$, and the signs of (A) and (B) are different, add 64.75 microseconds.

7. Mechanical time varies with peripheral equipments and with time at which peripheral order is issued.

8. n = number of non-significant decimal zeros outside of sign position.

   If $6 \leq n < 8$, add 9.25 microseconds; if $n < 6$, add 18.25 microseconds. If p_1 is a plus or minus sign, add 10 microseconds. If p_2 is F (for floating), add 9.25 microseconds.

9. n = number of shifts; $n = \frac{\text{Number of bits shifted}}{4} + \frac{\text{Remainder}}{2} + \frac{\text{Remainder}}{1}$

ARITHMETIC INSTRUCTIONS

ADD	SUB	BAD	BSU	MPY	DIV
32	33	30	31	64	65

1 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	LOCATION OF A OPERAND	LOCATION OF B OPERAND	LOCATION OF C OPERAND (RESULT LOCATION)

LOGICAL INSTRUCTIONS

EXT	SST	HAD	SMP
36	34	37	35

1 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	LOCATION OF A OPERAND	LOCATION OF B OPERAND	LOCATION OF C OPERAND (RESULT LOCATION)

TRANSFER AND SEQUENCE CHANGE INSTRUCTIONS

TSC

1 22 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	LOCATION OF WORD TO BE MOVED	LOCATION TO WHICH WORD IS TO BE MOVED	CHANGE SEQUENCE REGISTER TO

TSN

1 44 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	LOCATION OF FIRST WORD TO BE TRANSFERRED	NUMBER OF WORDS TO BE TRANSFERRED (n) $0 \leq n \leq 4095$	LOCATION TO WHICH FIRST WORD IS TO BE TRANSFERRED

SCH

1 07 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	ØØ		Ci	NOT USED	NOT USED	CHANGE SEQUENCE REGISTER TO

SCO

1 02 6	7 8	9 10	11 12	13 24	25 32	33	34	35	36	37 48
OPERATION CODE	Ai		Ci	IF COINCIDENCE OCCURS CHANGE SEQUENCE REGISTER TO	0 0 0 0 0 0 0 0	4	3	2	1	IF COINCIDENCE DOES NOT OCCUR, CHANGE SEQUENCE REGISTER TO
						BREAKPOINT				

SEL

1 20 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	A OPERAND (SOURCE OF AUGMENT)	B OPERAND	BASE TO WHICH AUGMENT IS ADDED

NOP

1 04 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	00		00	NOT USED	NOT USED	NOT USED

HLT

1 03 6	7 8	9 10	11 12	13 24	25 31	32	33	34	35	36	37 48
OPERATION CODE	00		00	0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0	STOP	4	3	2	1	NOT USED
							BREAKPOINT				

SUP

1 01 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	01	00	LOCATION OF STALL ORDER	0 0 0 0 0 1 0 0 0 0 0 0	NOT USED

LEGEND

Conventions used below are:

"NOT USED" - the address field is not interpreted at execution time.

00 in index bits - these bits normally contain 00 to prevent indexing a "NOT USED" address field

blank index bits - the bits are not interpreted at execution time

EDIT INSTRUCTIONS : Op Code : 12

ECA 00 ECD 10 ECU 11 ECO 01

1 12 6	7 8	9 10	11 12	13 24	25 28	29 36	37 48
OPERATION CODE	Ai	00 10 11 01	Ci	LOCATION OF FIRST DATA WORD AFTER EDITING	m RELATIVE CHARACTER POSITION OF FIRST CHARACTER IN THE WD AT A	n NUMBER OF CHARACTERS TO BE EDITED	FIRST CARD COLUMN TO BE EDITED

Op. Code : 13 Op. Code : 11

EPA 00 EPD 11 EPO 01 PCA 00 PCD 10 PCU 11 PCO 01

1 6	7 8	9 10	11 12	13 24	25 28	29 36	37 48
OPERATION CODE	Ai	00 10 11 01	Ci	WORD IN WHICH FIRST CHARACTER TO BE EDITED IS LOCATED	m RELATIVE CHARACTER POSITION OF FIRST CHARACTER IN WORD AT A	n NUMBER OF CHARACTERS TO BE EDITED	FIRST CARD COLUMN OR PRINT POSITION INTO WHICH DATA IS TO BE EDITED

PDE

1 10 6	7 8	9 10	11 12	13 24	25 28	29 32	33 36	37 48
OPERATION CODE	Ai	Bi	Ci	LOCATION OF WORD TO BE PREPARED FOR EDITING	0 0 0 0	PARAMETER 1	PARAMETER 2	STORAGE LOCATION OF PREPARED WORD

INDEX REGISTER AND CHECK INSTRUCTIONS

STX

1 05 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai		Ci	LOCATION WHERE REGISTERS ARE TO BE STORED	NOT USED	SEQUENCE CHANGE TO

RTX

1 27 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	LOCATION OF 36 BITS TO BE STORED IN INDEX REGISTERS	NOT USED	LOCATION OF 12 BITS TO BE STORED IN THE SEQUENCE REGISTER

SET

1 26 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	BASE VALUE FOR INDEX REGISTER 1	BASE VALUE FOR INDEX REGISTER 2	BASE VALUE FOR INDEX REGISTER 3

LUP

1 06 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	VALUE FOR COMPARISON	INCREMENT	SEQUENCE CHANGE TO

CHP

1 23 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	LOCATION OF FIRST WORD TO BE CHECKED	NUMBER OF WORDS TO CHECK	IF INCORRECT PARITY FOUND, SUBSEQUENCE TO

COC

1 21 6	7 8	9 10	11 12	13 24	25 27	28 36	37 48
OPERATION CODE	Ai	Bi	Ci	FIRST WORD TO BE ORTHOCOUNTED	0 0 0	n NUMBER OF WORDS TO BE ORTHOCOUNTED	LOCATION OF FIRST ORTHOWORD

SHIFT INSTRUCTIONS

SLP	SLB	SRP
47	46	45

1 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	LOCATION OF WORD TO BE SHIFTED	NUMBER OF DIGITS TO BE SHIFTED	STORAGE LOCATION OF SHIFTED WORD

DECISION INSTRUCTIONS

NAC	NNC	EXC	LNC	LAC
52	53	54	51	50

1 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	Ai	Bi	Ci	A OPERAND	B OPERAND	SEQUENCE CHANGE TO

PERIPHERAL INSTRUCTIONS

RDT

1 66 6	7 8	9 10	11 12	13 24	25 27	28 36	37 48
OPERATION CODE	Ai	00	00	LOCATION TO WHICH FIRST WORD OF RECORD IS TO BE READ	t TAPE UNIT	0 0 0 0 0 0 0 0 0 OR CHANNEL TO BE REGENERATED	0 0 0 0 0 0 0 0 0 0 0 0

Bi & Ci MUST BE 00 FOR SIMULTANEOUS READ AND WRITE.

WRT

1 67 6	7 8	9 10	11 12	13 24	25 27	28 36	37 48
OPERATION CODE	Ai	Bi	00	LOCATION OF FIRST WORD TO BE WRITTEN	t TAPE UNIT	n NUMBER OF WORDS ($n \leq 511$)	0 0 0 0 0 0 0 0 0 0 0 0

RWT

1 62 6	7 8	9 10	11 12	13 24	25 27	28 36	37 48
OPERATION CODE	00	00	00	NOT USED	t TAPE UNIT TO BE REWOUND	0 0 0 0 0 0 0 0 0	NOT USED

BST AND PERIPHERAL CONTROL

1 63 6	7 8	9 10	11 12	13 24	25 28	29 36	37 48
OPERATION CODE	00	Bi	00	NOT USED	t TAPE UNIT OR PERIPH. ADDRESS	ALL ZEROS EXCEPT WHEN INSTRUCTION IS USED FOR SPEC. OPERATIONS	NOT USED

TAC 10 TDC 11 TOC 01

1 01 6	7 8	9 10	11 12	13 24	25 35	36	37 48
OPERATION CODE	Ai	10 11 01	Ci	LOCATION OF WORD TO BE PRINTED	0 0 0 0 0 0 0 0 0 0 0	CAR/RET	NOT USED

PRS

1 76 6	7 8	9 10	11 12	13 24	25 28	29 30	31 36	37 48
OPERATION CODE	Ai 00	Bi	Ci 00	NOT USED	0 0 0 0	FOR-MAT	LINE COUNT	NOT USED

74 75 - GENERAL PURPOSE INPUT (74) AND OUTPUT (75)

1 6	7 8	9 10	11 12	13 24	25 28	29 36	37 48
OPERATION CODE	Ai	Bi	Ci	LOCATION WHERE FIRST CHARACTER IS READ INTO OR FROM	DEVICE ADDRESS	n NO. OF CHARACTERS TO BE READ OR PUNCHED	NOT USED

RCI 71 RCW 70 PCI 73 PCW 72

1 6	7 8	9 10	11 12	13 24	25 36	37 48
OPERATION CODE	00	00	00	NOT USED	NOT USED	NOT USED

APPENDIX E

HONEYWELL 400 CODES

Key Punch	Card Code	Honeywell 400 Code	Octal	High Speed Printer	Key Punch	Card Code	Honeywell 400 Code	Octal	High Speed Printer
0	0	000000	00	0	-	X	100000	40	-
1	1	000001	01	1	J	X, 1	100001	41	J
2	2	000010	02	2	K	X, 2	100010	42	K
3	3	000011	03	3	L	X, 3	100011	43	L
4	4	000100	04	4	M	X, 4	100100	44	M
5	5	000101	05	5	N	X, 5	100101	45	N
6	6	000110	06	6	O	X, 6	100110	46	O
7	7	000111	07	7	P	X, 7	100111	47	P
8	8	001000	10	8	Q	X, 8	101000	50	Q
9	9	001001	11	9	R	X, 9	101001	51	R
	8, 2	001010	12	'		X,8,2	101010	52	#
#	8, 3	001011	13	=	$	X,8,3	101011	53	$
@	8, 4	001100	14	:	*	X,8,4	101100	54	*
Space	Blank	001101	15	Blank		X,8,5	101101	55	"
	8, 6	001110	16	Blank*		X,8,6	101110	56	Blank*
	8, 7	001111	17	&		X, 0	101111	57	Blank*
&	R	010000	20	+		8, 5*	110000	60	Blank*
A	R, 1	010001	21	A	/	0, 1	110001	61	/
B	R, 2	010010	22	B	S	0, 2	110010	62	S
C	R, 3	010011	23	C	T	0, 3	110011	63	T
D	R, 4	010100	24	D	U	0, 4	110100	64	U
E	R, 5	010101	25	E	V	0, 5	110101	65	V
F	R, 6	010110	26	F	W	0, 6	110110	66	W
G	R, 7	010111	27	G	X	0, 7	110111	67	X
H	R, 8	011000	30	H	Y	0, 8	111000	70	Y
I	R, 9	011001	31	I	Z	0, 9	111001	71	Z
	R, 8, 2	011010	32	;		0,8,2	111010	72	@
•	R, 8, 3	011011	33	.	,	0,8,3	111011	73	,
□	R, 8, 4	011100	34	)	%	0,8,4	111100	74	(
	R, 8, 5	011101	35	%		0,8,5	111101	75	CR
	R, 8, 6	011110	36	■		0,8,6	111110	76	Blank*
	R, 0	011111	37	Blank*		0,8,7	111111	77	Blank*

Notes:	Key Punch:	Use MLT PCH key to overpunch omitted characters.
	Printer:	* indicates symbol which will be printed by otherwise non-standard printer bit configuration.

Figure E-1. Honeywell 400 Coding and Punched or Printed Equivalents

INDEX

References below are to page numbers of the basic discussion of each item.

Honeywell

H HONEYWELL Electronic Data Processing